Learning Ada

The complete contents of learn.adacore.com
Learning Ada

Release 2024-06

Various authors

Jun 30, 2024
## CONTENTS

### 1 Introduction to Ada

#### 1 Introduction
1.1 History ............................................. 7
1.2 Ada today ......................................... 7
1.3 Philosophy ......................................... 8
1.4 SPARK ............................................. 8

#### 2 Imperative language
2.1 Hello world ........................................ 9
2.2 Imperative language - If/Then/Else ............... 11
2.3 Imperative language - Loops ...................... 13
   2.3.1 For loops .................................... 13
   2.3.2 Bare loops .................................. 15
   2.3.3 While loops ................................ 16
2.4 Imperative language - Case statement ............ 16
2.5 Imperative language - Declarative regions ....... 18
2.6 Imperative language - conditional expressions... 20
   2.6.1 If expressions ............................... 20
   2.6.2 Case expressions ............................ 21

#### 3 Subprograms
3.1 Subprograms ...................................... 23
   3.1.1 Subprogram calls ......................... 24
   3.1.2 Nested subprograms ...................... 26
   3.1.3 Function calls ............................. 27
3.2 Parameter modes .................................. 28
3.3 Subprogram calls ................................ 28
   3.3.1 In parameters ............................... 28
   3.3.2 In out parameters ......................... 29
   3.3.3 Out parameters ............................. 30
   3.3.4 Forward declaration of subprograms ... 31
3.4 Renaming ......................................... 32

#### 4 Modular programming
4.1 Packages .......................................... 35
4.2 Using a package .................................. 37
4.3 Package body ...................................... 37
4.4 Child packages .................................. 39
   4.4.1 Child of a child package ............... 41
   4.4.2 Multiple children ......................... 42
   4.4.3 Visibility .................................. 43
4.5 Renaming ......................................... 46

#### 5 Strongly typed language
5.1 What is a type? .................................. 47
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Generics</td>
<td>123</td>
</tr>
<tr>
<td>13.1</td>
<td>Introduction</td>
<td>123</td>
</tr>
<tr>
<td>13.2</td>
<td>Formal type declaration</td>
<td>123</td>
</tr>
<tr>
<td>13.3</td>
<td>Formal object declaration</td>
<td>124</td>
</tr>
<tr>
<td>13.4</td>
<td>Generic body definition</td>
<td>125</td>
</tr>
<tr>
<td>13.5</td>
<td>Generic instantiation</td>
<td>125</td>
</tr>
<tr>
<td>13.6</td>
<td>Generic packages</td>
<td>126</td>
</tr>
<tr>
<td>13.7</td>
<td>Formal subprograms</td>
<td>128</td>
</tr>
<tr>
<td>13.8</td>
<td>Example: I/O instances</td>
<td>129</td>
</tr>
<tr>
<td>13.9</td>
<td>Example: ADTs</td>
<td>132</td>
</tr>
<tr>
<td>13.10</td>
<td>Example: Swap</td>
<td>133</td>
</tr>
<tr>
<td>13.11</td>
<td>Example: Reversing</td>
<td>136</td>
</tr>
<tr>
<td>13.12</td>
<td>Example: Test application</td>
<td>140</td>
</tr>
<tr>
<td>14</td>
<td>Exceptions</td>
<td>143</td>
</tr>
<tr>
<td>14.1</td>
<td>Exception declaration</td>
<td>143</td>
</tr>
<tr>
<td>14.2</td>
<td>Raising an exception</td>
<td>143</td>
</tr>
<tr>
<td>14.3</td>
<td>Handling an exception</td>
<td>144</td>
</tr>
<tr>
<td>14.4</td>
<td>Predefined exceptions</td>
<td>146</td>
</tr>
<tr>
<td>15</td>
<td>Tasking</td>
<td>147</td>
</tr>
<tr>
<td>15.1</td>
<td>Tasks</td>
<td>147</td>
</tr>
<tr>
<td>15.1.1</td>
<td>Simple task</td>
<td>147</td>
</tr>
<tr>
<td>15.1.2</td>
<td>Simple synchronization</td>
<td>149</td>
</tr>
<tr>
<td>15.1.3</td>
<td>Delay</td>
<td>151</td>
</tr>
<tr>
<td>15.1.4</td>
<td>Synchronization: rendezvous</td>
<td>151</td>
</tr>
<tr>
<td>15.1.5</td>
<td>Select loop</td>
<td>153</td>
</tr>
<tr>
<td>15.1.6</td>
<td>Cycling tasks</td>
<td>154</td>
</tr>
<tr>
<td>15.2</td>
<td>Protected objects</td>
<td>158</td>
</tr>
<tr>
<td>15.2.1</td>
<td>Simple object</td>
<td>158</td>
</tr>
<tr>
<td>15.2.2</td>
<td>Entries</td>
<td>159</td>
</tr>
<tr>
<td>15.3</td>
<td>Task and protected types</td>
<td>161</td>
</tr>
<tr>
<td>15.3.1</td>
<td>Task types</td>
<td>161</td>
</tr>
<tr>
<td>15.3.2</td>
<td>Protected types</td>
<td>163</td>
</tr>
<tr>
<td>16</td>
<td>Design by contracts</td>
<td>165</td>
</tr>
<tr>
<td>16.1</td>
<td>Pre- and postconditions</td>
<td>165</td>
</tr>
<tr>
<td>16.2</td>
<td>Predicates</td>
<td>168</td>
</tr>
<tr>
<td>16.3</td>
<td>Type invariants</td>
<td>172</td>
</tr>
<tr>
<td>17</td>
<td>Interfacing with C</td>
<td>175</td>
</tr>
<tr>
<td>17.1</td>
<td>Multi-language project</td>
<td>175</td>
</tr>
<tr>
<td>17.2</td>
<td>Type convention</td>
<td>175</td>
</tr>
<tr>
<td>17.3</td>
<td>Foreign subprograms</td>
<td>176</td>
</tr>
<tr>
<td>17.3.1</td>
<td>Calling C subprograms in Ada</td>
<td>176</td>
</tr>
<tr>
<td>17.3.2</td>
<td>Calling Ada subprograms in C</td>
<td>178</td>
</tr>
<tr>
<td>17.4</td>
<td>Foreign variables</td>
<td>179</td>
</tr>
<tr>
<td>17.4.1</td>
<td>Using Ada global variables in C</td>
<td>179</td>
</tr>
<tr>
<td>17.4.2</td>
<td>Using Ada variables in C</td>
<td>181</td>
</tr>
<tr>
<td>17.5</td>
<td>Generating bindings</td>
<td>182</td>
</tr>
<tr>
<td>17.5.1</td>
<td>Adapting bindings</td>
<td>184</td>
</tr>
<tr>
<td>18</td>
<td>Object-oriented programming</td>
<td>189</td>
</tr>
<tr>
<td>18.1</td>
<td>Derived types</td>
<td>190</td>
</tr>
<tr>
<td>18.2</td>
<td>Tagged types</td>
<td>191</td>
</tr>
<tr>
<td>18.3</td>
<td>Classwide types</td>
<td>193</td>
</tr>
<tr>
<td>18.4</td>
<td>Dispatching operations</td>
<td>194</td>
</tr>
<tr>
<td>18.5</td>
<td>Dot notation</td>
<td>196</td>
</tr>
<tr>
<td>18.6</td>
<td>Private &amp; Limited</td>
<td>197</td>
</tr>
</tbody>
</table>
26.2.6 Extended return statement .................................................. 615
26.3 Subprograms ............................................................................. 618
  26.3.1 Parameter Modes and Associations ........................................ 618
  26.3.2 Operators ........................................................................... 631
  26.3.3 Expression functions ............................................................. 637
  26.3.4 Overloading ......................................................................... 640
  26.3.5 Operator Overloading ............................................................. 645
  26.3.6 Operator Overriding ............................................................... 645
  26.3.7 Nonreturning procedures ....................................................... 648
  26.3.8 Inline subprograms ............................................................... 651
  26.3.9 Null Procedures ................................................................ 653
26.4 Exceptions .................................................................................. 656
  26.4.1 Asserts ................................................................................ 656
  26.4.2 Assertion policies ................................................................. 658
  26.4.3 Checks and exceptions .......................................................... 661
  26.4.4 Ada.Exceptions package ....................................................... 674
  26.4.5 Exception renaming .............................................................. 682
  26.4.6 Out and Uninitialized ............................................................ 684
  26.4.7 Suppressing checks .............................................................. 688

27 Modular programming .................................................................. 693
  27.1 Packages ................................................................................. 693
    27.1.1 Package renaming .............................................................. 693
    27.1.2 Private packages .............................................................. 696
    27.1.3 Private with clauses ........................................................... 704
    27.1.4 Limited Visibility ............................................................... 708
    27.1.5 Visibility ......................................................................... 712
    27.1.6 Use type clause ................................................................. 720
    27.1.7 Use clauses and naming conflicts ......................................... 724
  27.2 Subprograms and Modularity .................................................... 729
    27.2.1 Private subprograms .......................................................... 729

28 Resource Management ................................................................. 735
  28.1 Access Types ........................................................................... 735
    28.1.1 Access types: Terminology ................................................ 735
    28.1.2 Access types: Allocation .................................................... 737
    28.1.3 Discriminants as Access Values ......................................... 745
    28.1.4 Parameters as Access Values ............................................. 752
    28.1.5 Self-reference ................................................................... 762
    28.1.6 Mutually dependent types using access types ..................... 765
    28.1.7 Dereferencing ................................................................. 766
    28.1.8 Ragged arrays .................................................................. 772
    28.1.9 Aliasing ........................................................................... 777
    28.1.10 Accessibility Levels and Rules: An Introduction .................. 788
    28.1.11 Unchecked Access ........................................................... 798
    28.1.12 Unchecked Deallocation .................................................. 800
    28.1.13 Null & Not Null Access .................................................... 808
    28.1.14 Design strategies for access types ....................................... 812
    28.1.15 Access to subprograms .................................................... 820
    28.1.16 Accessibility Rules and Access-To-Subprograms .................. 845
    28.1.17 Access and Address ......................................................... 849
  28.2 Anonymous Access Types ......................................................... 853
    28.2.1 Named and Anonymous Access Types ............................... 853
    28.2.2 Anonymous Access-To-Object Types ................................. 858
    28.2.3 Access discriminants ......................................................... 868
    28.2.4 Self-reference ................................................................. 874
    28.2.5 Mutually dependent types using anonymous access types .... 877
    28.2.6 Access parameters ............................................................ 877
49 Enumeration representation 1261
  49.1 Literal positions ........................................... 1261
  49.2 Representation values .................................... 1262
  49.3 Before Ada 2022 ......................................... 1263
  49.4 References .............................................. 1264

50 Big Numbers 1265
  50.1 Big Integers ............................................... 1265
  50.2 Tiny RSA implementation ................................. 1265
  50.3 Big Reals ................................................ 1267
  50.4 References .............................................. 1268

51 Interfacing C variadic functions 1269
  51.1 References .............................................. 1271

VI Ada for the C++ or Java Developer 1273
52 Preface 1277
53 Basics 1279
54 Compilation Unit Structure 1281

55 Statements, Declarations, and Control Structures 1283
  55.1 Statements and Declarations ............................ 1283
  55.2 Conditions ............................................... 1285
  55.3 Loops .................................................... 1286

56 Type System 1289
  56.1 Strong Typing ............................................ 1289
  56.2 Language-Defined Types .................................. 1290
  56.3 Application-Defined Types ............................... 1290
  56.4 Type Ranges ............................................. 1292
  56.5 Generalized Type Contracts: Subtype Predicates ......... 1293
  56.6 Attributes ............................................... 1293
  56.7 Arrays and Strings ...................................... 1294
  56.8 Heterogeneous Data Structures ......................... 1297
  56.9 Pointers .................................................. 1298

57 Functions and Procedures 1303
  57.1 General Form ............................................ 1303
  57.2 Overloading .............................................. 1305
  57.3 Subprogram Contracts .................................... 1305

58 Packages 1307
  58.1 Declaration Protection .................................... 1307
  58.2 Hierarchical Packages .................................... 1308
  58.3 Using Entities from Packages ............................ 1308

59 Classes and Object Oriented Programming 1311
  59.1 Primitive Subprograms .................................... 1311
  59.2 Derivation and Dynamic Dispatch ....................... 1312
  59.3 Constructors and Destructors ............................ 1315
  59.4 Encapsulation ............................................ 1316
  59.5 Abstract Types and Interfaces ......................... 1316
  59.6 Invariants ............................................... 1318

60 Generics 1321
  60.1 Generic Subprograms .................................... 1321
60.2 Generic Packages .................................................. 1322
60.3 Generic Parameters .............................................. 1323

61 Exceptions ........................................................................ 1325
61.1 Standard Exceptions ................................................. 1325
61.2 Custom Exceptions .................................................. 1326

62 Concurrency ..................................................................... 1327
62.1 Tasks ........................................................................... 1327
62.2 Rendezvous .................................................................... 1330
62.3 Selective Rendezvous .................................................. 1332
62.4 Protected Objects ..................................................... 1333

63 Low Level Programming .................................................. 1337
63.1 Representation Clauses ................................................. 1337
63.2 Embedded Assembly Code ........................................... 1338
63.3 Interfacing with C ........................................................ 1339

64 Conclusion ....................................................................... 1341

65 References ....................................................................... 1343

VII Ada for the Embedded C Developer .................................... 1345

66 Introduction ..................................................................... 1349
66.1 So, what is this Ada thing anyway? ................................. 1349
66.2 Ada — The Technical Details ......................................... 1351

67 The C Developer’s Perspective on Ada .................................. 1353
67.1 What we mean by Embedded Software ............................. 1353
67.2 The GNAT Toolchain ................................................... 1353
67.3 The GNAT Toolchain for Embedded Targets .................. 1354
67.4 Hello World in Ada ..................................................... 1355
67.5 The Ada Syntax .......................................................... 1356
67.6 Compilation Unit Structure .......................................... 1357
67.7 Packages ...................................................................... 1357
   67.7.1 Declaration Protection ....................................... 1357
   67.7.2 Hierarchical Packages ......................................... 1358
   67.7.3 Using Entities from Packages ................................. 1359
67.8 Statements and Declarations ........................................ 1359
67.9 Conditions .................................................................... 1365
67.10 Loops .......................................................................... 1369
67.11 Type System .............................................................. 1376
   67.11.1 Strong Typing ..................................................... 1376
   67.11.2 Language-Defined Types ..................................... 1379
   67.11.3 Application-Defined Types ................................. 1379
   67.11.4 Type Ranges ...................................................... 1382
   67.11.5 Unsigned And Modular Types ......................... 1385
   67.11.6 Attributes ......................................................... 1389
   67.11.7 Arrays and Strings ............................................ 1391
   67.11.8 Heterogeneous Data Structures ......................... 1398
   67.11.9 Pointers ........................................................... 1400
67.12 Functions and Procedures ............................................ 1405
   67.12.1 General Form .................................................... 1405
   67.12.2 Overloading ....................................................... 1409
   67.12.3 Aspects ............................................................ 1411

68 Concurrency and Real-Time ............................................. 1415
68.1 Understanding the various options ................................. 1415
## 69 Writing Ada on Embedded Systems

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>69.1 Understanding the Ada Run-Time</td>
<td>1429</td>
</tr>
<tr>
<td>69.2 Low Level Programming</td>
<td>1430</td>
</tr>
<tr>
<td>69.2.1 Representation Clauses</td>
<td>1430</td>
</tr>
<tr>
<td>69.2.2 Embedded Assembly Code</td>
<td>1431</td>
</tr>
<tr>
<td>69.3 Interrupt Handling</td>
<td>1432</td>
</tr>
<tr>
<td>69.4 Dealing with Absence of FPU with Fixed Point</td>
<td>1434</td>
</tr>
<tr>
<td>69.5 Volatile and Atomic data</td>
<td>1439</td>
</tr>
<tr>
<td>69.5.1 Volatile</td>
<td>1439</td>
</tr>
<tr>
<td>69.5.2 Atomic</td>
<td>1441</td>
</tr>
<tr>
<td>69.6 Interfacing with Devices</td>
<td>1443</td>
</tr>
<tr>
<td>69.6.1 Size aspect and attribute</td>
<td>1443</td>
</tr>
<tr>
<td>69.6.2 Register overlays</td>
<td>1444</td>
</tr>
<tr>
<td>69.6.3 Data streams</td>
<td>1447</td>
</tr>
<tr>
<td>69.7 ARM and svd2ada</td>
<td>1453</td>
</tr>
</tbody>
</table>

## 70 Enhancing Verification with SPARK and Ada

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>70.1 Understanding Exceptions and Dynamic Checks</td>
<td>1455</td>
</tr>
<tr>
<td>70.2 Understanding Dynamic Checks versus Formal Proof</td>
<td>1462</td>
</tr>
<tr>
<td>70.3 Initialization and Correct Data Flow</td>
<td>1465</td>
</tr>
<tr>
<td>70.4 Contract-Based Programming</td>
<td>1466</td>
</tr>
<tr>
<td>70.5 Replacing Defensive Code</td>
<td>1469</td>
</tr>
<tr>
<td>70.6 Proving Absence of Run-Time Errors</td>
<td>1471</td>
</tr>
<tr>
<td>70.7 Proving Abstract Properties</td>
<td>1472</td>
</tr>
<tr>
<td>70.8 Final Comments</td>
<td>1473</td>
</tr>
</tbody>
</table>

## 71 C to Ada Translation Patterns

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>71.1 Naming conventions and casing considerations</td>
<td>1475</td>
</tr>
<tr>
<td>71.2 Manually interfacing C and Ada</td>
<td>1475</td>
</tr>
<tr>
<td>71.3 Building and Debugging mixed language code</td>
<td>1477</td>
</tr>
<tr>
<td>71.4 Automatic interfacing</td>
<td>1478</td>
</tr>
<tr>
<td>71.5 Using Arrays in C interfaces</td>
<td>1478</td>
</tr>
<tr>
<td>71.6 By-value vs. by-reference types</td>
<td>1481</td>
</tr>
<tr>
<td>71.7 Naming and prefixes</td>
<td>1482</td>
</tr>
<tr>
<td>71.8 Pointers</td>
<td>1483</td>
</tr>
<tr>
<td>71.9 Bitwise Operations</td>
<td>1486</td>
</tr>
<tr>
<td>71.10 Mapping Structures to Bit-Fields</td>
<td>1488</td>
</tr>
<tr>
<td>71.10.1 Overlays vs. Unchecked Conversions</td>
<td>1502</td>
</tr>
</tbody>
</table>

## 72 Handling Variability and Re-usability

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>72.1 Understanding static and dynamic variability</td>
<td>1507</td>
</tr>
<tr>
<td>72.2 Handling variability &amp; reusability statically</td>
<td>1507</td>
</tr>
<tr>
<td>72.2.1 Genericity</td>
<td>1507</td>
</tr>
<tr>
<td>72.2.2 Simple derivation</td>
<td>1511</td>
</tr>
<tr>
<td>72.2.3 Configuration pragma files</td>
<td>1516</td>
</tr>
<tr>
<td>72.2.4 Configuration packages</td>
<td>1518</td>
</tr>
<tr>
<td>72.3 Handling variability &amp; reusability dynamically</td>
<td>1522</td>
</tr>
<tr>
<td>72.3.1 Records with discriminants</td>
<td>1522</td>
</tr>
<tr>
<td>72.3.2 Variant records</td>
<td>1524</td>
</tr>
<tr>
<td>72.3.3 Object orientation</td>
<td>1532</td>
</tr>
<tr>
<td>72.3.4 Pointer to subprograms</td>
<td>1545</td>
</tr>
<tr>
<td>72.4 Design by components using dynamic libraries</td>
<td>1551</td>
</tr>
</tbody>
</table>
## 90 GNAT Tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>90.1 gnatchop</td>
<td>1702</td>
</tr>
<tr>
<td>90.2 gnatprep</td>
<td>1704</td>
</tr>
<tr>
<td>90.3 gnatmem</td>
<td>1705</td>
</tr>
<tr>
<td>90.4 gnatmetric</td>
<td>1707</td>
</tr>
<tr>
<td>90.5 gnatdoc</td>
<td>1708</td>
</tr>
<tr>
<td>90.6 gnatpp</td>
<td></td>
</tr>
<tr>
<td>90.7 gnatstub</td>
<td></td>
</tr>
</tbody>
</table>

## X Guidelines for Safe and Secure Ada/SPARK

### 91 Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>91.1 Scope</td>
<td>1713</td>
</tr>
<tr>
<td>91.2 Structure</td>
<td>1713</td>
</tr>
<tr>
<td>91.3 Enforcement</td>
<td>1714</td>
</tr>
<tr>
<td>91.4 About the Rules</td>
<td>1714</td>
</tr>
</tbody>
</table>

### 92 Definitions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>92.1 Level</td>
<td>1715</td>
</tr>
<tr>
<td>92.2 Remediation</td>
<td>1715</td>
</tr>
</tbody>
</table>

### 93 Dynamic Storage Management (DYN)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>93.1 Common High Integrity Restrictions (DYN01)</td>
<td>1718</td>
</tr>
<tr>
<td>93.1.1 Reference</td>
<td>1718</td>
</tr>
<tr>
<td>93.1.2 Description</td>
<td>1718</td>
</tr>
<tr>
<td>93.1.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1719</td>
</tr>
<tr>
<td>93.1.4 Noncompliant Code Example</td>
<td>1719</td>
</tr>
<tr>
<td>93.1.5 Compliant Code Example</td>
<td>1719</td>
</tr>
<tr>
<td>93.1.6 Notes</td>
<td>1720</td>
</tr>
<tr>
<td>93.2 Traditional Static Allocation Policy (DYN02)</td>
<td>1720</td>
</tr>
<tr>
<td>93.2.1 Reference</td>
<td>1720</td>
</tr>
<tr>
<td>93.2.2 Description</td>
<td>1720</td>
</tr>
<tr>
<td>93.2.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1721</td>
</tr>
<tr>
<td>93.2.4 Noncompliant Code Example</td>
<td>1721</td>
</tr>
<tr>
<td>93.2.5 Compliant Code Example</td>
<td>1721</td>
</tr>
<tr>
<td>93.2.6 Notes</td>
<td>1721</td>
</tr>
<tr>
<td>93.3 Access Types Without Allocators Policy (DYN03)</td>
<td>1721</td>
</tr>
<tr>
<td>93.3.1 Reference</td>
<td>1722</td>
</tr>
<tr>
<td>93.3.2 Description</td>
<td>1722</td>
</tr>
<tr>
<td>93.3.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1722</td>
</tr>
<tr>
<td>93.3.4 Noncompliant Code Example</td>
<td>1722</td>
</tr>
<tr>
<td>93.3.5 Compliant Code Example</td>
<td>1723</td>
</tr>
<tr>
<td>93.3.6 Notes</td>
<td>1723</td>
</tr>
<tr>
<td>93.4 Minimal Dynamic Allocation Policy (DYN04)</td>
<td>1723</td>
</tr>
<tr>
<td>93.4.1 Reference</td>
<td>1723</td>
</tr>
<tr>
<td>93.4.2 Description</td>
<td>1724</td>
</tr>
<tr>
<td>93.4.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1724</td>
</tr>
<tr>
<td>93.4.4 Noncompliant Code Example</td>
<td>1724</td>
</tr>
<tr>
<td>93.4.5 Compliant Code Example</td>
<td>1724</td>
</tr>
<tr>
<td>93.4.6 Notes</td>
<td>1724</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>93.5 User-Defined Storage Pools Policy (DYNO5)</td>
<td>1724</td>
</tr>
<tr>
<td>93.5.1 Reference</td>
<td>1725</td>
</tr>
<tr>
<td>93.5.2 Description</td>
<td>1725</td>
</tr>
<tr>
<td>93.5.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1725</td>
</tr>
<tr>
<td>93.5.4 Noncompliant Code Example</td>
<td>1726</td>
</tr>
<tr>
<td>93.5.5 Compliant Code Example</td>
<td>1726</td>
</tr>
<tr>
<td>93.5.6 Notes</td>
<td>1726</td>
</tr>
<tr>
<td>93.6 Synchronously Determine Maximum Stack Requirements (DYNO6)</td>
<td>1726</td>
</tr>
<tr>
<td>93.6.1 Reference</td>
<td>1727</td>
</tr>
<tr>
<td>93.6.2 Description</td>
<td>1727</td>
</tr>
<tr>
<td>93.6.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1727</td>
</tr>
<tr>
<td>93.6.4 Noncompliant Code Example</td>
<td>1727</td>
</tr>
<tr>
<td>93.6.5 Compliant Code Example</td>
<td>1727</td>
</tr>
<tr>
<td>93.6.6 Notes</td>
<td>1727</td>
</tr>
<tr>
<td>94 Safe Reclamation (RCL)</td>
<td>1729</td>
</tr>
<tr>
<td>94.1 No Multiple Reclamations (RCL01)</td>
<td>1729</td>
</tr>
<tr>
<td>94.1.1 Reference</td>
<td>1730</td>
</tr>
<tr>
<td>94.1.2 Description</td>
<td>1730</td>
</tr>
<tr>
<td>94.1.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1730</td>
</tr>
<tr>
<td>94.1.4 Noncompliant Code Example</td>
<td>1730</td>
</tr>
<tr>
<td>94.1.5 Compliant Code Example</td>
<td>1730</td>
</tr>
<tr>
<td>94.1.6 Notes</td>
<td>1730</td>
</tr>
<tr>
<td>94.2 Only Reclaim Allocated Storage (RCL02)</td>
<td>1731</td>
</tr>
<tr>
<td>94.2.1 Reference</td>
<td>1731</td>
</tr>
<tr>
<td>94.2.2 Description</td>
<td>1731</td>
</tr>
<tr>
<td>94.2.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1731</td>
</tr>
<tr>
<td>94.2.4 Noncompliant Code Example</td>
<td>1731</td>
</tr>
<tr>
<td>94.2.5 Compliant Code Example</td>
<td>1732</td>
</tr>
<tr>
<td>94.2.6 Notes</td>
<td>1732</td>
</tr>
<tr>
<td>94.3 Only Reclaim to the Same Pool (RCL03)</td>
<td>1732</td>
</tr>
<tr>
<td>94.3.1 Reference</td>
<td>1732</td>
</tr>
<tr>
<td>94.3.2 Description</td>
<td>1732</td>
</tr>
<tr>
<td>94.3.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1733</td>
</tr>
<tr>
<td>94.3.4 Noncompliant Code Example</td>
<td>1733</td>
</tr>
<tr>
<td>94.3.5 Compliant Code Example</td>
<td>1733</td>
</tr>
<tr>
<td>94.3.6 Notes</td>
<td>1733</td>
</tr>
<tr>
<td>95 Concurrency (CON)</td>
<td>1735</td>
</tr>
<tr>
<td>95.1 Use the Ravenscar Profile (CON01)</td>
<td>1736</td>
</tr>
<tr>
<td>95.1.1 Reference</td>
<td>1737</td>
</tr>
<tr>
<td>95.1.2 Description</td>
<td>1737</td>
</tr>
<tr>
<td>95.1.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1737</td>
</tr>
<tr>
<td>95.1.4 Noncompliant Code Example</td>
<td>1738</td>
</tr>
<tr>
<td>95.1.5 Compliant Code Example</td>
<td>1738</td>
</tr>
<tr>
<td>95.1.6 Notes</td>
<td>1738</td>
</tr>
<tr>
<td>95.2 Use the Jorvik Profile (CON02)</td>
<td>1738</td>
</tr>
<tr>
<td>95.2.1 Reference</td>
<td>1739</td>
</tr>
<tr>
<td>95.2.2 Description</td>
<td>1739</td>
</tr>
<tr>
<td>95.2.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1740</td>
</tr>
<tr>
<td>95.2.4 Noncompliant Code Example</td>
<td>1740</td>
</tr>
<tr>
<td>95.2.5 Compliant Code Example</td>
<td>1740</td>
</tr>
<tr>
<td>95.2.6 Notes</td>
<td>1740</td>
</tr>
<tr>
<td>95.3 Avoid Shared Variables for Inter-task Communication (CON03)</td>
<td>1740</td>
</tr>
<tr>
<td>95.3.1 Reference</td>
<td>1741</td>
</tr>
<tr>
<td>95.3.2 Description</td>
<td>1741</td>
</tr>
<tr>
<td>95.3.3 Applicable Vulnerability within ISO TR 24772-2</td>
<td>1741</td>
</tr>
<tr>
<td>95.3.4 Noncompliant Code Example</td>
<td>1741</td>
</tr>
</tbody>
</table>
95.3.5 Compliant Code Example ........................................ 1742
95.3.6 Notes ............................................................. 1742

96 Robust Programming Practice (RPP) ................................... 1743
96.1 No Use of "others" in Case Constructs (RPP01) ................. 1743
  96.1.1 Reference ...................................................... 1744
  96.1.2 Description .................................................... 1744
  96.1.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1744
  96.1.4 Noncompliant Code Example ............................... 1744
  96.1.5 Compliant Code Example ................................... 1744
  96.1.6 Notes .......................................................... 1745
96.2 No Enumeration Ranges in Case Constructs (RPP02) ........... 1745
  96.2.1 Reference ...................................................... 1745
  96.2.2 Description .................................................... 1745
  96.2.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1746
  96.2.4 Noncompliant Code Example ............................... 1746
  96.2.5 Compliant Code Example ................................... 1746
  96.2.6 Notes .......................................................... 1746
96.3 Limited Use of "others" in Aggregates (RPP03) .................. 1746
  96.3.1 Reference ...................................................... 1747
  96.3.2 Description .................................................... 1747
  96.3.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1747
  96.3.4 Noncompliant Code Example ............................... 1747
  96.3.5 Compliant Code Example ................................... 1747
  96.3.6 Notes .......................................................... 1748
96.4 No Unassigned Mode-Out Procedure Parameters (RPP04) .... 1748
  96.4.1 Reference ...................................................... 1748
  96.4.2 Description .................................................... 1748
  96.4.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1749
  96.4.4 Noncompliant Code Example ............................... 1749
  96.4.5 Compliant Code Example ................................... 1749
  96.4.6 Notes .......................................................... 1749
96.5 No Use of "others" in Exception Handlers (RPP05) ............ 1750
  96.5.1 Reference ...................................................... 1750
  96.5.2 Description .................................................... 1750
  96.5.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1750
  96.5.4 Noncompliant Code Example ............................... 1750
  96.5.5 Compliant Code Example ................................... 1751
  96.5.6 Notes .......................................................... 1751
96.6 Avoid Function Side-Effects (RPP06) ................................ 1751
  96.6.1 Reference ...................................................... 1752
  96.6.2 Description .................................................... 1752
  96.6.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1752
  96.6.4 Noncompliant Code Example ............................... 1752
  96.6.5 Compliant Code Example ................................... 1752
  96.6.6 Notes .......................................................... 1752
96.7 Functions Only Have Mode "in" (RPP07) ............................. 1753
  96.7.1 Reference ...................................................... 1753
  96.7.2 Description .................................................... 1753
  96.7.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1753
  96.7.4 Noncompliant Code Example ............................... 1753
  96.7.5 Compliant Code Example ................................... 1754
  96.7.6 Notes .......................................................... 1754
96.8 Limit Parameter Aliasing (RPP08) .................................. 1754
  96.8.1 Reference ...................................................... 1755
  96.8.2 Description .................................................... 1755
  96.8.3 Applicable Vulnerability within ISO TR 24772-2 ......... 1756
  96.8.4 Noncompliant Code Example ............................... 1756
96.8.5 Compliant Code Example ................................................ 1756
96.8.6 Notes ................................................................. 1756
96.9 Use Precondition and Postcondition Contracts (RPP09) .......... 1756
  96.9.1 Reference ........................................................... 1757
  96.9.2 Description ........................................................... 1757
  96.9.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1757
  96.9.4 Noncompliant Code Example ...................................... 1757
  96.9.5 Compliant Code Example .......................................... 1758
  96.9.6 Notes .............................................................. 1758
96.10 Do Not Re-Verify Preconditions in Subprogram Bodies (RPP10) 1758
  96.10.1 Reference .......................................................... 1759
  96.10.2 Description .......................................................... 1759
  96.10.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1759
  96.10.4 Noncompliant Code Example ...................................... 1759
  96.10.5 Compliant Code Example .......................................... 1759
  96.10.6 Notes .............................................................. 1760
96.11 Always Use the Result of Function Calls (RPP11) .................. 1760
  96.11.1 Reference .......................................................... 1760
  96.11.2 Description .......................................................... 1760
  96.11.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1761
  96.11.4 Noncompliant Code Example ...................................... 1761
  96.11.5 Compliant Code Example .......................................... 1761
  96.11.6 Notes .............................................................. 1762
96.12 No Recursion (RPP12) .................................................... 1762
  96.12.1 Reference .......................................................... 1762
  96.12.2 Description .......................................................... 1762
  96.12.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1762
  96.12.4 Noncompliant Code Example ...................................... 1762
  96.12.5 Compliant Code Example .......................................... 1762
  96.12.6 Notes .............................................................. 1762
96.13 No Reuse of Standard Typemarks (RPP13) ........................... 1763
  96.13.1 Reference .......................................................... 1763
  96.13.2 Description .......................................................... 1763
  96.13.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1763
  96.13.4 Noncompliant Code Example ...................................... 1764
  96.13.5 Compliant Code Example .......................................... 1764
  96.13.6 Notes .............................................................. 1764
96.14 Use Symbolic Constants for Literal Values (RPP14) ............... 1764
  96.14.1 Reference .......................................................... 1764
  96.14.2 Description .......................................................... 1765
  96.14.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1765
  96.14.4 Noncompliant Code Example ...................................... 1765
  96.14.5 Compliant Code Example .......................................... 1765
  96.14.6 Notes .............................................................. 1765

97 Exception Usage (EXU) ..................................................... 1767
97.1 Do Not Raise Language-Defined Exceptions (EXU01) ............... 1768
  97.1.1 Reference .......................................................... 1768
  97.1.2 Description .......................................................... 1768
  97.1.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1768
  97.1.4 Noncompliant Code Example ...................................... 1769
  97.1.5 Compliant Code Example .......................................... 1769
  97.1.6 Notes .............................................................. 1769
97.2 No Unhandled Application-Defined Exceptions (EXU02) ............ 1769
  97.2.1 Reference .......................................................... 1770
  97.2.2 Description .......................................................... 1770
  97.2.3 Applicable Vulnerability within ISO TR 24772-2 ............... 1771
  97.2.4 Noncompliant Code Example ...................................... 1771
98.1 No Class-wide Constructs Policy (OOP01) 1776
  98.1.1 Reference ...................................... 1776
  98.1.2 Description ................................... 1776
  98.1.3 Applicable Vulnerability within ISO TR 24772-2 .... 1776
  98.1.4 Noncompliant Code Example ...................... 1777
  98.1.5 Compliant Code Example ........................ 1777
  98.1.6 Notes ........................................... 1777

98.2 Static Dispatching Only Policy (OOP02) 1777
  98.2.1 Reference ...................................... 1777
  98.2.2 Description ................................... 1778
  98.2.3 Applicable Vulnerability within ISO TR 24772-2 .... 1778
  98.2.4 Noncompliant Code Example ...................... 1778
  98.2.5 Compliant Code Example ........................ 1778
  98.2.6 Notes ........................................... 1778

98.3 Limit Inheritance Hierarchy Depth (OOP03) 1778
  98.3.1 Reference ...................................... 1779
  98.3.2 Description ................................... 1779
  98.3.3 Applicable Vulnerability within ISO TR 24772-2 .... 1779
  98.3.4 Noncompliant Code Example ...................... 1779
  98.3.5 Compliant Code Example ........................ 1780
  98.3.6 Notes ........................................... 1780

98.4 Limit Statically-Dispatched Calls to Primitive Operations (OOP04) 1780
  98.4.1 Reference ...................................... 1780
  98.4.2 Description ................................... 1781
  98.4.3 Applicable Vulnerability within ISO TR 24772-2 .... 1781
  98.4.4 Noncompliant Code Example ...................... 1781
  98.4.5 Compliant Code Example ........................ 1782
  98.4.6 Notes ........................................... 1782

98.5 Use Explicit OverridingAnnotations (OOP05) 1782
  98.5.1 Reference ...................................... 1783
  98.5.2 Description ................................... 1783
  98.5.3 Applicable Vulnerability within ISO TR 24772-2 .... 1783
  98.5.4 Noncompliant Code Example ...................... 1784
  98.5.5 Compliant Code Example ........................ 1784
  98.5.6 Notes ........................................... 1784

98.6 Use Class-wide Pre/Post Contracts (OOP06) 1784
  98.6.1 Reference ...................................... 1785
  98.6.2 Description ................................... 1785
  98.6.3 Applicable Vulnerability within ISO TR 24772-2 .... 1785
  98.6.4 Noncompliant Code Example ...................... 1785
### Modular Programming
- 103.1 Months .................................................. 1819
- 103.2 Operations .............................................. 1820

### Strongly typed language
- 104.1 Colors .................................................. 1823
- 104.2 Integers ................................................ 1826
- 104.3 Temperatures .......................................... 1830

### Records
- 105.1 Directions .............................................. 1835
- 105.2 Colors .................................................. 1837
- 105.3 Inventory ............................................... 1841

### Arrays
- 106.1 Constrained Array .................................... 1845
- 106.2 Colors: Lookup-Table ................................. 1847
- 106.3 Unconstrained Array .................................. 1850
- 106.4 Product info ........................................... 1853
- 106.5 String_10 ............................................... 1856
- 106.6 List of Names .......................................... 1858

### More About Types
- 107.1 Aggregate Initialization .............................. 1863
- 107.2 Versioning ............................................. 1865
- 107.3 Simple todo list ....................................... 1867
- 107.4 Price list ............................................... 1869

### Privacy
- 108.1 Directions .............................................. 1875
- 108.2 Limited Strings ...................................... 1877
- 108.3 Bonus exercise ....................................... 1881
  - 108.3.1 Colors ............................................. 1882
  - 108.3.2 List of Names .................................... 1882
  - 108.3.3 Price List ........................................ 1882

### Generics
- 109.1 Display Array ......................................... 1883
- 109.2 Average of Array of Float .......................... 1885
- 109.3 Average of Array of Any Type ...................... 1887
- 109.4 Generic list ......................................... 1890

### Exceptions
- 110.1 Uninitialized Value .................................. 1893
- 110.2 Numerical Exception ................................. 1894
- 110.3 Re-raising Exceptions ............................... 1896

### Tasking
- 111.1 Display Service ....................................... 1899
- 111.2 Event Manager ........................................ 1900
- 111.3 Generic Protected Queue ............................ 1902

### Design by contracts
- 112.1 Price Range ........................................... 1905
- 112.2 Pythagorean Theorem: Predicate .................... 1906
- 112.3 Pythagorean Theorem: Precondition ................ 1908
- 112.4 Pythagorean Theorem: Postcondition ............... 1910
- 112.5 Pythagorean Theorem: Type Invariant .............. 1912
- 112.6 Primary Color ........................................ 1914
Part I

Introduction to Ada
This course will teach you the basics of the Ada programming language and is intended for those who already have a basic understanding of programming techniques. You will learn how to apply those techniques to programming in Ada.

This document was written by Raphaël Amiard and Gustavo A. Hoffmann, with review from Richard Kenner.

**Note:** The code examples in this course use a 50-column limit, which greatly improves the readability of the code on devices with a small screen size. This constraint, however, leads to an unusual coding style. For instance, instead of calling `Put_Line` in a single line, we have this:

```ada
Put_Line (" is in the northeast quadrant");
```

or this:

```ada
Put_Line (" (X => 
  & Integer'Image (P.X)
  & ")");
```

Note that typical Ada code uses a limit of at least 79 columns. Therefore, please don't take the coding style from this course as a reference!

**Note:** Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

`projects/Courses/Intro_To_Ada/Imperative_Language/Greet/cba89a34b87c9dfa71533d982d05e6ab/`

1. http://creativecommons.org/licenses/by-sa/4.0
In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).
INTRODUCTION

1.1 History

In the 1970s the United States Department of Defense (DOD) suffered from an explosion of the number of programming languages, with different projects using different and non-standard dialects or language subsets / supersets. The DOD decided to solve this problem by issuing a request for proposals for a common, modern programming language. The winning proposal was one submitted by Jean Ichbiah from CII Honeywell-Bull.

The first Ada standard was issued in 1983; it was subsequently revised and enhanced in 1995, 2005 and 2012, with each revision bringing useful new features.

This tutorial will focus on Ada 2012 as a whole, rather than teaching different versions of the language.

1.2 Ada today

Today, Ada is heavily used in embedded real-time systems, many of which are safety critical. While Ada is and can be used as a general-purpose language, it will really shine in low-level applications:

• Embedded systems with low memory requirements (no garbage collector allowed).
• Direct interfacing with hardware.
• Soft or hard real-time systems.
• Low-level systems programming.

Specific domains seeing Ada usage include Aerospace & Defense, civil aviation, rail, and many others. These applications require a high degree of safety: a software defect is not just an annoyance, but may have severe consequences. Ada provides safety features that detect defects at an early stage — usually at compilation time or using static analysis tools. Ada can also be used to create applications in a variety of other areas, such as:

• Video game programming
• Real-time audio
• Kernel modules

This is a non-comprehensive list that hopefully sheds light on which kind of programming Ada is good at.

3 https://github.com/AdaDoom3/AdaDoom3
4 http://www.electronicdesign.com/embedded-revolution/assessing-ada-language-audio-applications
5 http://www.nhamkin.com/tag/kernel.html
In terms of modern languages, the closest in terms of targets and level of abstraction are probably [C++](https://en.wikipedia.org/wiki/C%2B%2B) and [Rust](https://www.rust-lang.org/en-US/).

### 1.3 Philosophy

Ada’s philosophy is different from most other languages. Underlying Ada’s design are principles that include the following:

- **Readability is more important than conciseness.** Syntactically this shows through the fact that keywords are preferred to symbols, that no keyword is an abbreviation, etc.
- **Very strong typing.** It is very easy to introduce new types in Ada, with the benefit of preventing data usage errors.
  - It is similar to many functional languages in that regard, except that the programmer has to be much more explicit about typing in Ada, because there is almost no type inference.
- **Explicit is better than implicit.** Although this is a [Python](https://www.python.org) commandment, Ada takes it way further than any language we know of:
  - There is mostly no structural typing, and most types need to be explicitly named by the programmer.
  - As previously said, there is mostly no type inference.
  - Semantics are very well defined, and undefined behavior is limited to an absolute minimum.
  - The programmer can generally give a lot of information about what their program means to the compiler (and other programmers). This allows the compiler to be extremely helpful (read: strict) with the programmer.

During this course, we will explain the individual language features that are building blocks for that philosophy.

### 1.4 SPARK

While this class is solely about the Ada language, it is worth mentioning that another language, extremely close to and interoperable with Ada, exists: the SPARK language.

SPARK is a subset of Ada, designed so that the code written in SPARK is amenable to automatic proof. This provides a level of assurance with regard to the correctness of your code that is much higher than with a regular programming language.

There is a dedicated [course for the SPARK language](page973) but keep in mind that every time we speak about the specification power of Ada during this course, it is power that you can leverage in SPARK to help proving the correctness of program properties ranging from absence of run-time errors to compliance with formally specified functional requirements.
Ada is a multi-paradigm language with support for object orientation and some elements of functional programming, but its core is a simple, coherent procedural/imperative language akin to C or Pascal.

In other languages
One important distinction between Ada and a language like C is that statements and expressions are very clearly distinguished. In Ada, if you try to use an expression where a statement is required then your program will fail to compile. This rule supports a useful stylistic principle: expressions are intended to deliver values, not to have side effects. It can also prevent some programming errors, such as mistakenly using the equality operator = instead of the assignment operation := in an assignment statement.

2.1 Hello world

Here’s a very simple imperative Ada program:

Listing 1: greet.adb

```ada
with Ada.Text_IO;

procedure Greet is
begin
  -- Print "Hello, World!" to the screen
  Ada.Text_IO.Put_Line ("Hello, World!");
end Greet;
```

Code block metadata

- **Project:** Courses.Intro_To_Ada.Imperative_Language.Greet
- **MD5:** cba89a34b87c9dfa71533d982d05e6ab

Runtime output

Hello, World!

which we’ll assume is in the source file greet.adb.

There are several noteworthy things in the above program:

- A subprogram in Ada can be either a procedure or a function. A procedure, as illustrated above, does not return a value when called.

- `with` is used to reference external modules that are needed in the procedure. This is similar to `import` in various languages or roughly similar to `#include` in C and C++.
Learning Ada

We'll see later how they work in detail. Here, we are requesting a standard library module, the Ada.Text_IO package, which contains a procedure to print text on the screen: Put_Line.

- Greet is a procedure, and the main entry point for our first program. Unlike in C or C++, it can be named anything you prefer. The builder will determine the entry point. In our simple example, gprbuild, GNAT's builder, will use the file you passed as parameter.

- Put_Line is a procedure, just like Greet, except it is declared in the Ada.Text_IO module. It is the Ada equivalent of C's printf.

- Comments start with -- and go to the end of the line. There is no multi-line comment syntax, that is, it is not possible to start a comment in one line and continue it in the next line. The only way to create multiple lines of comments in Ada is by using -- on each line. For example:

```
-- We start a comment in this line...
-- and we continue on the second line...
```

In other languages

Procedures are similar to functions in C or C++ that return void. We'll see later how to declare functions in Ada.

Here is a minor variant of the "Hello, World" example:

Listing 2: greet.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
begin
  -- Print "Hello, World!" to the screen
  Put_Line ("Hello, World!");
end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_2
MD5: a58a1193207df44aa6edaa4fe1c14280

Runtime output

Hello, World!

This version utilizes an Ada feature known as a use clause, which has the form use package-name. As illustrated by the call on Put_Line, the effect is that entities from the named package can be referenced directly, without the package-name. prefix.
2.2 Imperative language - If/Then/Else

This section describes Ada's if statement and introduces several other fundamental language facilities including integer I/O, data declarations, and subprogram parameter modes.

Ada's if statement is pretty unsurprising in form and function:

Listing 3: check_positive.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Check_Positive is
  N : Integer;
begin
  -- Put a String
  Put ("Enter an integer value: ");
  -- Read in an integer value
  Get (N);

  if N > 0 then
    -- Put an Integer
    Put (N);
    Put_Line (" is a positive number");
  end if;
end Check_Positive;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Check_Positive
MD5: 2e8b4b2f3f258fd9e02c2d65b46a1f01

The if statement minimally consists of the reserved word if, a condition (which must be a Boolean value), the reserved word then and a non-empty sequence of statements (the then part) which is executed if the condition evaluates to True, and a terminating end if.

This example declares an integer variable N, prompts the user for an integer, checks if the value is positive and, if so, displays the integer's value followed by the string " is a positive number". If the value is not positive, the procedure does not display any output.

The type Integer is a predefined signed type, and its range depends on the computer architecture. On typical current processors Integer is 32-bit signed.

The example illustrates some of the basic functionality for integer input-output. The relevant subprograms are in the predefined package Ada.Integer_Text_IO and include the Get procedure (which reads in a number from the keyboard) and the Put procedure (which displays an integer value).

Here's a slight variation on the example, which illustrates an if statement with an else part:

Listing 4: check_positive.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Check_Positive is
  N : Integer;
begin
  -- Put a String
  Put ("Enter an integer value: ");
  -- Read in an integer value
  Get (N);

  if N > 0 then
    -- Put an Integer
    Put (N);
    Put_Line (" is a positive number");
  else
    put ("is a negative number");
  end if;
end Check_Positive;
```

(continues on next page)
-- Reads in an integer value
Get (N);

-- Put an Integer
Put (N);

if N > 0 then
  Put_Line (" is a positive number");
else
  Put_Line (" is not a positive number");
end if;
end Check_Positive;

In this example, if the input value is not positive then the program displays the value followed by the String " is not a positive number".

Our final variation illustrates an if statement with elsif sections:

Listing 5: check_direction.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Check_Direction is
  N : Integer;
begin
  Put ("Enter an integer value: ");
  Get (N);
  Put (N);

  if N = 0 or N = 360 then
    Put_Line (" is due north");
  elsif N in 1 .. 89 then
    Put_Line (" is in the northeast quadrant");
  elsif N = 90 then
    Put_Line (" is due east");
  elsif N in 91 .. 179 then
    Put_Line (" is in the southeast quadrant");
  elsif N = 180 then
    Put_Line (" is due south");
  elsif N in 181 .. 269 then
    Put_Line (" is in the southwest quadrant");
  elsif N = 270 then
    Put_Line (" is in the northwest quadrant");
  elsif N in 271 .. 359 then
    Put_Line (" is in the northwest quadrant");
  else
    Put_Line (" is not in the range 0..360");
  end if;
end Check_Direction;
This example expects the user to input an integer between 0 and 360 inclusive, and displays which quadrant or axis the value corresponds to. The in operator in Ada tests whether a scalar value is within a specified range and returns a Boolean result. The effect of the program should be self-explanatory; later we'll see an alternative and more efficient style to accomplish the same effect, through a case statement.

Ada's elsif keyword differs from C or C++, where nested else if blocks would be used instead. And another difference is the presence of the end if in Ada, which avoids the problem known as the "dangling else".

### 2.3 Imperative language - Loops

Ada has three ways of specifying loops. They differ from the C / Java / Javascript for-loop, however, with simpler syntax and semantics in line with Ada's philosophy.

#### 2.3.1 For loops

The first kind of loop is the for loop, which allows iteration through a discrete range.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet_5a is
begin
  for I in 1 .. 5 loop
    -- Put_Line is a procedure call
    Put_Line ("Hello, World!"
                & Integer'Image (I));
  end loop;
end Greet_5a;
```

A few things to note:

- `1 .. 5` is a discrete range, from 1 to 5 inclusive.
- The loop parameter I (the name is arbitrary) in the body of the loop has a value within this range.
- I is local to the loop, so you cannot refer to I outside the loop.
- Although the value of I is incremented at each iteration, from the program's perspective it is constant. An attempt to modify its value is illegal; the compiler would reject the program.
Learning Ada

- **Integer'Image** is a function that takes an Integer and converts it to a **String**. It is an example of a language construct known as an **attribute**, indicated by the ' syntax, which will be covered in more detail later.

- The & symbol is the concatenation operator for String values

- The **end loop** marks the end of the loop

The "step" of the loop is limited to 1 (forward direction) and -1 (backward). To iterate backwards over a range, use the **reverse** keyword:

Listing 7: greet_5a_reverse.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet_5a_Reverse is
begin
  for I in reverse 1 .. 5 loop
    Put_Line ("Hello, World!"
      & Integer'Image (I));
  end loop;
end Greet_5a_Reverse;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_5a_Reverse
MD5: a0d5dcfc471fb1a107477c934fa527c2

Runtime output

Hello, World! 5
Hello, World! 4
Hello, World! 3
Hello, World! 2
Hello, World! 1

The bounds of a **for** loop may be computed at run-time; they are evaluated once, before the loop body is executed. If the value of the upper bound is less than the value of the lower bound, then the loop is not executed at all. This is the case also for **reverse** loops. Thus no output is produced in the following example:

Listing 8: greet_no_op.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet_No_Op is
begin
  for I in reverse 5 .. 1 loop
    Put_Line ("Hello, World!"
      & Integer'Image (I));
  end loop;
end Greet_No_Op;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_No_Op
MD5: 5070693fb0324d3e4e43a8c8c4f046e1

Build output

```
greet_no_op.adb:5:23: warning: loop range is null, loop will not execute [enabled, by default]
```

The **for** loop is more general than what we illustrated here; more on that later.
2.3.2 Bare loops

The simplest loop in Ada is the bare loop, which forms the foundation of the other kinds of Ada loops.

Listing 9: greet_5b.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet_5b is
  -- Variable declaration:
  I : Integer := 1;
  -- ^ Type
  -- ^ Initial value
begin
  loop
    Put_Line ("Hello, World!" & Integer'Image (I));
    -- Exit statement:
    exit when I = 5;
    -- ^ Boolean condition
  end loop;
end Greet_5b;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_5b
MD5: 5b218a64a07f64bd97774b5748b83c44a

Runtime output

Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5

This example has the same effect as Greet_5a shown earlier.

It illustrates several concepts:

- We have declared a variable named I between the `is` and the `begin`. This constitutes a declarative region. Ada clearly separates the declarative region from the statement part of a subprogram. A declaration can appear in a declarative region but is not allowed as a statement.

- The bare loop statement is introduced by the keyword `loop` on its own and, like every kind of loop statement, is terminated by the combination of keywords `end loop`. On its own, it is an infinite loop. You can break out of it with an `exit` statement.

- The syntax for assignment is `:=`, and the one for equality is `=`. There is no way to confuse them, because as previously noted, in Ada, statements and expressions are distinct, and expressions are not valid statements.
2.3.3 While loops

The last kind of loop in Ada is the **while** loop.

Listing 10: greet_5c.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet_5c is
  I : Integer := 1;
begin
  -- Condition must be a Boolean value
  -- (no Integers).
  -- Operator "=\" returns a Boolean
  while I =< 5 loop
    Put_Line ("Hello, World!"
              & Integer'Image (I));
    I := I + 1;
  end loop;
end Greet_5c;
```

The condition is evaluated before each iteration. If the result is false, then the loop is terminated.

This program has the same effect as the previous examples.

---

In other languages

Note that Ada has different semantics than C-based languages with respect to the condition in a while loop. In Ada the condition has to be a Boolean value or the compiler will reject the program; the condition is not an integer that is treated as either **True** or **False** depending on whether it is non-zero or zero.

2.4 Imperative language - Case statement

Ada's **case** statement is similar to the C and C++ **switch** statement, but with some important differences.

Here's an example, a variation of a program that was shown earlier with an **if** statement:

Listing 11: check_direction.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
```
procedure Check_Direction is
    N : Integer;
begin
    loop
        Put ("Enter an integer value: ");
        Get (N);
        Put (N);
        case N is
            when 0 | 360 =>
                Put_Line (" is due north");
            when 1 .. 89 =>
                Put_Line (" is in the northeast quadrant");
            when 90 =>
                Put_Line (" is due east");
            when 91 .. 179 =>
                Put_Line (" is in the southeast quadrant");
            when 180 =>
                Put_Line (" is due south");
            when 181 .. 269 =>
                Put_Line (" is in the southwest quadrant");
            when 270 =>
                Put_Line (" is due west");
            when 271 .. 359 =>
                Put_Line (" is in the northwest quadrant");
            when others =>
                Put_Line (" Au revoir");
                exit;
        end case;
    end loop;
end Check_Direction;

This program repeatedly prompts for an integer value and then, if the value is in the range \(0 \ldots 360\), displays the associated quadrant or axis. If the value is an Integer outside this range, the loop (and the program) terminate after outputting a farewell message.

The effect of the case statement is similar to the if statement in an earlier example, but the case statement can be more efficient because it does not involve multiple range tests.

Notable points about Ada's case statement:

- The case expression (here the variable \(N\)) must be of a discrete type, i.e. either an integer type or an enumeration type. Discrete types will be covered in more detail later discrete types (page 47).
- Every possible value for the case expression needs to be covered by a unique branch of the case statement. This will be checked at compile time.
Learning Ada

• A branch can specify a single value, such as 0; a range of values, such as 1 .. 89; or any combination of the two (separated by a ).

• As a special case, an optional final branch can specify others, which covers all values not included in the earlier branches.

• Execution consists of the evaluation of the case expression and then a transfer of control to the statement sequence in the unique branch that covers that value.

• When execution of the statements in the selected branch has completed, control resumes after the end case. Unlike C, execution does not fall through to the next branch. So Ada doesn't need (and doesn't have) a break statement.

2.5 Imperative language - Declarative regions

As mentioned earlier, Ada draws a clear syntactic separation between declarations, which introduce names for entities that will be used in the program, and statements, which perform the processing. The areas in the program where declarations may appear are known as declarative regions.

In any subprogram, the section between the is and the begin is a declarative region. You can have variables, constants, types, inner subprograms, and other entities there.

We've briefly mentioned variable declarations in previous subsection. Let's look at a simple example, where we declare an integer variable X in the declarative region and perform an initialization and an addition on it:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  X : Integer;
begin
  X := 0;
  Put_Line ("The initial value of X is ",
              & Integer'Image (X));
  Put_Line ("Performing operation on X...");
  X := X + 1;
  Put_Line ("The value of X now is ",
              & Integer'Image (X));
end Main;
```

Let's look at an example of a nested procedure:
Listing 13: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  procedure Nested is
    begin
      Put_Line ("Hello World");
    end Nested;
    begin
      Nested;
      -- Call to Nested
    end Main;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Imperative_Language.Nested_Procedure

MD5: 2e7fb267e31232196065febd5e35e6ef

**Runtime output**

Hello World

A declaration cannot appear as a statement. If you need to declare a local variable amidst the statements, you can introduce a new declarative region with a block statement:

Listing 14: greet.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
  begin
    loop
      declare
        Name : String := Get_Line;
        -- ^ Call to the
        -- Get_Line function
      begin
        exit when Name = "";
        Put_Line ("Hi " & Name & "!");
      end;
      -- Name is undefined here
    end loop;
    Put_Line ("Bye!");
  end Greet;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_6

MD5: a9c0c14a1b3e2ebe07cd88f442787e3a

**Attention:** The Get_Line function allows you to receive input from the user, and get the result as a string. It is more or less equivalent to the scanf C function.

It returns a `String`, which, as we will see later, is an *Unconstrained array type* (page 78). For now we simply note that, if you wish to declare a `String` variable and do not know
its size in advance, then you need to initialize the variable during its declaration.

## 2.6 Imperative language - conditional expressions

Ada 2012 introduced an expression analog for conditional statements (`if` and `case`).

### 2.6.1 If expressions

Here's an alternative version of an example we saw earlier; the `if` statement has been replaced by an `if` expression:

```ada
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Check_Positive is
  N : Integer;
begin
  Put ("Enter an integer value: ");
  Get (N);
  Put (N);

  declare
    S : constant String :=
      (if N > 0 then " is a positive number"
        else " is not a positive number");
  begin
    Put_Line (S);
  end;
end Check_Positive;
```

The `if` expression evaluates to one of the two Strings depending on `N`, and assigns that value to the local variable `S`.

Ada's `if` expressions are similar to `if` statements. However, there are a few differences that stem from the fact that it is an expression:

- All branches' expressions must be of the same type
- It ***must*** be surrounded by parentheses if the surrounding expression does not already contain them
- An `else` branch is mandatory unless the expression following `then` has a Boolean value. In that case an `else` branch is optional and, if not present, defaults to `else True`.

Here's another example:
Listing 16: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
  for I in 1 .. 10 loop
    Put_Line (if I mod 2 = 0
      then "Even"
      else "Odd");
  end loop;
end Main;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Imperative.Language.Even_Odd
MD5: c89c3233ab8822c028f7a7bba8fd3f1c

**Runtime output**

Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even

This program produces 10 lines of output, alternating between "Odd" and "Even".

### 2.6.2 Case expressions

Analogous to *if* expressions, Ada also has *case* expressions. They work just as you would expect.

Listing 17: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
  for I in 1 .. 10 loop
    Put_Line (case I is
      when 1 | 3 | 5 | 7 | 9 => "Odd",
        when 2 | 4 | 6 | 8 | 10 => "Even");
  end loop;
end Main;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Imperative.Language.Case_Expression
MD5: 6ce40efc987c2665960b1f08d30d780d

**Runtime output**

Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even

**2.6. Imperative language - conditional expressions**
This program has the same effect as the preceding example.

The syntax differs from case statements, with branches separated by commas.
3.1 Subprograms

So far, we have used procedures, mostly to have a main body of code to execute. Procedures are one kind of subprogram.

There are two kinds of subprograms in Ada, functions and procedures. The distinction between the two is that a function returns a value, and a procedure does not.

This example shows the declaration and definition of a function:

Listing 1: increment.ads

```
function Increment (I : Integer) return Integer;
```

Listing 2: increment.adb

```
-- We declare (but don't define) a function with
-- one parameter, returning an integer value
function Increment (I : Integer) return Integer is
    -- We define the Increment function
    begin
        return I + 1;
    end Increment;
```

Subprograms in Ada can, of course, have parameters. One syntactically important note is that a subprogram which has no parameters does not have a parameter section at all, for example:

```
procedure Proc;
function Func return Integer;
```

Here's another variation on the previous example:

Listing 3: increment_by.ads

```
function Increment_By
    (I   : Integer := 0;
     Incr : Integer := 1) return Integer;
    ^ Default value for parameters
```

Code block metadata

Project: Courses.Intro To Ada.Subprograms.Increment
MD5: 582fe283730a130cec071c455a0ce3d4
In this example, we see that parameters can have default values. When calling the subprogram, you can then omit parameters if they have a default value. Unlike C/C++, a call to a subprogram without parameters does not include parentheses.

This is the implementation of the function above:

Listing 4: increment_by.adb

```
function Increment_By
  (I  : Integer  := 0;
   Incr  : Integer  := 1)
return Integer
is
begin
  return I + Incr;
end Increment_By;
```

In the GNAT toolchain

The Ada standard doesn't mandate in which file the specification or the implementation of a subprogram must be stored. In other words, the standard doesn't require a specific file structure or specific file name extensions. For example, we could save both the specification and the implementation of the `Increment` function above in a file called `increment.txt`. (We could even store the entire source code of a system in a single file.) From the standard's perspective, this would be completely acceptable.

The GNAT toolchain, however, requires the following file naming scheme:

- files with the `.ads` extension contain the specification, while
- files with the `.adb` extension contain the implementation.

Therefore, in the GNAT toolchain, the specification of the `Increment` function must be stored in the `increment.ads` file, while its implementation must be stored in the `increment.adb` file. This rule always applies to packages, which we discuss later (page 35). (Note, however, that it's possible to circumvent this rule.) For more details, you may refer to the *Introduction to GNAT Toolchain* (page 1681) course or the *GPRbuild User’s Guide*.

### 3.1.1 Subprogram calls

We can then call our subprogram this way:

Listing 5: show_increment.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Increment_By;

procedure Show_Increment is
  A, B, C : Integer;
begin
  C := Increment_By;
end Show_Increment;
```

---

9 https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html
-- ^ Parameterless call,
-- value of I is 0
-- and Incr is 1

Put_Line ("Using defaults for Increment_By is "
& Integer'Image (C));

A := 10;
B := 3;
C := Increment_By (A, B);

-- ^ Regular parameter passing

Put_Line ("Increment of 
& Integer'Image (A)
& " with 
& Integer'Image (B)
& " is 
& Integer'Image (C));

A := 20;
B := 5;
C := Increment_By (I => A,
                      Incr => B);

-- ^ Named parameter passing

Put_Line ("Increment of 
& Integer'Image (A)
& " with 
& Integer'Image (B)
& " is 
& Integer'Image (C));

end Show_Increment;

Ada allows you to name the parameters when you pass them, whether they have a default or not. There are some rules:

- Positional parameters come first.
- A positional parameter cannot follow a named parameter.

As a convention, people usually name parameters at the call site if the function's corresponding parameters has a default value. However, it is also perfectly acceptable to name every parameter if it makes the code clearer.
3.1.2 Nested subprograms

As briefly mentioned earlier, Ada allows you to declare one subprogram inside another. This is useful for two reasons:

- It lets you organize your programs in a cleaner fashion. If you need a subprogram only as a "helper" for another subprogram, then the principle of localization indicates that the helper subprogram should be declared nested.
- It allows you to share state easily in a controlled fashion, because the nested subprograms have access to the parameters, as well as any local variables, declared in the outer scope.

For the previous example, we can move the duplicated code (call to Put_Line) to a separate procedure. This is a shortened version with the nested Display_Result procedure.

Listing 6: show_increment.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Increment_By;

procedure Show_Increment is
   A, B, C : Integer;

   procedure Display_Result is
      begin
         Put_Line ("Increment of ",
                    & Integer'Image (A),
                    & " with ",
                    & Integer'Image (B),
                    & " is ",
                    & Integer'Image (C));
      end Display_Result;

   begin
      A := 10;
      B := 3;
      C := Increment_By (A, B);
      Display_Result;
      A := 20;
      B := 5;
      C := Increment_By (A, B);
      Display_Result;
   end Show_Increment;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Increment_By
MD5: 23ec8ae3080c042123a9e82ee6b3d9e3

Runtime output

Increment of 10 with 3 is 13
Increment of 20 with 5 is 25
3.1.3 Function calls

An important feature of function calls in Ada is that the return value at a call cannot be ignored; that is, a function call cannot be used as a statement. If you want to call a function and do not need its result, you will still need to explicitly store it in a local variable.

Listing 7: quadruple.adb

```ada
function Quadruple (I : Integer) return Integer is

  function Double (I : Integer) return Integer is
  begin
    return I * 2;
  end Double;

  Res : Integer := Double (Double (I));
  -- ^ Calling the Double
  -- function

begin
  Double (I);
  -- ERROR: cannot use call to function
  -- "Double" as a statement

  return Res;
end Quadruple;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Quadruple
MD5: 44326f12a9d797ea13ffe52ea48fc36f

Build output

quadruple.adb:14:04: error: cannot use call to function "Double" as a statement
quadruple.adb:14:04: error: return value of a function call cannot be ignored
gprbuild: *** compilation phase failed

In the GNAT toolchain

In GNAT, with all warnings activated, it becomes even harder to ignore the result of a function, because unused variables will be flagged. For example, this code would not be valid:

```ada
function Read_Int
  (Stream : Network_Stream; Result : out Integer) return Boolean;

procedure Main is
  Stream : Network_Stream := Get_Stream;
  My_Int : Integer;

  -- Warning: in the line below, B is
  -- never read.
  B : Boolean := Read_Int (Stream, My_Int);
begin
  null;
end Main;
```

You then have two solutions to silence this warning:

3.1. Subprograms
Learning Ada

- Either annotate the variable with `pragma Unreferenced`, e.g.:

  ```ada
  B : Boolean := Read_Int (Stream, My_Int);
  pragma Unreferenced (B);
  ```

- Or give the variable a name that contains any of the strings `discard dummy ignore junk unused` (case insensitive)

3.2 Parameter modes

So far we have seen that Ada is a safety-focused language. There are many ways this is realized, but two important points are:

- Ada makes the user specify as much as possible about the behavior expected for the program, so that the compiler can warn or reject if there is an inconsistency.
- Ada provides a variety of techniques for achieving the generality and flexibility of pointers and dynamic memory management, but without the latter's drawbacks (such as memory leakage and dangling references).

Parameter modes are a feature that helps achieve the two design goals above. A subprogram parameter can be specified with a mode, which is one of the following:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in</code></td>
<td>Parameter can only be read, not written</td>
</tr>
<tr>
<td><code>out</code></td>
<td>Parameter can be written, then read</td>
</tr>
<tr>
<td><code>in out</code></td>
<td>Parameter can be both read and written</td>
</tr>
</tbody>
</table>

The default mode for parameters is `in`; so far, most of the examples have been using `in` parameters.

Historically

Functions and procedures were originally more different in philosophy. Before Ada 2012, functions could only take `in` parameters.

3.3 Subprogram calls

3.3.1 In parameters

The first mode for parameters is the one we have been implicitly using so far. Parameters passed using this mode cannot be modified, so that the following program will cause an error:

```
procedure Swap (A, B : Integer) is
  Tmp : Integer;
begin
  Tmp := A;
  -- Error: assignment to "in" mode
  -- parameter not allowed
```

(Listing 8: swap.adb)

(continues on next page)
A := B;

-- Error: assignment to "in" mode
-- parameter not allowed
B := Tmp;
end Swap;

The fact that in is the default mode is very important. It means that a parameter will not be modified unless you explicitly specify a mode in which modification is allowed.

### 3.3.2 In out parameters

To correct our code above, we can use an in out parameter.

Listing 9: in_out_params.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure In_Out_Params is
  procedure Swap (A, B : in out Integer) is
    Tmp : Integer;
  begin
    Tmp := A;
    A := B;
    B := Tmp;
  end Swap;

  A : Integer := 12;
  B : Integer := 44;
begin
  Swap (A, B);
  -- Prints 44
  Put_Line (Integer'Image (A));
end In_Out_Params;
```

An in out parameter will allow read and write access to the object passed as parameter, so in the example above, we can see that A is modified after the call to Swap.
Attention: While in out parameters look a bit like references in C++, or regular parameters in Java that are passed by-reference, the Ada language standard does not mandate "by reference" passing for in out parameters except for certain categories of types as will be explained later.

In general, it is better to think of modes as higher level than by-value versus by-reference semantics. For the compiler, it means that an array passed as an in parameter might be passed by reference, because it is more efficient (which does not change anything for the user since the parameter is not assignable). However, a parameter of a discrete type will always be passed by copy, regardless of its mode (which is more efficient on most architectures).

3.3.3 Out parameters

The out mode applies when the subprogram needs to write to a parameter that might be uninitialized at the point of call. Reading the value of an out parameter is permitted, but it should only be done after the subprogram has assigned a value to the parameter. Out parameters behave a bit like return values for functions. When the subprogram returns, the actual parameter (a variable) will have the value of the out parameter at the point of return.

In other languages

Ada doesn't have a tuple construct and does not allow returning multiple values from a subprogram (except by declaring a full-fledged record type). Hence, a way to return multiple values from a subprogram is to use out parameters.

For example, a procedure reading integers from the network could have one of the following specifications:

```ada
procedure Read_Int
  (Stream :   Network_Stream;
   Success : out Boolean;
   Result : out Integer);

function Read_Int
  (Stream :   Network_Stream;
   Result : out Integer) return Boolean;
```

While reading an out variable before writing to it should, ideally, trigger an error, imposing that as a rule would cause either inefficient run-time checks or complex compile-time rules. So from the user's perspective an out parameter acts like an uninitialized variable when the subprogram is invoked.

In the GNAT toolchain

GNAT will detect simple cases of incorrect use of out parameters. For example, the compiler will emit a warning for the following program:

```
Listing 10: outp.adb

procedure Outp is
  procedure Foo (A : out Integer) is
    B : Integer := A;
    -- ^ Warning on reference
    -- to uninitialized A
```

(continues on next page)
### 3.3.4 Forward declaration of subprograms

As we saw earlier, a subprogram can be declared without being fully defined. This is possible in general, and can be useful if you need subprograms to be mutually recursive, as in the example below:

Listing 11: mutually_recursive_subprograms.adb

```ada
procedure Mutually_Recursive_Subprograms is
  procedure Compute_A (V : Natural); -- Forward declaration of Compute_A

  procedure Compute_B (V : Natural) is
    begin
      if V > 5 then
        Compute_A (V - 1);
        -- Call to Compute_A
      end if;
    end Compute_B;

  procedure Compute_A (V : Natural) is
    begin
      if V > 2 then
        Compute_B (V - 1);
        -- Call to Compute_B
      end if;
    end Compute_A;

begin
  Compute_A (15);
end Mutually_Recursive_Subprograms;
```

---

**Code block metadata**

Project: Courses.Intro_To_Ada.Subprograms.Mutually_Recursive_Subprograms
MD5: 5ee030cdecc64ea8916cbb763e8526
3.4 Renaming

Subprograms can be renamed by using the `renames` keyword and declaring a new name for a subprogram:

```ada
procedure New_Proc renames Original_Proc;
```

This can be useful, for example, to improve the readability of your application when you're using code from external sources that cannot be changed in your system. Let's look at an example:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
(A_Message : String);
```

Listing 12: a_procedure_with_very_long_name_that_cannot_be_changed.ads

```
procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
(A_Message : String);
```

Listing 13: a_procedure_with_very_long_name_that_cannot_be_changed.adb

As the wording in the name of procedure above implies, we cannot change its name. We can, however, rename it to something like `Show` in our test application and use this shorter name. Note that we also have to declare all parameters of the original subprogram — we may rename them, too, in the declaration. For example:

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming is
  procedure Show (S : String) renames A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
begin
  Show ("Hello World!");
end Show_Renaming;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Proc_Renaming
MD5: 6d4952e9dee8ef69a9e3c3e185c635f1

As the wording in the name of procedure above implies, we cannot change its name. We can, however, rename it to something like `Show` in our test application and use this shorter name. Note that we also have to declare all parameters of the original subprogram — we may rename them, too, in the declaration. For example:

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming is
  procedure Show (S : String) renames A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
begin
  Show ("Hello World!");
end Show_Renaming;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Proc_Renaming
MD5: 5b3b550f8a1cbeb7d9cfd3673f6d42b3

Runtime output

Hello World!

Note that the original name `A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed` is still visible after the declaration of the `Show` procedure.
We may also rename subprograms from the standard library. For example, we may rename `Integer'Image` to `Img`:

Listing 15: show_image_renaming.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Image_Renaming is
  function Img (I : Integer) return String
    renames Integer'Image;

begin
  Put_Line (Img (2));
  Put_Line (Img (3));
end Show_Image_Renaming;
```

Renaming also allows us to introduce default expressions that were not available in the original declaration. For example, we may specify "Hello World!" as the default for the `String` parameter of the `Show` procedure:

```ada
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming_Defaults is
  procedure Show (S : String := "Hello World!")
    renames A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

begin
  Show;
end Show_Renaming_Defaults;
```
So far, our examples have been simple standalone subprograms. Ada is helpful in that regard, since it allows arbitrary declarations in a declarative part. We were thus able to declare our types and variables in the bodies of main procedures.

However, it is easy to see that this is not going to scale up for real-world applications. We need a better way to structure our programs into modular and distinct units.

Ada encourages the separation of programs into multiple packages and sub-packages, providing many tools to a programmer on a quest for a perfectly organized code-base.

### 4.1 Packages

Here is an example of a package declaration in Ada:

```ada
package Week is

    Mon : constant String := "Monday";
    Tue : constant String := "Tuesday";
    Wed : constant String := "Wednesday";
    Thu : constant String := "Thursday";
    Fri : constant String := "Friday";
    Sat : constant String := "Saturday";
    Sun : constant String := "Sunday";

end Week;
```

And here is how you use it:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Week;

-- References the Week package, and
-- adds a dependency from Main to Week

procedure Main is
begin
    Put_Line ("First day of the week is ")
```

(continues on next page)
Packages let you make your code modular, separating your programs into semantically significant units. Additionally the separation of a package's specification from its body (which we will see below) can reduce compilation time.

While the `with` clause indicates a dependency, you can see in the example above that you still need to prefix the referencing of entities from the `Week` package by the name of the package. (If we had included a `use` `Week` clause, then such a prefix would not have been necessary.)

Accessing entities from a package uses the dot notation, `A.B`, which is the same notation as the one used to access record fields.

A `with` clause can only appear in the prelude of a compilation unit (i.e., before the reserved word, such as `procedure`, that marks the beginning of the unit). It is not allowed anywhere else. This rule is only needed for methodological reasons: the person reading your code should be able to see immediately which units the code depends on.

In other languages

Packages look similar to, but are semantically very different from, header files in C/C++.

- The first and most important distinction is that packages are a language-level mechanism. This is in contrast to a `#include`'d header file, which is a functionality of the C preprocessor.
- An immediate consequence is that the `with` construct is a semantic inclusion mechanism, not a text inclusion mechanism. Hence, when you `with` a package, you are saying to the compiler "I'm depending on this semantic unit", and not "include this bunch of text in place here".
- The effect of a package thus does not vary depending on where it has been `with`ed from. Contrast this with C/C++, where the meaning of the included text depends on the context in which the `#include` appears.

This allows compilation/recompilation to be more efficient. It also allows tools like IDEs to have correct information about the semantics of a program. In turn, this allows better tooling in general, and code that is more analyzable, even by humans.

An important benefit of Ada `with` clauses when compared to `#include` is that it is stateless. The order of `with` and `use` clauses does not matter, and can be changed without side effects.

In the GNAT toolchain

The Ada language standard does not mandate any particular relationship between source files and packages; for example, in theory you can put all your code in one file, or use your own file naming conventions. In practice, however, an implementation will have specific rules. With GNAT, each top-level compilation unit needs to go into a separate file. In the
example above, the Week package will be in an .ads file (for Ada specification), and the Main procedure will be in an .adb file (for Ada body).

### 4.2 Using a package

As we have seen above, the **with** clause indicates a dependency on another package. However, every reference to an entity coming from the Week package had to be prefixed by the full name of the package. It is possible to make every entity of a package visible directly in the current scope, using the **use** clause.

In fact, we have been using the **use** clause since almost the beginning of this tutorial.

Listing 3: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
-- ^ Make every entity of the Ada.Text_IO package directly visible.
with Week;

procedure Main is
  use Week;
  -- Make every entity of the Week package directly visible.
begin
  Put_Line ("First day of the week is " & Mon);
end Main;
```

**Code block metadata**

- **Project**: Courses.Intro_To_Ada.Modular_Programming.Week
- **MD5**: ea54077d4ae165b28ae8facfe8ba2db7

**Runtime output**

First day of the week is Monday

As you can see in the example above:

- **Put_Line** is a subprogram that comes from the Ada.Text_IO package. We can reference it directly because we have used the package at the top of the Main unit.
- Unlike **with** clauses, a **use** clause can be placed either in the prelude, or in any declarative region. In the latter case the **use** clause will have an effect in its containing lexical scope.

### 4.3 Package body

In the simple example above, the Week package only has declarations and no body. That's not a mistake: in a package specification, which is what is illustrated above, you cannot declare bodies. Those have to be in the package body.
Here we can see that the body of the Increment_By function has to be declared in the body. Coincidentally, introducing a body allows us to put the Last_Increment variable in the body, and make them inaccessible to the user of the Operations package, providing a first form of encapsulation.

This works because entities declared in the body are only visible in the body.

This example shows how Last_Increment is used indirectly:

```
with Ada.Text_IO; use Ada.Text_IO;
with Operations;

procedure Main is
  use Operations;
  I : Integer := 0;
  R : Integer;
  procedure Display_Update_Values is
  (continues on next page)
```
Incr : constant Integer := Get_Increment_Value;

begin
  Put_Line (Integer’Image (I)
            & " incremented by "
            & Integer’Image (Incr)
            & " is "
            & Integer’Image (R));

  I := R;
end Display_Update_Values;

begin
  R := Increment_By (I);
  Display_Update_Values;
  R := Increment_By (I);
  Display_Update_Values;
  R := Increment_By (I, 5);
  Display_Update_Values;
  R := Increment_By (I);
  Display_Update_Values;
  R := Increment_By (I, 10);
  Display_Update_Values;
  R := Increment_By (I);
  Display_Update_Values;
end Main;

4.4 Child packages

Packages can be used to create hierarchies. We achieve this by using child packages, which extend the functionality of their parent package. One example of a child package that we've been using so far is the Ada.Text_IO package. Here, the parent package is called Ada, while the child package is called Text_IO. In the previous examples, we've been using the Put_Line procedure from the Text_IO child package.

Important

Ada also supports nested packages. However, since they can be more complicated to use, the recommendation is to use child packages instead. Nested packages will be covered in the advanced course.

Let's begin our discussion on child packages by taking our previous Week package:
package Week is

  Mon : constant String := "Monday";
  Tue : constant String := "Tuesday";
  Wed : constant String := "Wednesday";
  Thu : constant String := "Thursday";
  Fri : constant String := "Friday";
  Sat : constant String := "Saturday";
  Sun : constant String := "Sunday";

end Week;

Code block metadata

MD5: 0fa033dc8fe2b9741483de273354e7ee

If we want to create a child package for Week, we may write:

package Week.Child is

  function Get_First_Of_Week return String;

end Week.Child;

Code block metadata

MD5: a7db38e772cf6153b5eb95069517e833

Here, Week is the parent package and Child is the child package. This is the corresponding package body of Week.Child:

package body Week.Child is

  function Get_First_Of_Week return String is
  begin
    return Mon;
  end Get_First_Of_Week;

end Week.Child;

Code block metadata

MD5: 04dad82685ad9f0231c3084266b0af83

In the implementation of the Get_First_Of_Week function, we can use the Mon string directly, even though it was declared in the parent package Week. We don't write with Week here because all elements from the specification of the Week package — such as Mon, Tue and so on — are visible in the child package Week.Child.

Now that we've completed the implementation of the Week.Child package, we can use elements from this child package in a subprogram by simply writing with Week.Child. Similarly, if we want to use these elements directly, we write use Week.Child in addition. For example:
with Ada.Text_IO; use Ada.Text_IO;

with Week.Child; use Week.Child;

procedure Main is
begin
  Put_Line ("First day of the week is " & Get_First_Of_Week);
end Main;

4.4.1 Child of a child package

So far, we've seen a two-level package hierarchy. But the hierarchy that we can potentially create isn't limited to that. For instance, we could extend the hierarchy of the previous source code example by declaring a Week.Child.Grandchild package. In this case, Week.Child would be the parent of the Grandchild package. Let's consider this implementation:

Listing 11: week-child-grandchild.ads

package Week.Child.Grandchild is
  function Get_Second_Of_Week return String;
end Week.Child.Grandchild;

Listing 12: week-child-grandchild.adb

package body Week.Child.Grandchild is
  function Get_Second_Of_Week return String is
    begin
      return Tue;
    end Get_Second_Of_Week;
end Week.Child.Grandchild;

We can use this new Grandchild package in our test application in the same way as before: we can reuse the previous test application and adapt the with and use, and the function call. This is the updated code:

Listing 13: main.adb

with Ada.Text_IO; use Ada.Text_IO;

(continues on next page)
with Week.Child.Grandchild;
use Week.Child.Grandchild;

procedure Main is
begin
  Put_Line ("Second day of the week is "
    & Get_Second_Of_Week);
end Main;

4.4.2 Multiple children

So far, we've seen a single child package of a parent package. However, a parent package can also have multiple children. We could extend the example above and implement a Week.Child_2 package. For example:

Listing 14: week-child_2.ads

package Week.Child_2 is
  function Get_Last_Of_Week return String;
end Week.Child_2;

Listing 15: week-child_2.adb

package body Week.Child_2 is
  function Get_Last_Of_Week return String is
    begin
      return Sun;
    end Get_Last_Of_Week;
end Week.Child_2;
We can now reference both children in our test application:

```ada
with Ada.Text_IO;    use Ada.Text_IO;
with Week.Child;    use Week.Child;
with Week.Child_2; use Week.Child_2;

procedure Main is
begin
  Put_Line ("First day of the week is ", Get_First_Of_Week);
  Put_Line ("Last day of the week is ", Get_Last_Of_Week);
end Main;
```

### 4.4.3 Visibility

In the previous section, we've seen that elements declared in a parent package specification are visible in the child package. This is, however, not the case for elements declared in the package body of a parent package.

Let's consider the package `Book` and its child `Additional_Operations`:

```ada
package Book is
  Title : constant String := "Visible for my children";
  function Get_Title return String;
  function Get_Author return String;
end Book;
```

```ada
package Book.Additional_Operations is
  function Get_Extended_Title return String;
  function Get_Extended_Author return String;
end Book.Additional_Operations;
```
Learning Ada

Code block metadata

This is the body of both packages:

Listing 19: book.adb

```ada
package body Book is

  Author : constant String :=
    "Author not visible for my children";

  function Get_Title return String is
  begin
    return Title;
  end Get_Title;

  function Get_Author return String is
  begin
    return Author;
  end Get_Author;

end Book;
```

Listing 20: book-additional_operations.adb

```ada
package body Book.Additional_Operations is

  function Get_Extended_Title return String is
  begin
    return "Book Title: " & Title;
  end Get_Extended_Title;

  function Get_Extended_Author return String is
  begin
    -- "Author" string declared in the body
    -- of the Book package is not visible
    -- here. Therefore, we cannot write:
    --
    -- return "Book Author: " & Author;

    return "Book Author: Unknown";
  end Get_Extended_Author;

end Book.Additional_Operations;
```

In the implementation of the Get_Extended_Title, we're using the Title constant from the parent package Book. However, as indicated in the comments of the Get_Extended_Author function, the Author string — which we declared in the body of the Book package — isn't visible in the Book.Additional_Operations package. Therefore, we cannot use it to implement the Get_Extended_Author function.

We can, however, use the Get_Author function from Book in the implementation of the Get_Extended_Author function to retrieve this string. Likewise, we can use this strategy to implement the Get_Extended_Title function. This is the adapted code:
Listing 21: book-additional_operations.adb

```ada
package body Book.Additional_Operations is

  function Get_Extended_Title return String is
  begin
    return "Book Title: " & Get_Title;
  end Get_Extended_Title;

  function Get_Extended_Author return String is
  begin
    return "Book Author: " & Get_Author;
  end Get_Extended_Author;

end Book.Additional_Operations;
```

This is a simple test application for the packages above:

Listing 22: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Book.Additional_Operations;
use Book.Additional_Operations;

procedure Main is
begin
  Put_Line (Get_Extended_Title);
  Put_Line (Get_Extended_Author);
end Main;
```

By declaring elements in the body of a package, we can implement encapsulation in Ada. Those elements will only be visible in the package body, but nowhere else. This isn't, however, the only way to achieve encapsulation in Ada: we'll discuss other approaches in the *Privacy* (page 113) chapter.

4.4. Child packages
4.5 Renaming

Previously, we’ve mentioned that subprograms can be renamed (page 32). We can rename packages, too. Again, we use the renames keyword for that. The following example renames the Ada.Text_IO package as TIO:

Listing 23: main.adb

```ada
with Ada.Text_IO;

procedure Main is
    package TIO renames Ada.Text_IO;
begin
    TIO.Put_Line ("Hello");
end Main;
```

We can use renaming to improve the readability of our code by using shorter package names. In the example above, we write TIO.Put_Line instead of the longer version (Ada.Text_IO.Put_Line). This approach is especially useful when we don’t use packages and want to avoid that the code becomes too verbose.

Note we can also rename subprograms and objects inside packages. For instance, we could have just renamed the Put_Line procedure in the source code example above:

Listing 24: main.adb

```ada
with Ada.Text_IO;

procedure Main is
    procedure Say (Something : String) renames Ada.Text_IO.Put_Line;
begin
    Say ("Hello");
end Main;
```

In this example, we rename the Put_Line procedure to Say.
Ada is a strongly typed language. It is interestingly modern in that respect: strong static typing has been increasing in popularity in programming language design, owing to factors such as the growth of statically typed functional programming, a big push from the research community in the typing domain, and many practical languages with strong type systems.

5.1 What is a type?

In statically typed languages, a type is mainly (but not only) a compile time construct. It is a construct to enforce invariants about the behavior of a program. Invariants are unchangeable properties that hold for all variables of a given type. Enforcing them ensures, for example, that variables of a data type never have invalid values.

A type is used to reason about the objects a program manipulates (an object is a variable or a constant). The aim is to classify objects by what you can accomplish with them (i.e., the operations that are permitted), and this way you can reason about the correctness of the objects' values.

5.2 Integers

A nice feature of Ada is that you can define your own integer types, based on the requirements of your program (i.e., the range of values that makes sense). In fact, the definitional mechanism that Ada provides forms the semantic basis for the predefined integer types. There is no "magical" built-in type in that regard, which is unlike most languages, and arguably very elegant.

Listing 1: integer_type_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Integer_Type_Example is
   -- Declare a signed integer type,
   -- and give the bounds
   type My_Int is range -1 .. 20;
   -- ^ High bound
   -- ^ Low bound
   -- Like variables, type declarations can
   -- only appear in declarative regions.
begin
   for I in My_Int loop
      Put_Line (My_Int'Image (I));
      -- 'Image attribute
```

(continues on next page)
-- converts a value
-- to a String.
end loop;
end Integer_Type_Example;

This example illustrates the declaration of a signed integer type, and several things we can
do with them.

Every type declaration in Ada starts with the type keyword (except for task types (page 161)). After the type, we can see a range that looks a lot like the ranges that we use in for loops, that defines the low and high bound of the type. Every integer in the inclusive range of the bounds is a valid value for the type.

Ada integer types

In Ada, an integer type is not specified in terms of its machine representation, but rather by its range. The compiler will then choose the most appropriate representation.

Another point to note in the above example is the My_Int'Image (I) expression. The Name'Attribute (optional params) notation is used for what is called an attribute in Ada. An attribute is a built-in operation on a type, a value, or some other program entity. It is accessed by using a ' symbol (the ASCII apostrophe).

Ada has several types available as "built-ins"; Integer is one of them. Here is how Integer might be defined for a typical processor:

type Integer is
  range -(2 ** 31) .. +(2 ** 31 - 1);

** is the exponent operator, which means that the first valid value for Integer is \(-2^{31}\), and the last valid value is \(2^{31} - 1\).
Ada does not mandate the range of the built-in type `Integer`. An implementation for a 16-bit target would likely choose the range \(-2^{15}\) through \(2^{15} - 1\).

### 5.2.1 Operational semantics

Unlike some other languages, Ada requires that operations on integers should be checked for overflow.

Listing 2: main.adb

```ada
procedure Main is
   A : Integer := Integer'Last;
   B : Integer;
begin
   B := A + 5;  -- This operation will overflow, eg. it
   -- will raise an exception at run time.
end Main;
```

#### Code block metadata


MD5: bddd15b394f043442024899d12b982fb

#### Build output

main.adb:5:11: warning: value not in range of type "Standard.Integer" [enabled by default]
main.adb:5:11: warning: Constraint_Error will be raised at run time [enabled by default]

#### Runtime output

raised CONSTRAINT_ERROR : main.adb:5 overflow check failed

There are two types of overflow checks:

- Machine-level overflow, when the result of an operation exceeds the maximum value (or is less than the minimum value) that can be represented in the storage reserved for an object of the type, and
- Type-level overflow, when the result of an operation is outside the range defined for the type.

Mainly for efficiency reasons, while machine-level overflow always results in an exception, type-level overflows will only be checked at specific boundaries, like assignment:

Listing 3: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is  
   type My_Int is range 1 .. 20;
   A : My_Int := 12;
   B : My_Int := 15;
   M : My_Int := (A + B) / 2;
   -- No overflow here, overflow checks
   -- are done at specific boundaries.
begin
   for I in 1 .. M loop
```

(continues on next page)
Learning Ada

12      Put_Line ("Hello, World!");
13    end loop;
14    -- Loop body executed 13 times
15    end Main;

Code block metadata

MD5: d24283cbb42c0be5b5fa215eb16ad2e7

Runtime output

Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!

Type-level overflow will only be checked at specific points in the execution. The result, as we see above, is that you might have an operation that overflows in an intermediate computation, but no exception will be raised because the final result does not overflow.

5.3 Unsigned types

Ada also features unsigned Integer types. They’re called **modular** types in Ada parlance. The reason for this designation is due to their behavior in case of overflow: They simply "wrap around", as if a modulo operation was applied.

For machine sized modular types, for example a modulus of $2^{32}$, this mimics the most common implementation behavior of unsigned types. However, an advantage of Ada is that the modulus is more general:

Listing 4: main.adb

1         with Ada.Text_IO; use Ada.Text_IO;
2        procedure Main is
3          type Mod_Int is mod 2 ** 5;
4            -- ^ Range is 0 .. 31
5          A : constant Mod_Int := 20;
6          B : constant Mod_Int := 15;
7          M : constant Mod_Int := A + B;
8            -- No overflow here,
9            -- M = (20 + 15) mod 32 = 3
10         begin
11            for I in 1 .. M loop
12                Put_Line ("Hello, World!");
13            end loop;
14            -- Loop body executed 13 times
15         end Main;
Unlike in C/C++, since this wraparound behavior is guaranteed by the Ada specification, you can rely on it to implement portable code. Also, being able to leverage the wrapping on arbitrary bounds is very useful — the modulus does not need to be a power of 2 — to implement certain algorithms and data structures, such as ring buffers\(^\text{10}\).

### 5.4 Enumerations

Enumeration types are another nicety of Ada's type system. Unlike C's enums, they are not integers, and each new enumeration type is incompatible with other enumeration types. Enumeration types are part of the bigger family of discrete types, which makes them usable in certain situations that we will describe later but one context that we have already seen is a case statement.

#### Listing 5: enumeration_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Enumeration_Example is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday,
                Saturday, Sunday);
  -- An enumeration type
begin
  for I in Days loop
    case I is
      when Saturday .. Sunday =>
        Put_Line ("Week end!");
      when Monday .. Friday =>
        Put_Line ("Hello on 
                   & Days'Image (I));
        -- 'Image attribute, works on
        -- enums too
      end case;
  end loop;
end Enumeration_Example;
```

---

### 5.4. Enumerations

---

Hello on MONDAY
Hello on TUESDAY
Hello on WEDNESDAY
Hello on THURSDAY
Hello on FRIDAY
Week end!
Week end!

Enumeration types are powerful enough that, unlike in most languages, they’re used to
define the standard Boolean type:

```ada
type Boolean is (False, True);
```

As mentioned previously, every "built-in" type in Ada is defined with facilities generally
available to the user.

## 5.5 Floating-point types

### 5.5.1 Basic properties

Like most languages, Ada supports floating-point types. The most commonly used floating-
point type is `Float`:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Floating_Point_Demo is
  A : constant Float := 2.5;
begin
  Put_Line ("The value of A is " & Float'Image (A));
end Floating_Point_Demo;
```

The application will display `2.5` as the value of A.

The Ada language does not specify the precision (number of decimal digits in the mantissa)
for Float; on a typical 32-bit machine the precision will be 6.

All common operations that could be expected for floating-point types are available, includ-
ing absolute value and exponentiation. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Floating_Point_Operations is
  A : Float := 2.5;
begin
  (continues on next page)
```
A := abs (A - 4.5);
Put_Line ("The value of A is 
& Float'Image (A));

A := A ** 2 + 1.0;
Put_Line ("The value of A is 
& Float'Image (A));
end Floating_Point_Operations;

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Floating_Point_Operations
MD5: c280e0f23e020aaee1a8777e7fb4c242

Runtime output

The value of A is 2.00000E+00
The value of A is 5.00000E+00

The value of A is 2.0 after the first operation and 5.0 after the second operation.

In addition to Float, an Ada implementation may offer data types with higher precision such as Long_Float and Long_Long_Float. Like Float, the standard does not indicate the exact precision of these types: it only guarantees that the type Long_Float, for example, has at least the precision of Float. In order to guarantee that a certain precision requirement is met, we can define custom floating-point types, as we will see in the next section.

5.5.2 Precision of floating-point types

Ada allows the user to specify the precision for a floating-point type, expressed in terms of decimal digits. Operations on these custom types will then have at least the specified precision. The syntax for a simple floating-point type declaration is:

type T is digits <number_of_decimal_digits>;

The compiler will choose a floating-point representation that supports the required precision. For example:

Listing 8: custom_floating_types.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Custom_Floating_Types is
  type T3 is digits 3;
  type T15 is digits 15;
  type T18 is digits 18;
begin
  Put_Line ("T3 requires 
& Integer'Image (T3'Size)
& " bits");
  Put_Line ("T15 requires 
& Integer'Image (T15'Size)
& " bits");
  Put_Line ("T18 requires 
& Integer'Image (T18'Size)
& " bits");
end Custom_Floating_Types;

Code block metadata

5.5. Floating-point types
In this example, the attribute 'Size is used to retrieve the number of bits used for the specified data type. As we can see by running this example, the compiler allocates 32 bits for T3, 64 bits for T15 and 128 bits for T18. This includes both the mantissa and the exponent.

The number of digits specified in the data type is also used in the format when displaying floating-point variables. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_Custom_Floating_Types is
  type T3 is digits 3;
  type T18 is digits 18;
  C1 : constant := 1.0e-4;
  A : constant T3 := 1.0 + C1;
  B : constant T18 := 1.0 + C1;
begin
  Put_Line ("The value of A is " & T3'Image (A));
  Put_Line ("The value of B is " & T18'Image (B));
end Display_Custom_Floating_Types;
```

As expected, the application will display the variables according to specified precision (1.00E+00 and 1.00010000000000000E+00).

### 5.5.3 Range of floating-point types

In addition to the precision, a range can also be specified for a floating-point type. The syntax is similar to the one used for integer data types — using the range keyword. This simple example creates a new floating-point type based on the type Float, for a normalized range between -1.0 and 1.0:
Learning Ada

Listing 10: floating_point_range.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Floating_Point_Range is
  type T_Norm is new Float range -1.0 .. 1.0;
  A : T_Norm;

begin
  A := 1.0;
  Put_Line ("The value of A is "
               & T_Norm'Image (A));
end Floating_Point_Range;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Floating_Point_Range
MD5: b43d596682aa0fa11l24a3a3d0596abc

Runtime output

The value of A is 1.00000E+00

The application is responsible for ensuring that variables of this type stay within this range; otherwise an exception is raised. In this example, the exception Constraint_Error is raised when assigning 2.0 to the variable A:

Listing 11: floating_point_range_exception.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Floating_Point_Range_Exception is
  type T_Norm is new Float range -1.0 .. 1.0;
  A : T_Norm;

begin
  A := 2.0;
  Put_Line ("The value of A is "
               & T_Norm'Image (A));
end Floating_Point_Range_Exception;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Floating_Point_Range_Except
MD5: ecda66589ba28e453956dca159ea5f0d

Build output

floating_point_range_exception.adb:7:09: warning: value not in range of type "T_Norm" defined at line 4 [enabled by default]
floating_point_range_exception.adb:7:09: warning: Constraint_Error will be raised when value is assigned at run time [enabled by default]

Runtime output

raised CONSTRAINT_ERROR : floating_point_range_exception.adb:7 range check failed

Ranges can also be specified for custom floating-point types. For example:

5.5. Floating-point types
5.6 Strong typing

As noted earlier, Ada is strongly typed. As a result, different types of the same family are incompatible with each other; a value of one type cannot be assigned to a variable from the other type. For example:

Listing 13: illegal_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Illegal_Example is
  -- Declare two different floating point types
  type Meters is new Float;
  type Miles is new Float;

  Dist_Impperial : Miles;

  -- Declare a constant
  Dist_Metric : constant Meters := 1000.0;

begin
  -- Not correct: types mismatch
  Dist_Impperial := Dist_Metric * 621.371e-6;
  Put_Line (Miles'Image (Dist_Impperial));
end Illegal_Example;
```

A consequence of these rules is that, in the general case, a "mixed mode" expression like 2 * 3.0 will trigger a compilation error. In a language like C or Python, such expressions are made valid by implicit conversions. In Ada, such conversions must be made explicit:
Learning Ada

Listing 14: conv.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Conv is
    type Meters is new Float;
    type Miles is new Float;
    Dist_Impperial : Miles;
    Dist_Metric : constant Meters := 1000.0;
    begin
        Dist_Impperial := Miles (Dist_Metric) * 621.371e-6;
        -- ^^^^^^^^^^^^^^^^^
        -- Type conversion, from Meters to Miles
        -- Now the code is correct
        Put_Line (Miles'Image (Dist_Impperial));
    end Conv;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Imperial_Metric
MD5: e455641e86227e80e5f920b5af6315d4

Runtime output

6.21371E-01

Of course, we probably do not want to write the conversion code every time we convert from meters to miles. The idiomatic Ada way in that case would be to introduce conversion functions along with the types.

Listing 15: conv.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Conv is
    type Meters is new Float;
    type Miles is new Float;
    -- Function declaration, like procedure
    -- but returns a value.
    function To_Miles (M : Meters) return Miles is
        -- ^ Return type
        begin
            return Miles (M) * 621.371e-6;
        end To_Miles;
    Dist_Impperial : Miles;
    Dist_Metric : constant Meters := 1000.0;
    begin
        Dist_Impperial := To_Miles (Dist_Metric);
        Put_Line (Miles'Image (Dist_Impperial));
    end Conv;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Imperial_Metric_Func
MD5: 661737fa9f130ac3070210bbf6f08214

Runtime output

5.6. Strong typing
If you write a lot of numeric code, having to explicitly provide such conversions might seem painful at first. However, this approach brings some advantages. Notably, you can rely on the absence of implicit conversions, which will in turn prevent some subtle errors.

In other languages

In C, for example, the rules for implicit conversions may not always be completely obvious. In Ada, however, the code will always do exactly what it seems to do. For example:

```ada
int a = 3, b = 2;
float f = a / b;
```

This code will compile fine, but the result of f will be 1.0 instead of 1.5, because the compiler will generate an integer division (three divided by two) that results in one. The software developer must be aware of data conversion issues and use an appropriate casting:

```ada
int a = 3, b = 2;
float f = (float)a / b;
```

In the corrected example, the compiler will convert both variables to their corresponding floating-point representation before performing the division. This will produce the expected result.

This example is very simple, and experienced C developers will probably notice and correct it before it creates bigger problems. However, in more complex applications where the type declaration is not always visible — e.g. when referring to elements of a struct — this situation might not always be evident and quickly lead to software defects that can be harder to find.

The Ada compiler, in contrast, will always reject code that mixes floating-point and integer variables without explicit conversion. The following Ada code, based on the erroneous example in C, will not compile:

```ada
procedure Main is
   A : Integer := 3;
   B : Integer := 2;
   F : Float;
begin
   F := A / B;
end Main;
```

Listing 16: main.adb

```
procedure Main is
   A : Integer := 3;
   B : Integer := 2;
   F : Float;
begin
   F := A / B;
end Main;
```

The offending line must be changed to

```ada
F := Float (A) / Float (B);
```

in order to be accepted by the compiler.
You can use Ada's strong typing to help enforce invariants in your code, as in the example above: Since Miles and Meters are two different types, you cannot mistakenly convert an instance of one to an instance of the other.

### 5.7 Derived types

In Ada you can create new types based on existing ones. This is very useful: you get a type that has the same properties as some existing type but is treated as a distinct type in the interest of strong typing.

Listing 17: main.adb

```ada
procedure Main is
  -- ID card number type,
  -- incompatible with Integer.
  type Social_Security_Number is new Integer
    range 0 .. 999_99_9999;

  SSN : Social_Security_Number := 555_55_5555;
  -- ^ Since a SSN has 9 digits
  -- max., and cannot be
  -- negative, we enforce
  -- a validity constraint.

  I : Integer;

  -- The value -1 below will cause a
  -- runtime error and a compile time
  -- warning with GNAT.
  Invalid : Social_Security_Number := -1;
begin
  -- Illegal, they have different types:
  I := SSN;

  -- Likewise illegal:
  SSN := I;

  -- OK with explicit conversion:
  I := Integer (SSN);

  -- Likewise OK:
  SSN := Social_Security_Number (I);
end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Derived_Types
MD5: 63445601ddb5e52dceab095d3305623a

Build output

```
main.adb:21:40: warning: value not in range of type "Social_Security_Number" defined at line 4 [enabled by default]
main.adb:21:40: warning: Constraint_Error will be raised at run time [enabled by default]
```

(continues on next page)
Learning Ada

main.adb:24:09: error: expected type "Standard.Integer"
main.adb:24:09: error: found type "Social_Security_Number" defined at line 4
main.adb:27:11: error: expected type "Social_Security_Number" defined at line 4
main.adb:27:11: error: found type "Standard.Integer"
gprbuild: *** compilation phase failed

The type Social_Security is said to be a derived type; its parent type is Integer.

As illustrated in this example, you can refine the valid range when defining a derived scalar type (such as integer, floating-point and enumeration).

The syntax for enumerations uses the range <range> syntax:

Listing 18: greet.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
               Thursday, Friday,
               Saturday, Sunday);

  type Weekend_Days is new
    Days range Saturday .. Sunday;
    -- New type, where only Saturday and Sunday
    -- are valid literals.

begin
  null;
end Greet;
```

5.8 Subtypes

As we are starting to see, types may be used in Ada to enforce constraints on the valid range of values. However, we sometimes want to enforce constraints on some values while staying within a single type. This is where subtypes come into play. A subtype does not introduce a new type.

Listing 19: greet.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
  subtype Weekend_Days is
    Days range Saturday .. Sunday;
    -- ^ Constraint of the subtype
  M : Days := Sunday;
```

(continues on next page)
S : Weekend_Days := M;
-- No error here, Days and Weekend_Days
-- are of the same type.
begin
for I in Days loop
  case I is
    when Weekend_Days =>
      Put_Line ("Week end!");
    when others =>
      Put_Line ("Hello on " & Days'Image (I));
  end case;
end loop;
end Greet;

Several subtypes are predefined in the standard package in Ada, and are automatically available to you:

\[
\text{subtype Natural is Integer range } 0 \ldots \text{ Integer'Last;}
\]
\[
\text{subtype Positive is Integer range } 1 \ldots \text{ Integer'Last;}
\]

While subtypes of a type are statically compatible with each other, constraints are enforced at run time: if you violate a subtype constraint, an exception will be raised.

Listing 20: greet.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
               Thursday, Friday,
               Saturday, Sunday);

  subtype Weekend_Days is
    Days range Saturday .. Sunday;

  Day : Days := Saturday;
  Weekend : Weekend_Days;
begin
  Weekend := Day;
  -- Correct: Same type, subtype constraints are respected
  Weekend := Monday;
  -- Wrong value for the subtype

5.8. Subtypes
Learning Ada

(continued from previous page)

```ada
19    -- Compiles, but exception at runtime
20 end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Days_Subtype_Error
MD5: 84d42d276d26544f35edab5870459378

Build output

```
greet.adb:17:15: warning: value not in range of type "Weekend_Days" defined at, line 8 [enabled by default]
greet.adb:17:15: warning: Constraint_Error will be raised at run time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : greet.adb:17 range check failed
```

5.8.1 Subtypes as type aliases

Previously, we've seen that we can create new types by declaring e.g. `type Miles is new Float`. We could also create type aliases, which generate alternative names — *aliases* — for known types. Note that type aliases are sometimes called *type synonyms*.

We achieve this in Ada by using subtypes without new constraints. In this case, however, we don't get all of the benefits of Ada's strong type checking. Let's rewrite an example using type aliases:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Undetected_Imperial_Metric_Error is
    -- Declare two type aliases
    subtype Meters is Float;
    subtype Miles is Float;

    Dist_Imperial : Miles;

    -- Declare a constant
    Dist_Metric : constant Meters := 100.0;

begin
    -- No conversion to Miles type required:
    Dist_Imperial := (Dist_Metric * 1609.0) / 1000.0;

    -- Not correct, but undetected:
    Dist_Imperial := Dist_Metric;

    Put_Line (Miles'Image (Dist_Imperial));
end Undetected_Imperial_Metric_Error;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Undetected_Imperial_Metric_Error
MD5: cdb8f949c69f3c480502b859dac298ee
In the example above, the fact that both Meters and Miles are subtypes of Float allows us to mix variables of both types without type conversion. This, however, can lead to all sorts of programming mistakes that we'd like to avoid, as we can see in the undetected error highlighted in the code above. In that example, the error in the assignment of a value in meters to a variable meant to store values in miles remains undetected because both Meters and Miles are subtypes of Float. Therefore, the recommendation is to use strong typing — via type X is new Y — for cases such as the one above.

There are, however, many situations where type aliases are useful. For example, in an application that uses floating-point types in multiple contexts, we could use type aliases to indicate additional meaning to the types or to avoid long variable names. For example, instead of writing:

```plaintext
Paid_Amount, Due_Amount : Float;
```

We could write:

```plaintext
subtype Amount is Float;
Paid, Due : Amount;
```

### In other languages

In C, for example, we can use a typedef declaration to create a type alias. For example:

```plaintext
typedef float meters;
```

This corresponds to the declaration that we've seen above using subtypes. Other programming languages include this concept in similar ways. For example:

- C++: using meters = float;
- Swift: typealias Meters = Double
- Kotlin: typealias Meters = Double
- Haskell: type Meters = Float

Note, however, that subtypes in Ada correspond to type aliases if, and only if, they don't have new constraints. Thus, if we add a new constraint to a subtype declaration, we don't have a type alias anymore. For example, the following declaration can't be considered a type alias of Float:

```plaintext
subtype Meters is Float range 0.0 .. 1_000_000.0;
```

Let's look at another example:

```plaintext
subtype Degree_Celsius is Float;
```

```plaintext
subtype Liquid_Water_Temperature is
    Degree_Celsius range 0.0 .. 100.0;
```

```plaintext
subtype Running_Water_Temperature is
    Liquid_Water_Temperature;
```

In this example, Liquid_Water_Temperature isn't an alias of Degree_Celsius, since it adds a new constraint that wasn't part of the declaration of the Degree_Celsius. However, we do have two type aliases here:
• Degree_Celsius is an alias of \texttt{Float};
• Running_Water_Temperature is an alias of Liquid_Water_Temperature, even if Liquid_Water_Temperature itself has a constrained range.
So far, all the types we have encountered have values that are not decomposable: each instance represents a single piece of data. Now we are going to see our first class of composite types: records.

Records allow composing a value out of instances of other types. Each of those instances will be given a name. The pair consisting of a name and an instance of a specific type is called a field, or a component.

**6.1 Record type declaration**

Here is an example of a simple record declaration:

```haskell
<table>
<thead>
<tr>
<th>type Date is record</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- The following declarations are components of the record</td>
</tr>
<tr>
<td>Day : Integer range 1 .. 31;</td>
</tr>
<tr>
<td>Month : Months;</td>
</tr>
<tr>
<td>-- You can add custom constraints on fields</td>
</tr>
<tr>
<td>Year : Integer range 1 .. 3000;</td>
</tr>
<tr>
<td>end record;</td>
</tr>
</tbody>
</table>
```

Fields look a lot like variable declarations, except that they are inside of a record definition. And as with variable declarations, you can specify additional constraints when supplying the subtype of the field.

```haskell
<table>
<thead>
<tr>
<th>type Date is record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day : Integer range 1 .. 31;</td>
</tr>
<tr>
<td>Month : Months := January;</td>
</tr>
<tr>
<td>-- This component has a default value</td>
</tr>
<tr>
<td>Year : Integer range 1 .. 3000 := 2012;</td>
</tr>
<tr>
<td>-- Default value</td>
</tr>
<tr>
<td>end record;</td>
</tr>
</tbody>
</table>
```

Record components can have default values. When a variable having the record type is declared, a field with a default initialization will be automatically set to this value. The value can be any expression of the component type, and may be run-time computable.

In the remaining sections of this chapter, we see how to use record types. In addition to that, we discuss more about records in another chapter (page 101).
6.2 Aggregates

-- Positional components
Ada_Birthday : Date := (10, December, 1815);

-- Named components
Leap_Day_2020 : Date := (Day => 29, Month => February, Year => 2020);

Records have a convenient notation for expressing values, illustrated above. This notation is called aggregate notation, and the literals are called aggregates. They can be used in a variety of contexts that we will see throughout the course, one of which is to initialize records.

An aggregate is a list of values separated by commas and enclosed in parentheses. It is allowed in any context where a value of the record is expected.

Values for the components can be specified positionally, as in Ada_Birthday example, or by name, as in Leap_Day_2020. A mixture of positional and named values is permitted, but you cannot use a positional notation after a named one.

6.3 Component selection

To access components of a record instance, you use an operation that is called component selection. This is achieved by using the dot notation. For example, if we declare a variable Some_Day of the Date record type mentioned above, we can access the Year component by writing Some_Day.Year.

Let's look at an example:

Listing 1: record_selection.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Record_Selection is

  type Months is
  (January, February, March, April,
   May, June, July, August, September,
   October, November, December);

  type Date is record
  Day : Integer range 1 .. 31;
  Month : Months;
  Year : Integer range 1 .. 3000 := 2032;
  end record;

  procedure Display_Date (D : Date) is
  begin
    Put_Line ("Day:" & Integer'Image (D.Day) & ", Month:" & Months'Image (D.Month) & ", Year:" & Integer'Image (D.Year));
  end Display_Date;

  Some_Day : Date := (1, January, 2000);
```

(continues on next page)
begin
  Display_Date (Some_Day);
  Put_Line ("Changing year...");
  Some_Day.Year := 2001;
  Display_Date (Some_Day);
end Record_Selection;

As you can see in this example, we can use the dot notation in the expression D.Year or Some_Day.Year to access the information stored in that component, as well as to modify this information in assignments. To be more specific, when we use D.Year in the call to Put_Line, we're retrieving the information stored in that component. When we write Some_Day.Year := 2001, we're overwriting the information that was previously stored in the Year component of Some_Day.

### 6.4 Renaming

In previous chapters, we've discussed subprogram (page 32) and package (page 46) renaming. We can rename record components as well. Instead of writing the full component selection using the dot notation, we can declare an alias that allows us to access the same component. This is useful to simplify the implementation of a subprogram, for example.

We can rename record components by using the `renames` keyword in a variable declaration. For example:

```ada
Some_Day : Date;
Y : Integer renames Some_Day.Year;
```

Here, Y is an alias, so that every time we using Y, we are really using the Year component of Some_Day.

Let's look at a complete example:

Listing 2: dates.ads

```ada
package Dates is
  type Months is
    (January, February, March, April,
     May, June, July, August, September,
     October, November, December);
  type Date is record
    Day : Integer range 1 .. 31;
    Month : Months;
end Dates;
```

(continues on next page)
Year : Integer range 1 .. 3000 := 2032;
end record;

procedure Increase_Month
(Some_Day : in out Date);

procedure Display_Month
(Some_Day : Date);
end Dates;

Listing 3: dates.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Dates is

procedure Increase_Month
(Some_Day : in out Date)
is
    -- Renaming components from
    -- the Date record
    M : Months renames Some_Day.Month;
    Y : Integer renames Some_Day.Year;
    -- Renaming function (for Months
    -- enumeration)
    function Next (M : Months)
        return Months
            renames Months'_succ;
    begin
        if M = December then
            M := January;
            Y := Y + 1;
        else
            M := Next (M);
        end if;
    end Increase_Month;

procedure Display_Month
(Some_Day : Date)
is
    -- Renaming components from
    -- the Date record
    M : Months renames Some_Day.Month;
    Y : Integer renames Some_Day.Year;
    begin
        Put_Line ("Month: ",
            Months'Image (M)
            & ", Year:",
            Integer'Image (Y));
    end Display_Month;
end Dates;

Listing 4: main.adb

with Ada.Text_IO; use Ada.Text_IO;
with Dates; use Dates;
procedure Main is
D : Date := (1, January, 2000);
begin
   Display_Month (D);
   Put_Line ("Increasing month...");
   Increase_Month (D);
   Display_Month (D);
end Main;

Code block metadata
Project: Courses.Intro To Ada.Arrays.Record_Component_Renaming
MD5: 905390bd02b8417039052218800975a3

Runtime output
Month: JANUARY, Year: 2000
Increasing month...
Month: FEBRUARY, Year: 2000

We apply renaming to two components of the Date record in the implementation of the Increase_Month procedure. Then, instead of directly using Some_Day.Month and Some_Day.Year in the next operations, we simply use the renamed versions M and Y.

Note that, in the example above, we also rename Months.'Succ' — which is the function that gives us the next month — to Next.
Arrays provide another fundamental family of composite types in Ada.

### 7.1 Array type declaration

Arrays in Ada are used to define contiguous collections of elements that can be selected by indexing. Here's a simple example:

Listing 1: greet.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
  type My_Int is range 0 .. 1000;
  type Index is range 1 .. 5;

  type My_Int_Array is
    array (Index) of My_Int;
  -- ^ Type of elements
  -- ^ Bounds of the array
  Arr : My_Int_Array := (2, 3, 5, 7, 11);
  -- ^ Array literal
  -- (aggregate)

  V : My_Int;

begin
  for I in Index loop
    V := Arr (I);
    -- ^ Take the Ith element
    Put (My_Int'Image (V));
  end loop;
  New_Line;
end Greet;
```

The first point to note is that we specify the index type for the array, rather than its size. Here we declared an integer type named Index ranging from 1 to 5, so each array instance will have 5 elements, with the initial element at index 1 and the last element at index 5.
Although this example used an integer type for the index, Ada is more general: any discrete type is permitted to index an array, including *Enum types* (page 51). We will soon see what that means.

Another point to note is that querying an element of the array at a given index uses the same syntax as for function calls: that is, the array object followed by the index in parentheses. Thus when you see an expression such as A (B), whether it is a function call or an array subscript depends on what A refers to.

Finally, notice how we initialize the array with the (2, 3, 5, 7, 11) expression. This is another kind of aggregate in Ada, and is in a sense a literal expression for an array, in the same way that 3 is a literal expression for an integer. The notation is very powerful, with a number of properties that we will introduce later. A detailed overview appears in the notation of *aggregate types* (page 89).

Unrelated to arrays, the example also illustrated two procedures from *Ada.Text_IO*:

- *Put*, which displays a string without a terminating end of line
- *New_Line*, which outputs an end of line

Let’s now delve into what it means to be able to use any discrete type to index into the array.

### In other languages

Semantically, an array object in Ada is the entire data structure, and not simply a handle or pointer. Unlike C and C++, there is no implicit equivalence between an array and a pointer to its initial element.

#### Listing 2: array_bounds_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Array_Bounds_Example is
  type My_Int is range 0 .. 1000;
  type Index is range 11 .. 15;
  -- ^ Low bound can
  -- be any value
  type My_Int_Array is
    array (Index) of My_Int;
  Tab : constant My_Int_Array :=
    (2, 3, 5, 7, 11);
begin
  for I in Index loop
    Put (My_Int'Image (Tab (I)));
  end loop;
  New_Line;
end Array_Bounds_Example;
```

#### Code block metadata

- **Project**: Courses.Intro_To_Ada.Arrays.Array_Bounds_Example
- **MD5**: e5fe9e7b83055f3ae23dd890e29c22de

#### Runtime output

```
2 3 5 7 11
```
One effect is that the bounds of an array can be any values. In the first example we constructed an array type whose first index is 1, but in the example above we declare an array type whose first index is 11.

That's perfectly fine in Ada, and moreover since we use the index type as a range to iterate over the array indices, the code using the array does not need to change.

That leads us to an important consequence with regard to code dealing with arrays. Since the bounds can vary, you should not assume / hard-code specific bounds when iterating / using arrays. That means the code above is good, because it uses the index type, but a for loop as shown below is bad practice even though it works correctly:

```ada
for I in 11 .. 15 loop
  Tab (I) := Tab (I) + 2;
end loop;
```

Since you can use any discrete type to index an array, enumeration types are permitted.

Listing 3: month_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Month_Example is
  type Month_Duration is range 1 .. 31;
  type Month is (Jan, Feb, Mar, Apr,
                May, Jun, Jul, Aug,
                Sep, Oct, Nov, Dec);

type My_Int_Array is
  array (Month) of Month_Duration;

Tab : constant My_Int_Array :=
  (31, 28, 31, 30, 31, 30,
   31, 31, 30, 31, 30, 31);

Feb_Days : Month_Duration := Tab (Feb);   -- Maps months to number of days
begin
  for M in Month loop
    Put_Line
      (Month'Image (M) & " has " & Month_Duration'Image (Tab (M)) & " days.");
  end loop;
end Month_Example;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Month_Example
MD5: 420bb8faa36d0efd3d071c76c2033d21

Runtime output

JAN has 31 days.
FEB has 28 days.
MAR has 31 days.

(continues on next page)

7.1. Array type declaration
In the example above, we are:

- Creating an array type mapping months to month durations in days.
- Creating an array, and instantiating it with an aggregate mapping months to their actual durations in days.
- Iterating over the array, printing out the months, and the number of days for each.

Being able to use enumeration values as indices is very helpful in creating mappings such as shown above one, and is an often used feature in Ada.

### 7.2 Indexing

We have already seen the syntax for selecting elements of an array. There are however a few more points to note.

First, as is true in general in Ada, the indexing operation is strongly typed. If you use a value of the wrong type to index the array, you will get a compile-time error.

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
  type My_Int is range 0 .. 1000;
  type My_Index is range 1 .. 5;
  type Your_Index is range 1 .. 5;

  type My_Int_Array is
    array (My_Index) of My_Int;
  Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
  for I in Your_Index loop
    Put (My_Int'Image (Tab (I)));
    -- ^ Compile time error
  end loop;
end Greet;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Arrays.Greet_2
MD5: 54543017e4ec69d24bf9e43d507b50e6

**Build output**
Second, arrays in Ada are bounds checked. This means that if you try to access an element outside of the bounds of the array, you will get a run-time error instead of accessing random memory as in unsafe languages.

Listing 5: greet.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
    type My_Int is range 0 .. 1000;
    type Index is range 1 .. 5;

    type My_Int_Array is
        array (Index) of My_Int;

    Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
    for I in Index range 2 .. 6 loop
        Put (My_Int'Image (Tab (I)));
        -- ^ Will raise an exception when I = 6
    end loop;
    New_Line;
end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Greet_3
MD5: 0102674d089be838f1dfbf0791d99fce

Build output

greet.adb:12:30: warning: static value out of range of type "Index" defined at line 5 [enabled by default]
greet.adb:12:30: warning: Constraint_Error will be raised at run time [enabled by default]
greet.adb:12:30: warning: suspicious loop bound out of range of loop subtype [enabled by default]
greet.adb:12:30: warning: loop executes zero times or raises Constraint_Error [enabled by default]

Runtime output

raised CONSTRAINT_ERROR : greet.adb:12 range check failed
7.3 Simpler array declarations

In the previous examples, we have always explicitly created an index type for the array. While this can be useful for typing and readability purposes, sometimes you simply want to express a range of values. Ada allows you to do that, too.

Listing 6: simple_array_bounds.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Simple_Array_Bounds is
  type My_Int is range 0 .. 1000;
  type My_Int_Array is array (1 .. 5) of My_Int;
  -- ^ Subtype of Integer
  Tab : constant My_Int_Array :=
    (2, 3, 5, 7, 11);

begin
  for I in 1 .. 5 loop
    -- ^ Subtype of Integer
    Put (My_Int'Image (Tab (I)));
  end loop;
  New_Line;
end Simple_Array_Bounds;
```

This example defines the range of the array via the range syntax, which specifies an anonymous subtype of Integer and uses it to index the array.

This means that the type of the index is Integer. Similarly, when you use an anonymous range in a for loop as in the example above, the type of the iteration variable is also Integer, so you can use I to index Tab.

You can also use a named subtype for the bounds for an array.

7.4 Range attribute

We noted earlier that hard coding bounds when iterating over an array is a bad idea, and showed how to use the array’s index type/subtype to iterate over its range in a for loop. That raises the question of how to write an iteration when the array has an anonymous range for its bounds, since there is no name to refer to the range. Ada solves that via several attributes of array objects:

Listing 7: range_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Range_Example is
(continues on next page)
```
```ada
begin
  for I in Tab'Range loop
    Put (My_Int'Image (Tab (I)));
  end loop;
end Range_Example;
```

**Code block metadata**

- **Project:** Courses.Intro_To_Ada.Arrays.Range_Example
- **MD5:** 8b0d7bf346cb59999d9fd12dabaaf3e2a6

**Runtime output**

```
2 3 5 7 11
```

If you want more fine grained control, you can use the separate attributes 'First and 'Last.

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Array_Attributes_Example is
  type My_Int is range 0 .. 1000;
  type My_Int_Array is array (1 .. 5) of My_Int;
  Tab : constant My_Int_Array :=
    (2, 3, 5, 7, 11);
  begin
    for I in Tab'First .. Tab'Last - 1 loop
      Put (My_Int'Image (Tab (I)));
    end loop;
    New_Line;
end Array_Attributes_Example;
```

**Code block metadata**

- **Project:** Courses.Intro_To_Ada.Arrays.Array_Attributes_Example
- **MD5:** 95cc407c8aadd936e050fe3505e8fb46

**Runtime output**

```
2 3 5 7
```

The 'Range, 'First and 'Last attributes in these examples could also have been applied to the array type name, and not just the array instances.

Although not illustrated in the above examples, another useful attribute for an array instance A is A'Length, which is the number of elements that A contains.

### 7.4. Range attribute
It is legal and sometimes useful to have a "null array", which contains no elements. To get this effect, define an index range whose upper bound is less than the lower bound.

### 7.5 Unconstrained arrays

Let's now consider one of the most powerful aspects of Ada's array facility.

Every array type we have defined so far has a fixed size: every instance of this type will have the same bounds and therefore the same number of elements and the same size.

However, Ada also allows you to declare array types whose bounds are not fixed: in that case, the bounds will need to be provided when creating instances of the type.

Listing 9: unconstrained_array_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Unconstrained_Array_Example is

  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday,
                Saturday, Sunday);

  type Workload_Type is
    array (Days range <>) of Natural;

  Workload : constant Workload_Type (Monday .. Friday) :=
    (Friday => 7, others => 8);

begin
  for I in Workload'Range loop
    Put_Line (Integer'Image (Workload (I)));
  end loop;
end Unconstrained_Array_Example;
```

The fact that the bounds of the array are not known is indicated by the `Days range <>` syntax. Given a discrete type `Discrete_Type`, if we use `Discrete_Type` for the index in an array type then `Discrete_Type` serves as the type of the index and comprises the range of index values for each array instance.

If we define the index as `Discrete_Type range <>` then `Discrete_Type` serves as the type of the index, but different array instances may have different bounds from this type.
An array type that is defined with the Discrete_Type range <> syntax for its index is referred to as an unconstrained array type, and, as illustrated above, the bounds need to be provided when an instance is created.

The above example also shows other forms of the aggregate syntax. You can specify associations by name, by giving the value of the index on the left side of an arrow association. 1 => 2 thus means "assign value 2 to the element at index 1 in my array". others => 8 means "assign value 8 to every element that wasn't previously assigned in this aggregate".

Attention: The so-called "box" notation (<> is commonly used as a wildcard or placeholder in Ada. You will often see it when the meaning is "what is expected here can be anything".

In other languages

While unconstrained arrays in Ada might seem similar to variable length arrays in C, they are in reality much more powerful, because they're truly first-class values in the language. You can pass them as parameters to subprograms or return them from functions, and they implicitly contain their bounds as part of their value. This means that it is useless to pass the bounds or length of an array explicitly along with the array, because they are accessible via the 'First, 'Last, 'Range and 'Length attributes explained earlier.

Although different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime. This allows Ada to implement unconstrained arrays efficiently; instances can be stored on the stack and do not require heap allocation as in languages like Java.

7.6 Predefined array type: String

A recurring theme in our introduction to Ada types has been the way important built-in types like Boolean or Integer are defined through the same facilities that are available to the user. This is also true for strings: The String type in Ada is a simple array.

Here is how the string type is defined in Ada:

```ada
type String is
    array (Positive range <>) of Character;
```

The only built-in feature Ada adds to make strings more ergonomic is custom literals, as we can see in the example below.

Hint: String literals are a syntactic sugar for aggregates, so that in the following example, A and B have the same value.

Listing 10: string_literals.ads

```ada
package String_Literals is
    -- Those two declarations are equivalent
    A : String (1 .. 11) := "Hello World";
    B : String (1 .. 11) :=
        ( 'H', 'e', 'l', 'l', 'o', ' ',
        'W', 'o', 'r', 'l', 'd');
end String_Literals;
```

Code block metadata

7.6. Predefined array type: String
Listing 11: greet.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   Message : String (1 .. 11) := "dlroW olleH";
   -- ^ Pre-defined array type.
   -- Component type is Character
begin
   for I in reverse Message'Range loop
      -- ^ Iterate in reverse order
      Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

However, specifying the bounds of the object explicitly is a bit of a hassle; you have to manually count the number of characters in the literal. Fortunately, Ada gives you an easier way.

You can omit the bounds when creating an instance of an unconstrained array type if you supply an initialization, since the bounds can be deduced from the initialization expression.

Listing 12: greet.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is
   Message : constant String := "dlroW olleH";
   -- ^ Bounds are automatically computed from initialization value
begin
   for I in reverse Message'Range loop
      Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Greet_5
MD5: 21448a1007a07ec9d434880628625c3f

Runtime output

Hello World

Listing 13: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Integer_Array is
      array (Natural range <>) of Integer;
   My_Array : constant Integer_Array :=
```
(continued from previous page)

7.7 Restrictions

A very important point about arrays: bounds have to be known when instances are created. It is for example illegal to do the following.

```ada
declare
  A : String;
begn
  A := "World";
end;
```

Also, while you of course can change the values of elements in an array, you cannot change the array's bounds (and therefore its size) after it has been initialized. So this is also illegal:

```ada
declare
  A : String := "Hello";
begn
  A := "World";  -- OK: Same size
  A := "Hello World";  -- Not OK: Different size
end;
```

Also, while you can expect a warning for this kind of error in very simple cases like this one, it is impossible for a compiler to know in the general case if you are assigning a value of the correct length, so this violation will generally result in a run-time error.

**Attention**

While we will learn more about this later, it is important to know that arrays are not the only types whose instances might be of unknown size at compile-time.

Such objects are said to be of an indefinite subtype, which means that the subtype size is not known at compile time, but is dynamically computed (at run time).

Listing 14: indefinite_subtypes.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Indefinite_Subtypes is
```

7.7. Restrictions
function Get_Number return Integer is
begin
  return Integer’Value (Get_Line);
end Get_Number;

A : String := "Hello";
-- Indefinite subtype

B : String (1 .. 5) := "Hello";
-- Definite subtype

C : String (1 .. Get_Number);
-- Indefinite subtype
-- (Get_Number's value is computed at
--  run-time)

begin
  null;
end Indefinite_Subtypes;

Here, the 'Value attribute converts the string to an integer.

## 7.8 Returning unconstrained arrays

The return type of a function can be any type; a function can return a value whose size is unknown at compile time. Likewise, the parameters can be of any type.

For example, this is a function that returns an unconstrained String:

Listing 15: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Days is (Monday, Tuesday, Wednesday,
               Thursday, Friday,
               Saturday, Sunday);

  function Get_Day_Name (Day : Days := Monday)
  return String is
begin
  return
    (case Day is
         when Monday => "Monday",
         when Tuesday => "Tuesday",
         when Wednesday => "Wednesday",
         when Thursday => "Thursday",
         when Friday => "Friday",
         when Saturday => "Saturday",
         when Sunday => "Sunday");
end Get_Day_Name;
```

(continues on next page)
begin
  Put_Line ("First day is 
             & Get_Day_Name (Days'First));
end Main;

Code block metadata
Project: Courses.Intro_To_Ada.Arrays.Day_Name_1
MD5: 0b7c567c723ded52d8e95c4ef46bcecc

Runtime output
First day is Monday

(This example is for illustrative purposes only. There is a built-in mechanism, the 'Image attribute for scalar types, that returns the name (as a String) of any element of an enumeration type. For example Days’Image(Monday) is "MONDAY".)

In other languages
Returning variable size objects in languages lacking a garbage collector is a bit complicated implementation-wise, which is why C and C++ don’t allow it, preferring to depend on explicit dynamic allocation / free from the user.

The problem is that explicit storage management is unsafe as soon as you want to collect unused memory. Ada's ability to return variable size objects will remove one use case for dynamic allocation, and hence, remove one potential source of bugs from your programs.

Rust follows the C/C++ model, but with safe pointer semantics. However, dynamic allocation is still used. Ada can benefit from a possible performance edge because it can use any model.

7.9 Declaring arrays (2)

While we can have array types whose size and bounds are determined at run time, the array's component type needs to be of a definite and constrained type.

Thus, if you need to declare, for example, an array of strings, the String subtype used as component will need to have a fixed size.

Listing 16: show_days.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Days is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday,
                Saturday, Sunday);

  subtype Day_Name is String (1 .. 2);
  -- Subtype of string with known size

type Days_Name_Type is
  array (Days) of Day_Name;
  -- Type of the index
  -- Type of the element.
  -- Must be definite

(continues on next page)
Names : constant Days_Name_Type :=
("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");

begin
  for I in Names'Range loop
    Put_Line (Names (I));
  end loop;
end Show_Days;

7.10 Array slices

One last feature of Ada arrays that we're going to cover is array slices. It is possible to take and use a slice of an array (a contiguous sequence of elements) as a name or a value.

Listing 17: main.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  Buf : String := "Hello ...";
  Full_Name : String := "John Smith";
begin
  Buf (7 .. 9) := "Bob";
-- Careful! This works because the string
-- on the right side is the same length as
-- the replaced slice!
  -- Prints "Hello Bob"
  Put_Line (Buf);
  -- Prints "Hi John"
  Put_Line ("Hi " & Full_Name (1 .. 4));
end Main;
As we can see above, you can use a slice on the left side of an assignment, to replace only part of an array. A slice of an array is of the same type as the array, but has a different subtype, constrained by the bounds of the slice.

**Attention:** Ada has multidimensional arrays\(^{11}\), which are not covered in this course. Slices will only work on one dimensional arrays.

### 7.11 Renaming

So far, we’ve seen that the following elements can be renamed: subprograms (page 32), packages (page 46), and record components (page 67). We can also rename objects by using the renames keyword. This allows for creating alternative names for these objects. Let’s look at an example:

Listing 18: measurements.ads

```ada
package Measurements is
    subtype Degree_Celsius is Float;
    Current_Temperature : Degree_Celsius;
end Measurements;
```

Listing 19: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Measurements;

procedure Main is
    subtype Degrees is Measurements.Degree_Celsius;
    T : Degrees
        renames Measurements.Current_Temperature;
begin
    T := 5.0;
    Put_Line (Degrees'Image (T));
    Put_Line (Degrees'Image (Measurements.Current_Temperature));
    T := T + 2.5;
    Put_Line (Degrees'Image (T));
    Put_Line (Degrees'Image (Measurements.Current_Temperature));
end Main;
```

In the example above, we declare a variable T by renaming the Current_Temperature object from the Measurements package. As you can see by running this example, both Current_Temperature and its alternative name T have the same values:

- first, they show the value 5.0
- after the addition, they show the value 7.5.

This is because they are essentially referring to the same object, but with two different names.

Note that, in the example above, we're using Degrees as an alias of Degree_Celsius. We discussed this method earlier in the course (page 62).

Renaming can be useful for improving the readability of more complicated array indexing. Instead of explicitly using indices every time we're accessing certain positions of the array, we can create shorter names for these positions by renaming them. Let's look at the following example:

```
package Colors is

  type Color is (Black,
                 Red,
                 Green,
                 Blue,
                 White);

  type Color_Array is
               array (Positive range <>) of Color;

  procedure Reverse_It (X : in out Color_Array);

end Colors;
```

```
package body Colors is

  procedure Reverse_It (X : in out Color_Array)
  is
     begin
       for I in X'First ..
         (X'Last + X'First) / 2
      loop
        declare
          Tmp  : Color;
          X_L  : Color
              renames X (I);
          X_R  : Color
              renames X (X'Last + X'First - I);
          begin
```
Listing 22: test_reverse_colors.adb

```
with Ada.Text_Io; use Ada.Text_Io;
with Colors; use Colors;

procedure Test_Reverse_Colors is
    My_Colors : Color_Array (1 .. 5) :=
        (Black, Red, Green, Blue, White);

begin
    for C of My_Colors loop
        Put_Line ("My_Color: "
            & Color'Image (C));
    end loop;
    New_Line;
    Put_Line ("Reversing My_Color..." );
    New_Line;
    Reverse_It (My_Colors);
    for C of My_Colors loop
        Put_Line ("My_Color: "
            & Color'Image (C));
    end loop;
end Test_Reverse_Colors;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Reverse_Colors
MD5: cd9fd7f64d1ec8967e340d57fd7af0a

Runtime output

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK

In the example above, package Colors implements the procedure Reverse_It by declaring new names for two positions of the array. The actual implementation becomes easy to read:
begin
  Tmp := X_Left;
  X_Left := X_Right;
  X_Right := Tmp;
end;

Compare this to the alternative version without renaming:

begin
  Tmp := X (I);
  X (I) := X (X'Last +
           X'First - I);
  X (X'Last + X'First - I) := Tmp;
end;
8.1 Aggregates: A primer

So far, we have talked about aggregates quite a bit and have seen a number of examples. Now we will revisit this feature in some more detail.

An Ada aggregate is, in effect, a literal value for a composite type. It's a very powerful notation that helps you to avoid writing procedural code for the initialization of your data structures in many cases.

A basic rule when writing aggregates is that every component of the array or record has to be specified, even components that have a default value.

This means that the following code is incorrect:

```ada
package Incorrect is
    type Point is record
        X, Y : Integer := 0;
    end record;

    Origin : Point := (X => 0);
end Incorrect;
```

There are a few shortcuts that you can use to make the notation more convenient:

- To specify the default value for a component, you can use the <> notation.
- You can use the | symbol to give several components the same value.
- You can use the others choice to refer to every component that has not yet been specified, provided all those fields have the same type.
- You can use the range notation .. to refer to specify a contiguous sequence of indices in an array.

However, note that as soon as you used a named association, all subsequent components likewise need to be specified with named associations.
### Listing 2: points.ads

```ada
package Points is
  type Point is record
    X, Y : Integer := 0;
  end record;

  type Point_Array is
    array (Positive range <>) of Point;

  -- use the default values
  Origin : Point := (X | Y => <>);

  -- likewise, use the defaults
  Origin_2 : Point := (others => <>);

  Points_1 : Point_Array := ((1, 2), (3, 4));
  Points_2 : Point_Array := (1 => (1, 2),
      2 => (3, 4),
      3 .. 20 => <>);
end Points;
```

### 8.2 Overloading and qualified expressions

Ada has a general concept of name overloading, which we saw earlier in the section on enumeration types (page 51).

Let's take a simple example: it is possible in Ada to have functions that have the same name, but different types for their parameters.

#### Listing 3: pkg.ads

```ada
package Pkg is
  function F (A : Integer) return Integer;
  function F (A : Character) return Integer;
end Pkg;
```

This is a common concept in programming languages, called overloading\(^\text{12}\), or name overloading.

One of the novel aspects of Ada's overloading facility is the ability to resolve overloading based on the return type of a function.

#### Listing 4: pkg.ads

```ada
package Pkg is
  type SSID is new Integer;
```

\(^{12}\) [https://en.wikipedia.org/wiki/Function_overloading](https://en.wikipedia.org/wiki/Function_overloading)
function Convert (Self : SSID)
  return Integer;
function Convert (Self : SSID)
  return String;
end Pkg;

Listing 5: main.adb

with Ada.Text_IO; use Ada.Text_IO;
with Pkg; use Pkg;
procedure Main is
  S : String := Convert (123_145_299);
  -- ^ Valid, will choose the proper Convert
begin
  Put_Line (S);
end Main;

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Overloading
MD5: aa556b55ee89f9c5f8f7e138d84c27b8

Attention: Note that overload resolution based on the type is allowed for both functions and enumeration literals in Ada - which is why you can have multiple enumeration literals with the same name. Semantically, an enumeration literal is treated like a function that has no parameters.

However, sometimes an ambiguity makes it impossible to resolve which declaration of an overloaded name a given occurrence of the name refers to. This is where a qualified expression becomes useful.

Listing 6: pkg.ads

package Pkg is
  type SSID is new Integer;
  function Convert (Self : SSID)
    return Integer;
  function Convert (Self : SSID)
    return String;
  function Convert (Self : Integer)
    return String;
end Pkg;

Listing 7: main.adb

with Ada.Text_IO; use Ada.Text_IO;
with Pkg; use Pkg;
procedure Main is
  S : String := Convert (123_145_299);
  -- ^ Valid, will choose the proper Convert
  S2 : String := Convert (SSID'(123_145_299));

8.2. Overloading and qualified expressions
Learning Ada

-- ^ We specify that the
-- type of the
-- expression is SSID.
-- We could also have declared a temporary
I : SSID := 123_145_299;
S3 : String := Convert (I);
begin
  Put_Line (S);
end Main;

Code block metadata
Project: Courses.Intro_To_Ada.MoreAbout_Types.Overloading_Error
MD5: 722660d8b692cde65a1c2b7800dd78c4

Syntactically the target of a qualified expression can be either any expression in parentheses, or an aggregate:

Listing 8: qual_expr.ads

package Qual_Expr is
type Point is record
  A, B : Integer;
end record;
P : Point := Point'(12, 15);
A : Integer := Integer'(12);
end Qual_Expr;

Code block metadata
Project: Courses.Intro_To_Ada.MoreAbout_Types.Qual_Expr
MD5: e71523eb441a28a4f6549d5f0418620a

This illustrates that qualified expressions are a convenient (and sometimes necessary) way for the programmer to make the type of an expression explicit, for the compiler of course, but also for other programmers.

Attention: While they look and feel similar, type conversions and qualified expressions are not the same.

A qualified expression specifies the exact type that the target expression will be resolved to, whereas a type conversion will try to convert the target and issue a run-time error if the target value cannot be so converted.

Note that you can use a qualified expression to convert from one subtype to another, with an exception raised if a constraint is violated.

X : Integer := Natural'(1);
8.3 Character types

As noted earlier, each enumeration type is distinct and incompatible with every other enumeration type. However, what we did not mention previously is that character literals are permitted as enumeration literals. This means that in addition to the language’s strongly typed character types, user-defined character types are also permitted:

Listing 9: character_example.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Character_Example is
  type My_Char is ('a', 'b', 'c');
  -- Our custom character type, an
  -- enumeration type with 3 valid values.

  C : Character;
  -- ^ Built-in character type
  -- (it's an enumeration type)

  M : My_Char;

begin
  C := '?';;
  -- ^ Character literal
  -- (enumeration literal)

  M := 'a';

  C := 65;
  -- ^ Invalid: 65 is not a
  -- Character value

  C := Character'Val (65);
  -- Assign the character at
  -- position 65 in the
  -- enumeration (which is 'A')

  M := C;
  -- ^ Invalid: C is of type Character,
  -- and M is a My_Char

  M := 'd';
  -- ^ Invalid: 'd' is not a valid
  -- literal for type My_Char
end Character_Example;
```

In this example, we’re using characters in the definition of My_Char.
9.1 Overview

Pointers are a potentially dangerous construct, which conflicts with Ada’s underlying philosophy.

There are two ways in which Ada helps shield programmers from the dangers of pointers:

1. One approach, which we have already seen, is to provide alternative features so that the programmer does not need to use pointers. Parameter modes, arrays, and varying size types are all constructs that can replace typical pointer usages in C.

2. Second, Ada has made pointers as safe and restricted as possible, but allows "escape hatches" when the programmer explicitly requests them and presumably will be exercising such features with appropriate care.

Here is how you declare a simple pointer type, or access type, in Ada:

Listing 1: dates.ads

```ada
package Dates is
  type Months is
    (January, February, March, April,
     May, June, July, August, September,
     October, November, December);

  type Date is record
    Day : Integer range 1 .. 31;
    Month : Months;
    Year : Integer;
  end record;
end Dates;
```

Listing 2: access_types.ads

```ada
with Dates; use Dates;

package Access_Types is
  -- Declare an access type
  type Date_Acc is access Date;
  -- ^ "Designated type"
  -- ^ Date_Acc values
  -- point to Date
  -- objects
  D : Date_Acc := null;
  -- ^ Literal for
  -- "access to nothing"
```

(continues on next page)
Learning Ada

```
end Access_Types;
```

**Code block metadata**

**Project:** Courses.Intro_To_Ada.Access_Types.Access_Types  
**MD5:** d3421918c48c221b36bdf03b9e68bfb5

This illustrates how to:

- Declare an access type whose values point to ("designate") objects from a specific type
- Declare a variable (access value) from this access type
- Give it a value of `null`

In line with Ada's strong typing philosophy, if you declare a second access type whose designated type is `Date`, the two access types will be incompatible with each other:

```
    with Dates; use Dates;

    package Access_Types is
      -- Declare an access type
      type Date_Acc is access Date;
      type Date_Acc_2 is access Date;
      D : Date_Acc := null;
      D2 : Date_Acc_2 := D;
      -- ^ Invalid! Different types
    end Access_Types;
```

**Build output**

```
access_types.ads:9:24: error: expected type "Date_Acc_2" defined at line 6
access_types.ads:9:24: error: found type "Date_Acc" defined at line 5
```

**In other languages**

In most other languages, pointer types are structurally, not nominally typed, like they are in Ada, which means that two pointer types will be the same as long as they share the same target type and accessibility rules.

Not so in Ada, which takes some time getting used to. A seemingly simple problem is, if you want to have a canonical access to a type, where should it be declared? A commonly used pattern is that if you need an access type to a specific type you "own", you will declare it along with the type:

```
package Access_Types is
  type Point is record
    X, Y : Natural;
  end record;
```

(continues on next page)
type Point_Access is access Point;
end Access_Types;

9.2 Allocation (by type)

Once we have declared an access type, we need a way to give variables of the types a meaningful value! You can allocate a value of an access type with the new keyword in Ada.

Listing 4: access_types.ads

with Dates; use Dates;
package Access_Types is
type Date_Acc is access Date;
D : Date_Acc := new Date;
-- ^ Allocate a new Date record
end Access_Types;

Code block metadata
Project: Courses.Intro_To_Ada.Access_Types.Access_Types
MD5: e0be95b966e4ae8aaf25db646d60c35c

If the type you want to allocate needs constraints, you can put them in the subtype indication, just as you would do in a variable declaration:

Listing 5: access_types.ads

with Dates; use Dates;
package Access_Types is
type String_Acc is access String;
-- ^ Access to unconstrained array type
Msg : String_Acc;
-- ^ Default value is null
Buffer : String_Acc :=
new String (1 .. 10);
-- ^ Constraint required
end Access_Types;

Code block metadata
Project: Courses.Intro_To_Ada.Access_Types.Access_Types
MD5: 83cf7a1874ff1b739658508098aa8208

In some cases, though, allocating just by specifying the type is not ideal, so Ada also allows you to initialize along with the allocation. This is done via the qualified expression syntax:

Listing 6: access_types.ads

with Dates; use Dates;
package Access_Types is

(continues on next page)
9.3 Dereferencing

The last important piece of Ada's access type facility is how to get from an access value to the object that is pointed to, that is, how to dereference the pointer. Dereferencing a pointer uses the .all syntax in Ada, but is often not needed — in many cases, the access value will be implicitly dereferenced for you:

Listing 7: access_types.ads

```ada
with Dates; use Dates;

package Access_Types is
  type Date_Acc is access Date;
  type String_Acc is access String;
  D : Date_Acc := new Date'(30, November, 2011);
  Msg : String_Acc := new String'("Hello");
end Access_Types;
```

9.4 Other features

As you might know if you have used pointers in C or C++, we are still missing features that are considered fundamental to the use of pointers, such as:

- Pointer arithmetic (being able to increment or decrement a pointer in order to point to the next or previous object)
- Manual deallocation - what is called free or delete in C. This is a potentially unsafe operation. To keep within the realm of safe Ada, you need to never deallocate manually.

Those features exist in Ada, but are only available through specific standard library APIs.

**Attention:** The guideline in Ada is that most of the time you can avoid manual allocation, and you should.
There are many ways to avoid manual allocation, some of which have been covered (such as parameter modes). The language also provides library abstractions to avoid pointers:

1. One is the use of containers (page 203). Containers help users avoid pointers, because container memory is automatically managed.

2. A container to note in this context is the Indefinite holder\textsuperscript{13}. This container allows you to store a value of an indefinite type such as String.

3. GNATCOLL has a library for smart pointers, called Refcount\textsuperscript{14}. Those pointers’ memory is automatically managed, so that when an allocated object has no more references to it, the memory is automatically deallocated.

### 9.5 Mutually recursive types

The linked list is a common idiom in data structures; in Ada this would be most naturally defined through two types, a record type and an access type, that are mutually dependent. To declare mutually dependent types, you can use an incomplete type declaration:

```
package Simple_List is
  type Node;  -- This is an incomplete type declaration,

  type Node_Acc is access Node;  -- which is completed in the same

  type Node is record
    Content : Natural;
    Prev, Next : Node_Acc;
  end record;

end Simple_List;
```

In this example, the Node and Node_Acc types are mutually dependent.

---

\textsuperscript{13} http://www.ada-auth.org/standards/12rat/html/Rat12-8-5.html  
\textsuperscript{14} https://github.com/AdaCore/gnatcoll-core/blob/master/src/gnatcoll-refcount.ads
10.1 Dynamically sized record types

We have previously seen some simple examples of record types (page 65). Let’s now look at some of the more advanced properties of this fundamental language feature.

One point to note is that object size for a record type does not need to be known at compile time. This is illustrated in the example below:

Listing 1: runtime_length.ads

```adatrue
package Runtime_Length is
  function Compute_Max_Len return Natural;
end Runtime_Length;
```

Listing 2: var_size_record.ads

```adatrue
with Runtime_Length; use Runtime_Length;

package Var_Size_Record is
  Max_Len : constant Natural := Compute_Max_Len;
  -- ^ Not known at compile time

type Items_Array is
  array (Positive range <>) of Integer;

type Growable_Stack is record
  Items : Items_Array (1 .. Max_Len);
  Len : Natural;
end record;
-- Growable_Stack is a definite type, but
-- size is not known at compile time.

G : Growable_Stack;
end Var_Size_Record;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Var_Size_Record
MD5: 6fb0b3f2b685a72ec694640ce378f77c

It is completely fine to determine the size of your records at run time, but note that all objects of this type will have the same size.
10.2 Records with discriminant

In the example above, the size of the Items field is determined once, at run-time, but every Growable_Stack instance will be exactly the same size. But maybe that’s not what you want to do. We saw that arrays in general offer this flexibility: for an unconstrained array type, different objects can have different sizes.

You can get analogous functionality for records, too, using a special kind of field that is called a discriminant:

Listing 3: var_size_record_2.ads

```ada
package Var_Size_Record_2 is
type Items_Array is
  array (Positive range <>) of Integer;
type Growable_Stack (Max_Len : Natural) is
  record
    -- ^ Discriminant. Cannot be modified once initialized.
    Items : Items_Array (1 .. Max_Len);
    Len : Natural := 0;
  end record;
-- Growable_Stack is an indefinite type (like an array)
end Var_Size_Record_2;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Var_Size_Record_2
MD5: 0c2ffe41b7553984e1ef48a50386559f

Discriminants, in their simple forms, are constant: You cannot modify them once you have initialized the object. This intuitively makes sense since they determine the size of the object.

Also, they make a type indefinite: Whether or not the discriminant is used to specify the size of an object, a type with a discriminant will be indefinite if the discriminant is not declared with an initialization:

Listing 4: test_discriminants.ads

```ada
package Test_Discriminants is
type Point (X, Y : Natural) is record
  null;
end record;
P : Point;
-- ERROR: Point is indefinite, so you need to specify the discriminants or give a default value
P2 : Point (1, 2);
P3 : Point := (1, 2);
-- Those two declarations are equivalent.
end Test_Discriminants;
```

Code block metadata
This also means that, in the example above, you cannot declare an array of Point values, because the size of a Point is not known.

As mentioned in the example above, we could provide a default value for the discriminants, so that we could legally declare Point values without specifying the discriminants. For the example above, this is how it would look:

```
package Test_Discriminants is
  type Point (X, Y : Natural := 0) is record
    null;
  end record;

  P : Point;
  -- We can now simply declare a "Point"
  -- without further ado. In this case,
  -- we're using the default values (0)
  -- for X and Y.

  P2 : Point (1, 2);
  P3 : Point := (1, 2);
  -- We can still specify discriminants.
end Test_Discriminants;
```

Also note that, even though the Point type now has default discriminants, we can still specify discriminants, as we're doing in the declarations of P2 and P3.

In most other respects discriminants behave like regular fields: You have to specify their values in aggregates, as seen above, and you can access their values via the dot notation.
11   Put (" " & Integer'Image (G.Items (I)));
12 end loop;
13   Put_Line (">");
14 end Print_Stack;
15
16   S : Growable_Stack :=
17     (Max.Len => 128,
18      Items => (1, 2, 3, 4, others => <>),
19      Len => 4);
20 begin
21     Print_Stack (S);
22 end Main;

Code block metadata
MD5: 4e8c102cd93dc5d8aa1b402589c5239b

Runtime output
<Stack, items: [ 1 2 3 4]>

Note: In the examples above, we used a discriminant to determine the size of an array, but it is not limited to that, and could be used, for example, to determine the size of a nested discriminated record.

10.3 Variant records

The examples of discriminants thus far have illustrated the declaration of records of varying size, by having components whose size depends on the discriminant.

However, discriminants can also be used to obtain the functionality of what are sometimes called "variant records": records that can contain different sets of fields.

Listing 7: variant_record.ads

package Variant_Record is
   -- Forward declaration of Expr
   type Expr;
   -- Access to a Expr
   type Expr_Access is access Expr;
   type Expr_Kind_Type is (Bin.Op.Plus,
                            Bin.Op.Minus,
                            Num);
   -- A regular enumeration type
   type Expr (Kind : Expr_Kind_Type) is record
      -- ^ The discriminant is an enumeration value
      --
      --
                     Left, Right : Expr_Access;
      when Num =>
                     Val : Integer;
      end case;
end Variant_Record;
Learning Ada

(continued from previous page)

```
end case;
-- Variant part. Only one, at the end of
-- the record definition, but can be
-- nested
end record;
end Variant_Record;
```

The fields that are in a `when` branch will be only available when the value of the discriminant is covered by the branch. In the example above, you will only be able to access the fields `Left` and `Right` when the `Kind` is `Bin_Op_Plus` or `Bin_Op_Minus`.

If you try to access a field that is not valid for your record, a `Constraint_Error` will be raised.

```adablock
with Variant_Record; use Variant_Record;

procedure Main is
  E : Expr := (Num, 12);
begin
  E.Left := new Expr'(Num, 15);
  -- Will compile but fail at runtime
end Main;
```

Here is how you could write an evaluator for expressions:

```adablock
with Ada.Text_IO; use Ada.Text_IO;
with Variant_Record; use Variant_Record;

procedure Main is
  function Eval_Expr (E : Expr) return Integer is
    (case E.Kind is
      when Bin_Op_Plus =>
        Eval_Expr (E.Left.all) + Eval_Expr (E.Right.all),
      ...)
```

10.3. Variant records
In other languages

Ada’s variant records are very similar to Sum types in functional languages such as OCaml or Haskell. A major difference is that the discriminant is a separate field in Ada, whereas the ‘tag’ of a Sum type is kind of built in, and only accessible with pattern matching.

There are other differences (you can have several discriminants in a variant record in Ada). Nevertheless, they allow the same kind of type modeling as sum types in functional languages.

Compared to C/C++ unions, Ada variant records are more powerful in what they allow, and are also checked at run time, which makes them safer.
11.1 Decimal fixed-point types

We have already seen how to specify floating-point types. However, in some applications floating-point is not appropriate since, for example, the roundoff error from binary arithmetic may be unacceptable or perhaps the hardware does not support floating-point instructions. Ada provides a category of types, the decimal fixed-point types, that allows the programmer to specify the required decimal precision (number of digits) as well as the scaling factor (a power of ten) and, optionally, a range. In effect the values will be represented as integers implicitly scaled by the specified power of 10. This is useful, for example, for financial applications.

The syntax for a simple decimal fixed-point type is

```ada
type <type-name> is delta <delta-value> digits <digits-value>;
```

In this case, the `delta` and the `digits` will be used by the compiler to derive a range.

Several attributes are useful for dealing with decimal types:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>The first value of the type</td>
</tr>
<tr>
<td>Last</td>
<td>The last value of the type</td>
</tr>
<tr>
<td>Delta</td>
<td>The delta value of the type</td>
</tr>
</tbody>
</table>

In the example below, we declare two data types: T3_D3 and T6_D3. For both types, the delta value is the same: 0.001.

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Decimal_Fixed_Point_Types is
  type T3_D3 is delta 10.0 ** (-3) digits 3;
  type T6_D3 is delta 10.0 ** (-3) digits 6;
begin
  Put_Line ("The delta value of T3_D3 is ",
            T3_D3'Image (T3_D3'Delta));
  Put_Line ("The minimum value of T3_D3 is ",
            T3_D3'Image (T3_D3'First));
  Put_Line ("The maximum value of T3_D3 is ",
            T3_D3'Image (T3_D3'Last));
  New_Line;
  Put_Line ("The delta value of T6_D3 is ",
            T6_D3'Image (T6_D3'Delta));
```

(continues on next page)
17 Put_Line ("The minimum value of T6_D3 is \\
18 & T6_D3'Image (T6_D3'First));
19 Put_Line ("The maximum value of T6_D3 is \\
20 & T6_D3'Image (T6_D3'Last));
end Decimal_Fixed_Point_TYPES;

When running the application, we see that the delta value of both types is indeed the same: 0.001. However, because T3_D3 is restricted to 3 digits, its range is -0.999 to 0.999. For the T6_D3, we have defined a precision of 6 digits, so the range is -999.999 to 999.999.

Similar to the type definition using the range syntax, because we have an implicit range, the compiled code will check that the variables contain values that are not out-of-range. Also, if the result of a multiplication or division on decimal fixed-point types is smaller than the delta value required for the context, the actual result will be zero. For example:

Listing 2: decimal_fixed_point_smaller.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Decimal_Fixed_Point_Smaller is
  type T3_D3 is delta 10.0 ** (-3) digits 3;
  type T6_D6 is delta 10.0 ** (-6) digits 6;
  A : T3_D3 := T3_D3'Delta;
  B : T3_D3 := 0.5;
  C : T6_D6;
begin
  Put_Line ("The value of A is \\
            & T3_D3'Image (A));
  A := A * B;
  Put_Line ("The value of A * B is \\
            & T3_D3'Image (A));
  A := T3_D3'Delta;
  C := A * B;
  Put_Line ("The value of A * B is \\
            & T6_D6'Image (C));
end Decimal_Fixed_Point_Smaller;
```

108 Chapter 11. Fixed-point types
The value of A is 0.001
The value of A * B is 0.000
The value of A * B is 0.000500

In this example, the result of the operation 0.001 * 0.5 is 0.0005. Since this value is not representable for the T3_D3 type because the delta value is 0.001, the actual value stored in variable A is zero. However, accuracy is preserved during the arithmetic operations if the target has sufficient precision, and the value displayed for C is 0.000500.

11.2 Ordinary fixed-point types

Ordinary fixed-point types are similar to decimal fixed-point types in that the values are, in effect, scaled integers. The difference between them is in the scale factor: for a decimal fixed-point type, the scaling, given explicitly by the type's delta, is always a power of ten.

In contrast, for an ordinary fixed-point type, the scaling is defined by the type's small, which is derived from the specified delta and, by default, is a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types.

**Note:** Ordinary fixed-point types can be thought of being closer to the actual representation on the machine, since hardware support for decimal fixed-point arithmetic is not widespread (rescalings by a power of ten), while ordinary fixed-point types make use of the available integer shift instructions.

The syntax for an ordinary fixed-point type is

```ada
type <type-name> is
delta <delta-value>
range <lower-bound> .. <upper-bound>;
```

By default the compiler will choose a scale factor, or small, that is a power of 2 no greater than <delta-value>.

For example, we may define a normalized range between -1.0 and 1.0 as following:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Normalized_Fixed_Point_Type is
  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;
begin
  Put_Line ("TQ31 requires ". Integer'Image (TQ31'Size) & " bits");
  Put_Line ("The delta value of TQ31 is ". TQ31'Delta);
  Put_Line ("The minimum value of TQ31 is ". TQ31'First);
  Put_Line ("The maximum value of TQ31 is ". TQ31'Last);
end Normalized_Fixed_Point_Type;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Normalized_Fixed_Point_Type
MD5: 778dde401c7ff3dd42938dccfe6cf9d3

11.2. Ordinary fixed-point types 109
Learning Ada

Runtime output

TQ31 requires 32 bits
The delta value of TQ31 is 0.0000000005
The minimum value of TQ31 is -1.0000000000
The maximum value of TQ31 is 0.9999999995

In this example, we are defining a 32-bit fixed-point data type for our normalized range. When running the application, we notice that the upper bound is close to one, but not exact one. This is a typical effect of fixed-point data types — you can find more details in this discussion about the Q format\(^{15}\).

We may also rewrite this code with an exact type definition:

Listing 4: normalized_adapted_fixed_point_type.adb

```ada
procedure Normalized_Adapted_Fixed_Point_Type is
type TQ31 is
delta 2.0 ** (-31)
range -1.0 .. 1.0 - 2.0 ** (-31);
begin
null;
end Normalized_Adapted_Fixed_Point_Type;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Normalized_Adapted_Fixed_Point_Type
MD5: 3421800bb47b282d601a51d276944f62

We may also use any other range. For example:

Listing 5: custom_fixed_point_range.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics; use Ada.Numerics;

procedure Custom_Fixed_Point_Range is
type T_Inv_Trig is
delta 2.0 ** (-15) * Pi
range -Pi / 2.0 .. Pi / 2.0;
begin
Put_Line ("T_Inv_Trig requires "
& Integer"Image (T_Inv_Trig"Size)
& " bits");
Put_Line ("Delta value of T_Inv_Trig: "
& T_Inv_Trig"Image
(T_Inv_Trig'Delta));
Put_Line ("Minimum value of T_Inv_Trig: "
& T_Inv_Trig"Image
(T_Inv_Trig'First));
Put_Line ("Maximum value of T_Inv_Trig: "
& T_Inv_Trig"Image
(T_Inv_Trig'Last));
end Custom_Fixed_Point_Range;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Custom_Fixed_Point_Range
MD5: a3e6c549cb1070aa285857ae8813de27

Runtime output

\(^{15}\) https://en.wikipedia.org/wiki/Q_(number_format)
In this example, we are defining a 16-bit type called T_Inv_Trig, which has a range from -π/2 to π/2.

All standard operations are available for fixed-point types. For example:

Listing 6: fixed_point_op.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Fixed_Point_Op is
  type TQ31 is
    delta 2.0 ** (-31)
    range -1.0 .. 1.0 - 2.0 ** (-31);

  A, B, R : TQ31;

begin
  A := 0.25;
  B := 0.50;
  R := A + B;
  Put_Line ("R is " & TQ31'Image (R));
end Fixed_Point_Op;
```

As expected, R contains 0.75 after the addition of A and B.

In fact the language is more general than these examples imply, since in practice it is typical to need to multiply or divide values from different fixed-point types, and obtain a result that may be of a third fixed-point type. The details are outside the scope of this introductory course.

It is also worth noting, although again the details are outside the scope of this course, that you can explicitly specify a value for an ordinary fixed-point type's small. This allows non-binary scaling, for example:

```ada
type Angle is
  delta 1.0/3600.0
  range 0.0 .. 360.0 - 1.0 / 3600.0;

for Angle'Small use Angle'Delta;
```
One of the main principles of modular programming, as well as object oriented programming, is encapsulation\(^{16}\).

Encapsulation, briefly, is the concept that the implementer of a piece of software will distinguish between the code's public interface and its private implementation.

This is not only applicable to software libraries but wherever abstraction is used.

In Ada, the granularity of encapsulation is a bit different from most object-oriented languages, because privacy is generally specified at the package level.

### 12.1 Basic encapsulation

Listing 1: encapsulate.ads

```ada
package Encapsulate is
    procedure Hello;

private

    procedure Hello2;
    -- Not visible from external units
end Encapsulate;
```

Listing 2: encapsulate.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Encapsulate is

    procedure Hello is
    begin
        Put_Line ("Hello");
    end Hello;

    procedure Hello2 is
    begin
        Put_Line ("Hello #2");
    end Hello2;
end Encapsulate;
```

\(^{16}\) https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)
Learning Ada

Listing 3: main.adb

```ada
with Encapsulate;

procedure Main is
begin
    Encapsulate.Hello;
    Encapsulate.Hello2;
    -- Invalid: Hello2 is not visible
end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Encapsulate
MD5: cf56ee89481962d1e0a6d1e9ad888362

Build output

main.adb:6:15: error: "Hello2" is not a visible entity of "Encapsulate"
gprbuild: *** compilation phase failed

12.2 Abstract data types

With this high-level granularity, it might not seem obvious how to hide the implementation details of a type. Here is how it can be done in Ada:

Listing 4: stacks.ads

```ada
package Stacks is
type Stack is private;
    -- Declare a private type: You cannot depend
    -- on its implementation. You can only assign
    -- and test for equality.
    subtype Stack_Index is
        Natural range 1 .. 10;

    type Content_Type is
        array (Stack_Index) of Natural;

    type Stack is record
        Top : Stack_Index;
        Content : Content_Type;
    end record;
end Stacks;
```

Listing 5: stacks.adb

```ada
package body Stacks is
    procedure Push (S : in out Stack;
    Val : Integer) is
```
begin
   -- Missing implementation!
   null;
end Push;

procedure Pop (S : in out Stack;
   Val :   out Integer) is
begin
   -- Dummy implementation!
   Val := 0;
end Pop;
end Stacks;

Code block metadata

MD5: 364df7c6806af4a1bc957c2c2d53b2cc

In the above example, we define a stack type in the public part (known as the visible part of the package spec in Ada), but the exact representation of that type is private.

Then, in the private part, we define the representation of that type. We can also declare other types that will be used as helpers for our main public type. This is useful since declaring helper types is common in Ada.

A few words about terminology:

• The Stack type as viewed from the public part is called the partial view of the type. This is what clients have access to.

• The Stack type as viewed from the private part or the body of the package is called the full view of the type. This is what implementers have access to.

From the point of view of the client (the with'ing unit), only the public (visible) part is important, and the private part could as well not exist. It makes it very easy to read linearly the part of the package that is important for you.

-- No need to read the private part to use the package
package Stacks is
   type Stack is private;
   procedure Push (S : in out Stack;
      Val :   Integer);
   procedure Pop (S : in out Stack;
      Val :   out Integer);
private
   ...
end Stacks;

Here is how the Stacks package would be used:

-- Example of use
with Stacks; use Stacks;

procedure Test_Stack is
   S : Stack;
   Res : Integer;
begn
   Push (S, 5);
   Push (S, 7);
   Pop (S, Res);
end Test_Stack;
## 12.3 Limited types

Ada's *limited type* facility allows you to declare a type for which assignment and comparison operations are not automatically provided.

**Listing 6: stacks.ads**

```ada
package Stacks is
    type Stack is limited private;
    -- Limited type. Cannot assign nor compare.

    procedure Push (S : in out Stack;
        Val : Integer);
    procedure Pop (S : in out Stack;
        Val : Integer);

private
    subtype Stack_Index is
        Natural range 1 .. 10;

    type Content_Type is
        array (Stack_Index) of Natural;

    type Stack is limited record
        Top : Stack_Index;
        Content : Content_Type;
    end record;
end Stacks;
```

**Listing 7: stacks.adb**

```ada
package body Stacks is

    procedure Push (S : in out Stack;
        Val : Integer) is
    begin
        -- Missing implementation!
        null;
    end Push;

    procedure Pop (S : in out Stack;
        Val : out Integer) is
    begin
        -- Dummy implementation!
        Val := 0;
    end Pop;
end Stacks;
```
Listing 8: main.adb

```ada
with Stacks; use Stacks;

procedure Main is
  S, S2 : Stack;
begin
  S := S2;
  -- Illegal: S is limited.
end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Limited_Stacks
MD5: 811343b46f20ac6af5e1bf26561f8d8d

Build output

main.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

This is useful because, for example, for some data types the built-in assignment operation might be incorrect (for example when a deep copy is required).

Ada does allow you to overload the comparison operators = and /= for limited types (and to override the built-in declarations for non-limited types).

Ada also allows you to implement special semantics for assignment via controlled types\(^\text{17}\). However, in some cases assignment is simply inappropriate; one example is the File_Type from the Ada.Text_IO package, which is declared as a limited type and thus attempts to assign one file to another would be detected as illegal.

### 12.4 Child packages & privacy

We've seen previously (in the child packages section (page 39)) that packages can have child packages. Privacy plays an important role in child packages. This section discusses some of the privacy rules that apply to child packages.

Although the private part of a package P is meant to encapsulate information, certain parts of a child package P.C can have access to this private part of P. In those cases, information from the private part of P can then be used as if it were declared in the public part of its specification. To be more specific, the body of P.C and the private part of the specification of P.C have access to the private part of P. However, the public part of the specification of P.C only has access to the public part of P's specification. The following table summarizes this:

<table>
<thead>
<tr>
<th>Part of a child package</th>
<th>Access to the private part of its parent's specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification: public part</td>
<td></td>
</tr>
<tr>
<td>Specification: private part</td>
<td>✓</td>
</tr>
<tr>
<td>Body</td>
<td>✓</td>
</tr>
</tbody>
</table>

The rest of this section shows examples of how this access to private information actually works for child packages.

Let's first look at an example where the body of a child package P.C has access to the private part of the specification of its parent P. We've seen, in a previous source-code example,\(^\text{17}\) http://www.ada-auth.org/standards/12rm/html/RM-7-6.html

\(^{17}\) http://www.ada-auth.org/standards/12rm/html/RM-7-6.html
that the Hello2 procedure declared in the private part of the Encapsulate package cannot
be used in the Main procedure, since it's not visible there. This limitation doesn't apply,
however, for parts of the child packages of the Encapsulate package. In fact, the body of
its child package Encapsulate.Child has access to the Hello2 procedure and can call it
there, as you can see in the implementation of the Hello3 procedure of the Child package:

Listing 9: encapsulate.ads

```
package Encapsulate is
  procedure Hello;

private

  procedure Hello2;
  -- Not visible from external units
  -- But visible in child packages
end Encapsulate;
```

Listing 10: encapsulate.adb

```
with Ada.Text_IO; use Ada.Text_IO;

package body Encapsulate is
  procedure Hello is
    begin
      Put_Line ("Hello");
    end Hello;

  procedure Hello2 is
    begin
      Put_Line ("Hello #2");
    end Hello2;

end Encapsulate;
```

Listing 11: encapsulate-child.ads

```
package Encapsulate.Child is

  procedure Hello3;

end Encapsulate.Child;
```

Listing 12: encapsulate-child.adb

```
with Ada.Text_IO; use Ada.Text_IO;

package body Encapsulate.Child is
  procedure Hello3 is
    -- Using private procedure Hello2
    -- from the parent package
    Hello2;
    Put_Line ("Hello #3");
  end Hello3;

end Encapsulate.Child;
```
Listing 13: main.adb

```ada
with Encapsulate.Child;

procedure Main is
begin
    Encapsulate.Child.Hello3;
end Main;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Privacy.Encapsulate_Child
MD5: 1533f43ee8f8b4d14c9b2101f42f13a

**Runtime output**

Hello #2
Hello #3

The same mechanism applies to types declared in the private part of a parent package. For instance, the body of a child package can access components of a record declared in the private part of its parent package. Let's look at an example:

Listing 14: my_types.ads

```ada
package My_Types is
    type Priv_Rec is private;
private
    type Priv_Rec is record
        Number : Integer := 42;
    end record;
end My_Types;
```

Listing 15: my_types-ops.ads

```ada
package My_Types.Ops is
    procedure Display (E : Priv_Rec);
end My_Types.Ops;
```

Listing 16: my_types-ops.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body My_Types.Ops is
    procedure Display (E : Priv_Rec) is
    begin
        Put_Line ("Priv_Rec.Number: ", Integer'Image (E.Number));
    end Display;
end My_Types.Ops;
```
Learning Ada

Listing 17: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with My_Types; use My_Types;
with My_Types.Ops; use My_Types.Ops;

procedure Main is
    E : Priv_Rec;
begin
    -- The following code would trigger a
    -- compilation error here:
    -- Put_Line ("Priv_Rec.Number: "
    -- & Integer'Image (E.Number));
    Display (E);
end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Private_Type_Child
MD5: 9960611460bc1190b30949eca08fc02b

Runtime output

Presenting information:
Priv_Rec.Number: 42

In this example, we don't have access to the Number component of the record type Priv_Rec in the Main procedure. You can see this in the call to Put_Line that has been commented-out in the implementation of Main. Trying to access the Number component there would trigger a compilation error. But we do have access to this component in the body of the My_Types.Ops package, since it's a child package of the My_Types package. Therefore, Ops's body has access to the declaration of the Priv_Rec type — which is in the private part of its parent, the My_Types package. For this reason, the same call to Put_Line that would trigger a compilation error in the Main procedure works fine in the Display procedure of the My_Types.Ops package.

This kind of privacy rules for child packages allows for extending the functionality of a parent package and, at the same time, retain its encapsulation.

As we mentioned previously, in addition to the package body, the private part of the specification of a child package P.C also has access to the private part of the specification of its parent P. Let's look at an example where we declare an object of private type Priv_Rec in the private part of the child package My_Types.Child and initialize the Number component of the Priv_Rec record directly:

```ada
package My_Types.Child is
    private
        E : Priv_Rec := (Number => 99);
    end My_Types.Ops;
```

As expected, we wouldn't be able to initialize this component if we moved this declaration to the public (visible) part of the same child package:

Chapter 12. Privacy
package My_Types.Child is
    E : Priv_Rec := (Number => 99);
end My_Types.Ops;

The declaration above triggers a compilation error, since type Priv_Rec is private. Because the public part of My_Types.Child is also visible outside the child package, Ada cannot allow accessing private information in this part of the specification.
13.1 Introduction

Generics are used for metaprogramming in Ada. They are useful for abstract algorithms that share common properties with each other.

Either a subprogram or a package can be generic. A generic is declared by using the keyword `generic`. For example:

Listing 1: operator.ads

```ada
generic
  type T is private;
-- Declaration of formal types and objects
-- Below, we could use one of the following:
-- <procedure | function | package>
procedure Operator (Dummy : in out T);
```

Listing 2: operator.adb

```ada
procedure Operator (Dummy : in out T) is
begin
  null;
end Operator;
```

13.2 Formal type declaration

Formal types are abstractions of a specific type. For example, we may want to create an algorithm that works on any integer type, or even on any type at all, whether a numeric type or not. The following example declares a formal type `T` for the Set procedure.

Listing 3: set.ads

```ada
generic
  type T is private;
-- T is a formal type that indicates that
-- any type can be used, possibly a numeric
-- type or possibly even a record type.
procedure Set (Dummy : T);
```
Listing 4: set.adb

```ada
procedure Set (Dummy : T) is
begin
  null;
end Set;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Generics.Show_Formal_Type_Declaration
MD5: 668156f66b2479c4932d18b5ad35deba

The declaration of `T` as `private` indicates that you can map any definite type to it. But you can also restrict the declaration to allow only some types to be mapped to that formal type. Here are some examples:

<table>
<thead>
<tr>
<th>Formal Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any type</td>
<td><code>type T is private;</code></td>
</tr>
<tr>
<td>Any discrete type</td>
<td><code>type T is (&lt;&gt;);</code></td>
</tr>
<tr>
<td>Any floating-point type</td>
<td><code>type T is digits &lt;&gt;;</code></td>
</tr>
</tbody>
</table>

### 13.3 Formal object declaration

Formal objects are similar to subprogram parameters. They can reference formal types declared in the formal specification. For example:

Listing 5: set.ads

```ada
generic
  type T is private;
  X : in out T;
--  X can be used in the Set procedure
procedure Set (E : T);
```
13.4 Generic body definition

We don’t repeat the `generic` keyword for the body declaration of a generic subprogram or package. Instead, we start with the actual declaration and use the generic types and objects we declared. For example:

```
13.4. Generic body definition

13.4.1.135
```

13.5 Generic instantiation

Generic subprograms or packages can’t be used directly. Instead, they need to be instantiated, which we do using the `new` keyword, as shown in the following example:

```
13.5.135
```
Learning Ada

Listing 10: set.adb

```ada
procedure Set (E : T) is
begin
  X := E;
end Set;
```

Listing 11: show_generic_instantiation.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Set;

procedure Show_Generic_Instantiation is
  Main   : Integer := 0;
  Current: Integer;

  procedure Set_Main is new Set (T => Integer,
                                X => Main);

begin
  Current := 10;
  Set_Main (Current);
  Put_Line ("Value of Main is "
            & Integer'Image (Main));
end Show_Generic_Instantiation;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Generic_Instantiation
MD5: 13dc0692252496d954240952561e1c05

Runtime output

Value of Main is 10

In the example above, we instantiate the procedure Set by mapping the formal parameters T and X to actual existing elements, in this case the `Integer` type and the `Main` variable.

### 13.6 Generic packages

The previous examples focused on generic subprograms. In this section, we look at generic packages. The syntax is similar to that used for generic subprograms: we start with the `generic` keyword and continue with formal declarations. The only difference is that `package` is specified instead of a subprogram keyword.

Here's an example:
Listing 12: element.ads

generic
  type T is private;
package Element is
  procedure Set (E : T);
  procedure Reset;
  function Get return T;
  function Is_Valid return Boolean;
  Invalid_Element : exception;
private
  Value : T;
  Valid : Boolean := False;
end Element;

Listing 13: element.adb

package body Element is
  procedure Set (E : T) is
    begin
      Value := E;
      Valid := True;
    end Set;
  procedure Reset is
    begin
      Valid := False;
    end Reset;
  function Get return T is
    begin
      if not Valid then
        raise Invalid_Element;
      end if;
      return Value;
    end Get;
  function Is_Valid return Boolean is (Valid);
end Element;

Listing 14: show_generic_package.adb

with Ada.Text_IO; use Ada.Text_IO;
with Element;

procedure Show_Generic_Package is
  package I is new Element (T => Integer);
  procedure Display_Initialed is
    begin
      if I.Is_Valid then
        Put_Line ("Value is initialized");
      else
        Put_Line ("Value is not initialized");
      end if;
    end Display_Initialed;

(continues on next page)
begin
  Display_Initialized;

  Put_Line ("Initializing...");
  I.Set (5);
  Display_Initialized;
  Put_Line ("Value is now set to "
            & Integer'Image (I.Get));

  Put_Line ("Resetting...");
  I.Reset;
  Display_Initialized;
end Show_Generic_Package;

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Generic_Package
MD5: c5278a06c6d06f1f37353ee0ca6686ec

Runtime output

Value is not initialized
Initializing...
Value is initialized
Value is now set to 5
Resetting...
Value is not initialized

In the example above, we created a simple container named Element, with just one single element. This container tracks whether the element has been initialized or not.

After writing the package definition, we create the instance I of the Element. We use the instance by calling the package subprograms (Set, Reset, and Get).

13.7 Formal subprograms

In addition to formal types and objects, we can also declare formal subprograms or packages. This course only describes formal subprograms; formal packages are discussed in the advanced course.

We use the with keyword to declare a formal subprogram. In the example below, we declare a formal function (Comparison) to be used by the generic procedure Check.

Listing 15: check.ads

```ada
generic
  Description : String;
  type T is private;
  with function Comparison (X, Y : T)
       return Boolean;
procedure Check (X, Y : T);
```

Listing 16: check.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)
procedure Check (X, Y : T) is
  Result : Boolean;
begin
  Result := Comparison (X, Y);
  if Result then
    Put_Line
      ("Comparison (" & Description & ") between arguments is OK!");
  else
    Put_Line
      ("Comparison (" & Description & ") between arguments is not OK!");
  end if;
end Check;

with Check;

procedure Show_Formal_Subprogram is
  A, B : Integer;

  procedure Check_Is_Equal is new
    Check (Description => "equality",
      T => Integer,
      Comparison => Standard."="); -- Here, we are mapping the standard
  -- equality operator for Integer types to
  -- the Comparison formal function
begin
  A := 0;
  B := 1;
  Check_Is_Equal (A, B);
end Show_Formal_Subprogram;

13.8 Example: I/O instances

Ada offers generic I/O packages that can be instantiated for standard and derived types. One example is the generic Float_IO package, which provides procedures such as Put and Get. In fact, Float_Text_IO — available from the standard library — is an instance of the Float_IO package, and it's defined as:

with Ada.Text_IO;

package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO (Float);
Learning Ada

You can use it directly with any object of floating-point type. For example:

Listing 18: show_float_text_io.adb

```ada
with Ada.Float_Text_IO;

procedure Show_Float_Text_IO is
  X : constant Float := 2.5;
  use Ada.Float_Text_IO;
begin
  Put (X);
end Show_Float_Text_IO;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Float_Text_IO
MD5: 7cc9b547ef301a2071e9fb65caa4631b

Runtime output

2.50000E+00

Instantiating generic I/O packages can be useful for derived types. For example, let's create a new type Price that must be displayed with two decimal digits after the point, and no exponent.

Listing 19: show_float_io_inst.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Float_IO_Inst is
  type Price is digits 3;
  package Price_IO is new Ada.Text_IO.Float_IO (Price);
  P : Price;
begin
  -- Set to zero => don't display exponent
  Price_IO.Default_Exp := 0;
  P := 2.5;
  Price_IO.Put (P);
  New_Line;
  P := 5.75;
  Price_IO.Put (P);
  New_Line;
end Show_Float_IO_Inst;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Float_IO_Inst
MD5: 583c761421d7fdb812dd2a183b676bae

Runtime output

2.50
5.75

By adjusting Default_Exp from the Price_IO instance to remove the exponent, we can
Learning Ada

control how variables of Price type are displayed. Just as a side note, we could also have written:

```ada
-- [...] 

  type Price is new Float;

  package Price_IO is new
    Ada.Text_IO.Float_IO (Price);
  begin
    Price_IO.Default_Aft := 2;
    Price_IO.Default_Exp := 0;
```

In this case, we’re adjusting Default_Aft, too, to get two decimal digits after the point when calling Put.

In addition to the generic Float_IO package, the following generic packages are available from Ada.Text_IO:

- Enumeration_IO for enumeration types;
- Integer_IO for integer types;
- Modular_IO for modular types;
- Fixed_IO for fixed-point types;
- Decimal_IO for decimal types.

In fact, we could rewrite the example above using decimal types:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Decimal_IO_Inst is

  type Price is delta 10.0 ** (-2) digits 12;

  package Price_IO is new
    Ada.Text_IO.Decimal_IO (Price);

  P : Price;

begin
  Price_IO.Default.Exp := 0;
  P := 2.5;
  Price_IO.Put (P);
  New_Line;
  P := 5.75;
  Price_IO.Put (P);
  New_Line;
end Show_Decimal_IO_Inst;
```

Code block metadata

Project: Courses.Intro.To_Ada.Generics.Show_Decimal_IO.Inst
MD5: f413570759dcb32cc166078b3cee1a16

Runtime output

```
  2.50
  5.75
```

13.8. Example: I/O instances
13.9 Example: ADTs

An important application of generics is to model abstract data types (ADTs). In fact, Ada includes a library with numerous ADTs using generics: Ada.Containers (described in the containers section (page 203)).

A typical example of an ADT is a stack:

Listing 21: stacks.ads

generic
Max : Positive;
type T is private;
package Stacks is

type Stack is limited private;
Stack_Underflow, Stack_Overflow : exception;
function Is_Empty (S : Stack) return Boolean;
function Pop (S : in out Stack) return T;
procedure Push (S : in out Stack;
V : T);

private

type Stack_Array is
array (Natural range <>) of T;
Min : constant := 1;
type Stack is record
Container : Stack_Array (Min .. Max);
Top : Natural := Min - 1;
end record;
end Stacks;

Listing 22: stacks.adb

package body Stacks is

function Is_Empty (S : Stack) return Boolean is
(S.Top < S.Container'First);
function Is_Full (S : Stack) return Boolean is
(S.Top >= S.Container'Last);
function Pop (S : in out Stack) return T is
begin
if Is_Empty (S) then
raise Stack_Underflow;
else
return X : T do
X := S.Container (S.Top);
S.Top := S.Top - 1;
end return;
end if;
end Pop;

(continues on next page)
procedure Push (S : in out Stack; V : T) is
begin
  if Is_Full (S) then
    raise Stack_Overflow;
  else
    S.Top := S.Top + 1;
    S.Container (S.Top) := V;
  end if;
end Push;
end Stacks;

Listing 23: show_stack.adb

with Ada.Text_IO; use Ada.Text_IO;
with Stacks;
procedure Show_Stack is
  package Integer_Stacks is new Stacks (Max => 10, T => Integer);
  use Integer_Stacks;
  Values : Integer_Stacks.Stack;
begin
  Push (Values, 10);
  Push (Values, 20);
  Put_Line ("Last value was " & Integer'Image (Pop (Values)));
end Show_Stack;

Code block metadata
Project: Courses.Intro_To_Ada.Generics.Show_Stack
MD5: ee112d395552c1a02d2211b9e5425dc71

Runtime output
Last value was 20

In this example, we first create a generic stack package (Stacks) and then instantiate it to create a stack of up to 10 integer values.

13.10 Example: Swap

Let's look at a simple procedure that swaps variables of type Color:

Listing 24: colors.ads

package Colors is
type Color is (Black, Red, Green,
  Blue, White);

(continues on next page)
procedure Swap_Colors (X, Y : in out Color);
end Colors;

Listing 25: colors.adb
package body Colors is
    procedure Swap_Colors (X, Y : in out Color) is
        Tmp : constant Color := X;
    begin
        X := Y;
        Y := Tmp;
    end Swap_Colors;
end Colors;

Listing 26: test_non_generic_swap_colors.adb
with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Test_Non_Generic_Swap_Colors is
    A, B, C : Color;
    begin
        A := Blue;
        B := White;
        C := Red;
        Put_Line ("Value of A is " & Color'Image (A));
        Put_Line ("Value of B is " & Color'Image (B));
        Put_Line ("Value of C is " & Color'Image (C));
        New_Line;
        Put_Line ("Swapping A and C...");
        New_Line;
        Swap_Colors (A, C);
        Put_Line ("Value of A is " & Color'Image (A));
        Put_Line ("Value of B is " & Color'Image (B));
        Put_Line ("Value of C is " & Color'Image (C));
    end Test_Non_Generic_Swap_Colors;

Code block metadata
Project: Courses.Intro_To_Ada.Generics.Test_Non_Generic_Swap_Colors
MD5: 4d1cf826a1676c3750a8aabd484ac71f

Runtime output
Value of A is BLUE
Value of B is WHITE
Value of C is RED
Swapping A and C...
Value of A is RED
Value of B is WHITE
Value of C is BLUE

In this example, Swap_Colors can only be used for the Color type. However, this algorithm can theoretically be used for any type, whether an enumeration type or a complex record type with many elements. The algorithm itself is the same: it’s only the type that differs. If, for example, we want to swap variables of Integer type, we don’t want to duplicate the implementation. Therefore, such an algorithm is a perfect candidate for abstraction using generics.

In the example below, we create a generic version of Swap_Colors and name it Generic_Swap. This generic version can operate on any type due to the declaration of formal type T.

Listing 27: generic_swap.ads

```
generic
  type T is private;
procedure Generic_Swap (X, Y : in out T);
```

Listing 28: generic_swap.adb

```
procedure Generic_Swap (X, Y : in out T) is
  Tmp : constant T := X;
begin
  X := Y;
  Y := Tmp;
end Generic_Swap;
```

Listing 29: colors.ads

```
with Generic_Swap;

package Colors is
  type Color is (Black, Red, Green, Blue, White);
  procedure Swap_Colors is new
    Generic_Swap (T => Color);
end Colors;
```

Listing 30: test_swap_colors.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Test_Swap_Colors is
  A, B, C : Color;
begin
  A := Blue;
  B := White;
  C := Red;
  Put_Line ("Value of A is " & Color'Image (A));
  Put_Line ("Value of B is "
```
& Color’Image (B));
Put_Line (“Value of C is “
& Color’Image (C));
New_Line;
Put_Line (“Swapping A and C...”);
New_Line;
Swap_Colors (A, C);
Put_Line (“Value of A is “
& Color’Image (A));
Put_Line (“Value of B is “
& Color’Image (B));
Put_Line (“Value of C is “
& Color’Image (C));
end Test_Swap_Colors;

As we can see in the example, we can create the same Swap_Colors procedure as we had in the non-generic version of the algorithm by declaring it as an instance of the generic Generic_Swap procedure. We specify that the generic T type will be mapped to the Color type by passing it as an argument to the Generic_Swap instantiation.

13.11 Example: Reversing

The previous example, with an algorithm to swap two values, is one of the simplest examples of using generics. Next we study an algorithm for reversing elements of an array. First, let’s start with a non-generic version of the algorithm, one that works specifically for the Color type:

Listing 31: colors.ads

package Colors is
  type Color is (Black, Red, Green,
               Blue, White);
  type Color_Array is
    array (Integer range <>) of Color;
  procedure Reverse_It (X : in out Color_Array);
end Colors;
end Colors;

Listing 32: colors.adb

package body Colors is

procedure Reverse_It (X : in out Color_Array) is
begin
for I in X'First .. (X'Last + X'First) / 2 loop
declare
Tmp : Color;
X_Left : Color renames X (I);
X_Right : Color renames X (X'Last + X'First - I);
begnin
Tmp := X_Left;
X_Left := X_Right;
X_Right := Tmp;
end;
end loop;
end Reverse_It;

end Colors;

Listing 33: test_non_generic_reverse_colors.adb

with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Test_Non_Generic Reverse Colors is
My Colors : Color_Array (1 .. 5) :=
(Black, Red, Green, Blue, White);

begin
for C of My Colors loop
Put_Line ("My Color: " & Color'Image (C));
end loop;
New_Line;
Put_Line ("Reversing My Color...");
New_Line;
Reverse_It (My_Colors);
for C of My Colors loop
Put_Line ("My Color: " & Color'Image (C));
end loop;
end Test_Non_Generic Reverse Colors;

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Test_Non_Generic Reverse_Colors
MD5: 9b3a489d0bc8ecd79de6ba99fd7cd44f

Runtime output

13.11. Example: Reversing
The procedure `Reverse_It` takes an array of colors, starts by swapping the first and last elements of the array, and continues doing that with successive elements until it reaches the middle of array. At that point, the entire array has been reversed, as we see from the output of the test program.

To abstract this procedure, we declare formal types for three components of the algorithm:

- the elements of the array (Color type in the example)
- the range used for the array (`Integer` range in the example)
- the actual array type (Color_Array type in the example)

This is a generic version of the algorithm:

```ada
procedure Generic_Reverse (X : in out Array_T);
end Generic_Reverse;
```
type Color is (Black, Red, Green, Blue, White);

type Color_Array is array (Integer range <>) of Color;

procedure Reverse_It is new
  Generic_Reverse (T => Color,
                    Index => Integer,
                    Array_T => Color_Array);
end Colors;

Listing 37: test_reverse_colors.adb

with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Test.Reverse_Colors is
  My_Colors : Color_Array (1 .. 5) :=
    (Black, Red, Green, Blue, White);

begin
  for C of My_Colors loop
    Put_Line ("My_Color: "
              & Color'Image (C));
  end loop;

  New_Line;
  Put_Line ("Reversing My_Color...");
  New_Line;
  Reverse_It (My_Colors);

  for C of My_Colors loop
    Put_Line ("My_Color: "
              & Color'Image (C));
  end loop;
end Test.Reverse_Colors;

Code block metadata

MD5: 9ef175c517d7574b4b65b24ba0027f1f

Runtime output

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED

(continues on next page)
As mentioned above, we're abstracting three components of the algorithm:

- the T type abstracts the elements of the array
- the Index type abstracts the range used for the array
- the Array_T type abstracts the array type and uses the formal declarations of the T and Index types.

### 13.12 Example: Test application

In the previous example we've focused only on abstracting the reversing algorithm itself. However, we could have decided to also abstract our small test application. This could be useful if we, for example, decide to test other procedures that change elements of an array.

In order to do this, we again have to choose the elements to abstract. We therefore declare the following formal parameters:

- S: the string containing the array name
- a function Image that converts an element of type T to a string
- a procedure Test that performs some operation on the array

Note that Image and Test are examples of formal subprograms and S is an example of a formal object.

Here is a version of the test application making use of the generic Perform_Test procedure:

```ada
Listing 38: generic_reverse.ads

generic
  type T is private;
  type Index is range <>;
  type Array_T is
    array (Index range <>) of T;
procedure Generic_Reverse (X : in out Array_T);

Listing 39: generic_reverse.adb

procedure Generic_Reverse (X : in out Array_T) is
begin
  for I in X'First .. (X'Last + X'First) / 2 loop
    declare
      Tmp : T;
      X_Left : T
        renames X (I);
      X_Right : T
        renames X (X'Last + X'First - I);
    begin
      Tmp := X_Left;
      X_Left := X_Right;
      X_Right := Tmp;
    end;
  end loop;
end Generic_Reverse;
```
Learning Ada

### Listing 40: `perform_test.ads`
```
generic
    type T is private;
    type Index is range <;
    type Array_T is
        array (Index range <>) of T;
    S : String;
    with function Image (E : T)
        return String is <>;
    with procedure Test (X : in out Array_T);
procedure Perform_Test (X : in out Array_T);
```

### Listing 41: `perform_test.adb`
```
with Ada.Text_IO; use Ada.Text_IO;
procedure Perform_Test (X : in out Array_T) is
begin
    for C of X loop
        Put_Line (S & ": " & Image (C));
    end loop;
    New_Line;
    Put_Line ("Testing " & S & "..." );
    New_Line;
    Test (X);
    for C of X loop
        Put_Line (S & ": " & Image (C));
    end loop;
end Perform_Test;
```

### Listing 42: `colors.ads`
```
with Generic.Reverse;
package Colors is
    type Color is (Black, Red, Green, Blue, White);
    type Color_Array is
        array (Integer range <>) of Color;
    procedure Reverse_It is new
        Generic.Reverse (T => Color, Index => Integer,
            Array_T => Color_Array);
end Colors;
```

### Listing 43: `test_reverse_colors.adb`
```
with Colors; use Colors;
with Perform_Test;
procedure Test_Reverse_Colors is
    procedure Perform_Test_Reverse_It is new
        (continues on next page)
```

13.12. Example: Test application
In this example, we create the procedure `Perform_Test_Reverse_It` as an instance of the generic procedure (`Perform_Test`). Note that:

- For the formal `Image` function, we use the `Image` attribute of the `Color` type
- For the formal `Test` procedure, we reference the `Reverse_Array` procedure from the package.
Ada uses exceptions for error handling. Unlike many other languages, Ada speaks about *raising*, not *throwing*, an exception and *handling*, not *catching*, an exception.

### 14.1 Exception declaration

Ada exceptions are not types, but instead objects, which may be peculiar to you if you're used to the way Java or Python support exceptions. Here's how you declare an exception:

```
package Exceptions is
  My_Except : exception;
  -- Like an object. *NOT* a type!
end Exceptions;
```

**Listing 1: exceptions.ads**

Even though they're objects, you're going to use each declared exception object as a "kind" or "family" of exceptions. Ada does not require that a subprogram declare every exception it can potentially raise.

### 14.2 Raising an exception

To raise an exception of our newly declared exception kind, do the following:

```
with Exceptions; use Exceptions;

procedure Main is
begin
  raise My_Except;
  -- Execution of current control flow
  -- abandoned; an exception of kind
  -- "My_Except" will bubble up until it
  -- is caught.
end Main;
```

**Listing 2: main.adb**
14.3 Handling an exception

Next, we address how to handle exceptions that were raised by us or libraries that we call. The neat thing in Ada is that you can add an exception handler to any statement block as follows:

Listing 4: open_file.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Open_File is
  File : File_Type;
begin
  -- Block (sequence of statements)
  begin
    Open (File, In_File, "input.txt");
    exception
      when E : Name_Error =>
        -- ^ Exception to be handled
        Put ("Cannot open input file : ");
        Put_Line (Exception_Message (E));
        raise;
        -- Reraise current occurrence
```

(continues on next page)
17 end;
18 end Open_File;

Code block metadata
Project: Courses.Intro_To_Ada.Exceptions.Show_Exception_Handling
MD5: 4ea1d5da684a6d7d7ee32980810e9c8f

Runtime output
Cannot open input file: input.txt: No such file or directory
raised ADA.IO_EXCEPTIONS.NAME_ERROR: input.txt: No such file or directory

In the example above, we're using the Exception_Message function from the Ada.Exceptions package. This function returns the message associated with the exception as a string.

You don't need to introduce a block just to handle an exception: you can add it to the statements block of your current subprogram:

Listing 5: open_file.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Open_File is
  File : File_Type;
begin
  Open (File, In_File, "input.txt");
  exception
    when Name_Error =>
      Put ("Cannot open input file");
end Open_File;

Code block metadata
Project: Courses.Intro_To_Ada.Exceptions.Show_Exception_Message
MD5: 838e87ae416b3a717901cdc00eb71b40

Runtime output
Cannot open input file

Attention
Exception handlers have an important restriction that you need to be careful about: Exceptions raised in the declarative section are not caught by the handlers of that block. So for example, in the following code, the exception will not be caught.

Listing 6: be_careful.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Be_Careful is
  function Dangerous return Integer is
    begin
      raise Constraint_Error;
end Dangerous;

(continues on next page)
Learning Ada

return 42;
end Dangerous;

begin
declare
  A : Integer := Dangerous;
begin
  Put_Line (Integer'Image (A));
exception
  when Constraint_Error =>
    Put_Line ("error!");
end;
end Be_Careful;

Code block metadata

Project: Courses.Intro To Ada.Exceptions.Be_Careful
MD5: 6ea8a214bbbaca09d7444136d069e782

Runtime output

raised CONSTRAINT_ERROR : be_careful.adb:7 explicit raise

This is also the case for the top-level exception block that is part of the current subprogram.

14.4 Predefined exceptions

Ada has a very small number of predefined exceptions:

- **Constraint_Error** is the main one you might see. It's raised:
  - When bounds don't match or, in general, any violation of constraints.
  - In case of overflow
  - In case of null dereferences
  - In case of division by 0

- **Program_Error** might appear, but probably less often. It's raised in more arcane situations, such as for order of elaboration issues and some cases of detectable erroneous execution.

- **Storage_Error** will happen because of memory issues, such as:
  - Not enough memory (allocator)
  - Not enough stack

- **Tasking_Error** will happen with task related errors, such as any error happening during task activation.

You should not reuse predefined exceptions. If you do then, it won't be obvious when one is raised that it is because something went wrong in a built-in language operation.
Tasks and protected objects allow the implementation of concurrency in Ada. The following sections explain these concepts in more detail.

15.1 Tasks

A task can be thought as an application that runs concurrently with the main application. In other programming languages, a task might be called a thread\(^{18}\), and tasking might be called multithreading\(^{19}\).

Tasks may synchronize with the main application but may also process information completely independently from the main application. Here we show how this is accomplished.

15.1.1 Simple task

Tasks are declared using the keyword task. The task implementation is specified in a task body block. For example:

Listing 1: show_simple_task.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Task is
  task T is
    begin
      Put_Line ("In task T");
    end T;
  begin
    Put_Line ("In main");
  end Show_Simple_Task;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Simple_Task
MD5: b17d9b35b4b2b53bc59776749e1be219

Runtime output

In task T
In main

\(^{18}\) https://en.wikipedia.org/wiki/Thread_(computing)

\(^{19}\) https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading
Here, we're declaring and implementing the task T. As soon as the main application starts, task T starts automatically — it's not necessary to manually start this task. By running the application above, we can see that both calls to Put_Line are performed.

Note that:

- The main application is itself a task (the main or "environment" task).
  - In this example, the subprogram Show_Simple_Task is the main task of the application.
- Task T is a subtask.
  - Each subtask has a master, which represents the program construct in which the subtask is declared. In this case, the main subprogram Show_Simple_Task is T's master.
  - The master construct is executed by some enclosing task, which we will refer to as the "master task" of the subtask.
- The number of tasks is not limited to one: we could include a task T2 in the example above.
  - This task also starts automatically and runs concurrently with both task T and the main task. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Tasks is
  task T;
  task T2;

  task body T is
    begin
      Put_Line ("In task T");
      end T;

  task body T2 is
    begin
      Put_Line ("In task T2");
      end T2;

begin
  Put_Line ("In main");
end Show_Simple_Tasks;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Multiple_Simple_Task
MD5: 5e24b797e742bec306ad498f4f40d2b4

Runtime output

In task T
In task T2
In main
15.1.2 Simple synchronization

As we've just seen, as soon as the master construct reaches its “begin”, its subtasks also start automatically. The master continues its processing until it has nothing more to do. At that point, however, it will not terminate. Instead, the master waits until its subtasks have finished before it allows itself to complete. In other words, this waiting process provides synchronization between the master task and its subtasks. After this synchronization, the master construct will complete. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Sync is
  task T;
  task body T is
    begin
      for I in 1 .. 10 loop
        Put_Line ("hello");
      end loop;
    end T;
  begin
    null;
    -- Will wait here until all tasks
    -- have terminated
  end Show_Simple_Sync;
```

The same mechanism is used for other subprograms that contain subtasks: the subprogram execution will wait for its subtasks to finish. So this mechanism is not limited to the main subprogram and also applies to any subprogram called by the main subprogram, directly or indirectly.

Synchronization also occurs if we move the task to a separate package. In the example below, we declare a task T in the package Simple_Sync_Pkg.
Listing 4: simple_sync_pkg.ads

```ada
package Simple_Sync_Pkg is
task T;
end Simple_Sync_Pkg;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Tasking.Simple_Sync_Pkg
MD5: 2f9be044d04994240970f150e2293d5e

This is the corresponding package body:

Listing 5: simple_sync_pkg.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Simple_Sync_Pkg is
  task body T is
    begin
      for I in 1 .. 10 loop
        Put_Line ("hello");
      end loop;
    end T;
  end Simple_Sync_Pkg;
end Simple_Sync_Pkg;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Tasking.Simple_Sync_Pkg
MD5: b668451e4fb10e802f61989e8cd743ff

Because the package is `with`-ed by the main procedure, the task `T` defined in the package will become a subtask of the main task. For example:

Listing 6: test_simple_sync_pkg.adb

```ada
with Simple_Sync_Pkg;

procedure Test_Simple_Sync_Pkg is
begin
  null;
  -- Will wait here until all tasks
  -- have terminated
  end Test_Simple_Sync_Pkg;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Tasking.Simple_Sync_Pkg
MD5: e51565b91767ce198496ef3e9c582ac8

**Runtime output**

```
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

150  Chapter 15.  Tasking
As soon as the main subprogram returns, the main task synchronizes with any subtasks spawned by packages T from Simple_Sync_Pkg before finally terminating.

### 15.1.3 Delay

We can introduce a delay by using the keyword `delay`. This puts the current task to sleep for the length of time (in seconds) specified in the delay statement. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Delay is
  task T;
  task body T is
    begin
      for I in 1 .. 5 loop
        Put_Line("hello from task T");
        delay 1.0;
        -- ^ Wait 1.0 seconds
      end loop;
    end T;
    begin
      delay 1.5;
      Put_Line("hello from main");
    end Show_Delay;
end Show_Delay;
```

In this example, we're making the task T wait one second after each time it displays the "hello" message. In addition, the main task is waiting 1.5 seconds before displaying its own "hello" message.

### 15.1.4 Synchronization: rendezvous

The only type of synchronization we've seen so far is the one that happens automatically at the end of a master construct with a subtask. You can also define custom synchronization points using the keyword `entry`. An entry can be viewed as a special kind of subprogram, which is called by another task using a similar syntax, as we will see later.

In the task body definition, you define which part of the task will accept the entries by using the keyword `accept`. A task proceeds until it reaches an `accept` statement and then waits for some other task to synchronize with it. Specifically,

- The task with the entry waits at that point (in the `accept` statement), ready to accept a call to the corresponding entry from the master task.
Learning Ada

- The other task calls the task entry, in a manner similar to a procedure call, to synchronize with the entry.

This synchronization between tasks is called a rendezvous. Let’s see an example:

Listing 8: show_rendezvous.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Rendezvous is

  task T is
    entry Start;
  end T;

  task body T is
    begin
      accept Start;
      -- ^ Waiting for somebody
      -- to call the entry
      Put_Line ("In T");
    end T;
  begin
    Put_Line ("In Main");
    -- Calling T's entry:
    T.Start;
  end Show_Rendezvous;
```

In this example, we declare an entry Start for task T. In the task body, we implement this entry using accept Start. When task T reaches this point, it waits for some other task to call its entry. This synchronization occurs in the T.Start statement. After the rendezvous completes, the main task and task T again run concurrently until they synchronize one final time when the main subprogram Show_Rendezvous finishes.

An entry may be used to perform more than a simple task synchronization: it also may perform multiple statements during the time both tasks are synchronized. We do this with a do ... end block. For the previous example, we would simply write accept Start do <statements>; end;. We use this kind of block in the next example.
15.1.5 Select loop

There's no limit to the number of times an entry can be accepted. We could even create an infinite loop in the task and accept calls to the same entry over and over again. An infinite loop, however, prevents the subtask from finishing, so it blocks its master task when it reaches the end of its processing. Therefore, a loop containing accept statements in a task body can be used in conjunction with a select ... or terminate statement. In simple terms, this statement allows its master task to automatically terminate the subtask when the master construct reaches its end. For example:

Listing 9: show_rendezvous_loop.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Rendezvous_Loop is

  task T is
    entry Reset;
    entry Increment;
  end T;

  task body T is
    Cnt : Integer := 0;
  begin
    loop
      select
      accept Reset do
        Cnt := 0;
        end Reset;
        Put_Line ("Reset");
      or
      accept Increment do
        Cnt := Cnt + 1;
        end Increment;
        Put_Line ("In T's loop (" & Integer'Image (Cnt) & ")");
      or
      terminate;
      end select;
    end loop;
  end T;

  begin
    Put_Line ("In Main");
    for I in 1 .. 4 loop
      -- Calling T's entry multiple times
      T.Increment;
    end loop;
    T.Reset;
    for I in 1 .. 4 loop
      -- Calling T's entry multiple times
      T.Increment;
    end loop;
  end Show_Rendezvous_Loop;
```

Code block metadata
In this example, the task body implements an infinite loop that accepts calls to the Reset and Increment entry. We make the following observations:

- The accept E do ... end block is used to increment a counter.
  - As long as task T is performing the do ... end block, the main task waits for the block to complete.
- The main task is calling the Increment entry multiple times in the loop from 1 .. 4. It is also calling the Reset entry before the second loop.
  - Because task T contains an infinite loop, it always accepts calls to the Reset and Increment entries.
  - When the master construct of the subtask (the Show_Rendezvous_Loop subprogram) completes, it checks the status of the T task. Even though task T could accept new calls to the Reset or Increment entries, the master construct is allowed to terminate task T due to the or terminate part of the select statement.

### 15.1.6 Cycling tasks

In a previous example, we saw how to delay a task a specified time by using the delay keyword. However, using delay statements in a loop is not enough to guarantee regular intervals between those delay statements. For example, we may have a call to a computationally intensive procedure between executions of successive delay statements:

```ada
while True loop
   delay 1.0; -- ^ Wait 1.0 seconds
   Computational_Intensive_App;
end loop;
```

In this case, we can't guarantee that exactly 10 seconds have elapsed after 10 calls to the delay statement because a time drift may be introduced by the Computational_Intensive_App procedure. In many cases, this time drift is not relevant, so using the delay keyword is good enough.

However, there are situations where a time drift isn't acceptable. In those cases, we need to use the delay until statement, which accepts a precise time for the end of the delay, allowing us to define a regular interval. This is useful, for example, in real-time applications.

We will soon see an example of how this time drift may be introduced and how the delay until statement circumvents the problem. But before we do that, we look at a package containing a procedure allowing us to measure the elapsed time (Show_Elapsed_Time) and a dummy Computational_Intensive_App procedure which is simulated by using a simple delay. This is the complete package:
with Ada.Real_Time; use Ada.Real_Time;

package Delay_Aux_Pkg is

  function Get_Start_Time return Time
    with Inline;

  procedure Show_Elapsed_Time
    with Inline;

  procedure Computational_Intensive_App;

private

  Start_Time : Time := Clock;

function Get_Start_Time return Time is (Start_Time);

end Delay_Aux_Pkg;

with Ada.Text_IO; use Ada.Text_IO;

package body Delay_Aux_Pkg is

  procedure Show_Elapsed_Time is
    Now_Time : Time;
    Elapsed_Time : Time_Span;
    begin
      Now_Time := Clock;
      Elapsed_Time := Now_Time - Start_Time;
      Put_Line ("Elapsed time ".
        Duration'Image
        (To_Duration (Elapsed_Time))
        " seconds");
    end Show_Elapsed_Time;

  procedure Computational_Intensive_App is
    begin
      delay 0.5;
    end Computational_Intensive_App;

end Delay_Aux_Pkg;

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Time
MD5: 422a38c1afa0bbd659ec81de88479e0a

Using this auxiliary package, we’re now ready to write our time-drifting application:

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
with Delay_Aux_Pkg;

procedure Show_Time_Task is

(continues on next page)
package Aux renames Delay_Aux_Pkg;

task T;

task body T is
  Cnt : Integer := 1;
begin
  for I in 1..5 loop
    delay 1.0;
    Aux.Show_Elapsed_Time;
    Aux.Computational_Intensive_App;
    Put_Line ("Cycle # " & Integer'Image (Cnt));
    Cnt := Cnt + 1;
  end loop;
  Put_Line ("Finished time-drifting loop");
end T;
begin
  null;
end Show_Time_Task;

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Time
MD5: fe17c902fc127c0132677ea4005ff3f1

Runtime output

Elapsed time 1.000528681 seconds
Cycle # 1
Elapsed time 2.501496814 seconds
Cycle # 2
Elapsed time 4.002310614 seconds
Cycle # 3
Elapsed time 5.502655881 seconds
Cycle # 4
Elapsed time 7.012303672 seconds
Cycle # 5
Finished time-drifting loop

We can see by running the application that we already have a time difference of about four seconds after three iterations of the loop due to the drift introduced by Computational_Intensive_App. Using the delay_until statement, however, we're able to avoid this time drift and have a regular interval of exactly one second:

Listing 13: show_time_task.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
with Delay_Aux_Pkg;

procedure Show_Time_Task is
  package Aux renames Delay_Aux_Pkg;

  task T;

  task body T is
Cycle : constant Time_Span :=
  Milliseconds (1000);
Next : Time := Aux.Get_Start_Time
  + Cycle;

Cnt : Integer := 1;
beginn
  for I in 1 .. 5 loop
    delay until Next;
    Aux.Show_Elapsed_Time;
    Aux.Computational_Intensive_App;
    -- Calculate next execution time
    -- using a cycle of one second
    Next := Next + Cycle;
    Put_Line ("Cycle # 
      & Integer'Image (Cnt));
    Cnt := Cnt + 1;
  end loop;
  Put_Line ("Finished cycling");
end T;

begin
  null;
end Show_Time_Task;

Code block metadata
Project: Courses.Intro_To_Ada.Tasking.Show_Time
MD5: 1456c8f0ee6def8b370d994c0ab75a15

Runtime output
Elapsed time 1.000196297 seconds
Cycle # 1
Elapsed time 2.000175312 seconds
Cycle # 2
Elapsed time 3.000143616 seconds
Cycle # 3
Elapsed time 4.000112505 seconds
Cycle # 4
Elapsed time 5.000108386 seconds
Cycle # 5
Finished cycling

Now, as we can see by running the application, the delay until statement ensures that the Computational_Intensive_App doesn't disturb the regular interval of one second between iterations.
15.2 Protected objects

When multiple tasks are accessing shared data, corruption of that data may occur. For example, data may be inconsistent if one task overwrites parts of the information that's being read by another task at the same time. In order to avoid these kinds of problems and ensure information is accessed in a coordinated way, we use protected objects.

Protected objects encapsulate data and provide access to that data by means of protected operations, which may be subprograms or protected entries. Using protected objects ensures that data is not corrupted by race conditions or other concurrent access.

Important

Objects can be protected from concurrent access using Ada tasks. In fact, this was the only way of protecting objects from concurrent access in Ada 83 (the first version of the Ada language). However, the use of protected objects is much simpler than using similar mechanisms implemented using only tasks. Therefore, you should use protected objects when your main goal is only to protect data.

15.2.1 Simple object

You declare a protected object with the protected keyword. The syntax is similar to that used for packages: you can declare operations (e.g., procedures and functions) in the public part and data in the private part. The corresponding implementation of the operations is included in the protected body of the object. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Protected_Objects is

protected
    Obj is
        procedure Set (V : Integer);
        function Get return Integer;

    private
        Local : Integer := 0;

protected body Obj is
    procedure Set (V : Integer) is
        begin
            Local := V;
        end Set;

    function Get return Integer is
        begin
            return Local;
        end Get;

    begin
        Obj.Set (5);
    end
```

(continues on next page)
In this example, we define two operations for Obj: Set and Get. The implementation of these operations is in the Obj body. The syntax used for writing these operations is the same as that for normal procedures and functions. The implementation of protected objects is straightforward — we simply access and update Local in these subprograms. To call these operations in the main application, we use prefixed notation, e.g., Obj.Get.

15.2.2 Entries

In addition to protected procedures and functions, you can also define protected entry points. Do this using the entry keyword. Protected entry points allow you to define barriers using the when keyword. Barriers are conditions that must be fulfilled before the entry can start performing its actual processing — we speak of releasing the barrier when the condition is fulfilled.

The previous example used procedures and functions to define operations on the protected objects. However, doing so permits reading protected information (via Obj.Get) before it's set (via Obj.Set). To allow that to be a defined operation, we specified a default value (0). Instead, by rewriting Obj.Get using an entry instead of a function, we implement a barrier, ensuring no task can read the information before it's been set.

The following example implements the barrier for the Obj.Get operation. It also contains two concurrent subprograms (main task and task T) that try to access the protected object.

Listing 15: show_protected_objects_entries.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Protected_Objects_Entries is
  protected Obj is
    procedure Set (V : Integer);
      entry Get (V : out Integer);
  private
    Local : Integer;
    Is_Set : Boolean := False;
  end Obj;

  protected body Obj is
    procedure Set (V : Integer) is
      begin
        Local := V;
        Is_Set := True;
      end Set;

    entry Get (V : out Integer)
      when Is_Set is
          V;

end Show_Protected_Objects_Entries;
```

(continued on next page)
-- Entry is blocked until the
-- condition is true. The barrier
-- is evaluated at call of entries
-- and at exits of procedures and
-- entries. The calling task sleeps
-- until the barrier is released.
begin
  V := Local;
  Is_Set := False;
end Get;
end Obj;

N : Integer := 0;
task T;
task body T is
begin
  Put_Line
    ("Task T will delay for 4 seconds...");
delay 4.0;
Put_Line
  ("Task T will set Obj...");
Obj.Set (5);
Put_Line
  ("Task T has just set Obj...");
end T;
begin
  Put_Line
    ("Main application will get Obj...");
Obj.Get (N);
Put_Line
  ("Main application has retrieved Obj...");
Put_Line
  ("Number is: " & Integer'Image (N));
end Show_Protected_Objects_Entries;

As we see by running it, the main application waits until the protected object is set (by the
call to Obj.Set in task T) before it reads the information (via Obj.Get). Because a 4-second
delay has been added in task T, the main application is also delayed by 4 seconds. Only
after this delay does task T set the object and release the barrier in Obj.Get so that the
main application can then resume processing (after the information is retrieved from the
protected object).
15.3 Task and protected types

In the previous examples, we defined single tasks and protected objects. We can, however, generalize tasks and protected objects using typedefinitions. This allows us, for example, to create multiple tasks based on just a single task type.

15.3.1 Task types

A task type is a generalization of a task. The declaration is similar to simple tasks: you replace task with task type. The difference between simple tasks and task types is that task types don't create actual tasks that automatically start. Instead, a task object declaration is needed. This is exactly the way normal variables and types work: objects are only created by variable definitions, not type definitions.

To illustrate this, we repeat our first example:

Listing 16: show_simple_task.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Task is
  task T;
  task body T is
    begin
      Put_Line ("In task T");
      end T;
  begin
    Put_Line ("In main");
  end Show_Simple_Task;
```

We now rewrite it by replacing task T with task type TT. We declare a task (A_Task) based on the task type TT after its definition:

Listing 17: show_simple_task_type.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Task_Type is
  task type TT;
  task body TT is
    begin
      Put_Line ("In task type TT");
      end TT;
  A_Task : TT;
  begin
    Put_Line ("In main");
  end Show_Simple_Task_Type;
```
We can extend this example and create an array of tasks. Since we're using the same syntax as for variable declarations, we use a similar syntax for task types: `array (<>) of Task_Type`. Also, we can pass information to the individual tasks by defining a Start entry. Here's the updated example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Task_Type_Array is
  task type TT is
    entry Start (N : Integer);
  end TT;

  task body TT is
    Task_N : Integer;
  begin
    accept Start (N : Integer) do
      Task_N := N;
    end Start;
    Put_Line ("In task T: " & Integer'Image (Task_N));
  end TT;

  My_Tasks : array (1 .. 5) of TT;
begin
  Put_Line ("In main");
  for I in My_Tasks'Range loop
    My_Tasks (I).Start (I);
  end loop;
end Show_Task_Type_Array;
```

In this example, we're declaring five tasks in the array My_Tasks. We pass the array index to the individual tasks in the entry point (Start). After the synchronization between the individual subtasks and the main task, each subtask calls Put_Line concurrently.
15.3.2 Protected types

A protected type is a generalization of a protected object. The declaration is similar to that for protected objects: you replace `protected` with `protected type`. Like task types, protected types require an object declaration to create actual objects. Again, this is similar to variable declarations and allows for creating arrays (or other composite objects) of protected objects.

We can reuse a previous example and rewrite it to use a protected type:

Listing 19: show_protected_object_type.adb

```adaprogram
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Protected_Object_Type is
  protected type P_Obj_Type is
    procedure Set (V : Integer);
    function Get return Integer;
  private
    Local : Integer := 0;
  end P_Obj_Type;

protected body P_Obj_Type is
  procedure Set (V : Integer) is
    begin
      Local := V;
    end Set;

  function Get return Integer is
    begin
      return Local;
    end Get;
  end P_Obj_Type;

  Obj : P_Obj_Type;

begin
  Obj.Set (5);
  Put_Line ("Number is: "
            & Integer'Image (Obj.Get));
end Show_Protected_Object_Type;
```

In this example, instead of directly defining the protected object `Obj`, we first define a protected type `P_Obj_Type` and then declare `Obj` as an object of that protected type. Note that the main application hasn't changed: we still use `Obj.Set` and `Obj.Get` to access the protected object, just like in the original example.
Contracts are used in programming to codify expectations. Parameter modes of a subprogram can be viewed as a simple form of contracts. When the specification of subprogram \( \text{Op} \) declares a parameter using \textit{in} mode, the caller of \( \text{Op} \) knows that the \textit{in} argument won't be changed by \( \text{Op} \). In other words, the caller expects that \( \text{Op} \) doesn't modify the argument it's providing, but just reads the information stored in the argument. Constraints and subtypes are other examples of contracts. In general, these specifications improve the consistency of the application.

\textit{Design-by-contract} programming refers to techniques that include pre- and postconditions, subtype predicates, and type invariants. We study those topics in this chapter.

### 16.1 Pre- and postconditions

Pre- and postconditions provide expectations regarding input and output parameters of subprograms and return value of functions. If we say that certain requirements must be met before calling a subprogram \( \text{Op} \), those are preconditions. Similarly, if certain requirements must be met after a call to the subprogram \( \text{Op} \), those are postconditions. We can think of preconditions and postconditions as promises between the subprogram caller and the callee: a preconditions is a promise from the caller to the callee, and a postconditions is a promise in the other direction.

Pre- and postconditions are specified using an aspect clause in the subprogram declaration. A \texttt{with Pre => <condition>} clause specifies a precondition and a \texttt{with Post => <condition>} clause specifies a postcondition.

The following code shows an example of preconditions:

```ada
procedure Show_Simple_Precondition is

    procedure DB_Entry (Name : String; Age : Natural)
    with Pre => Name'Length > 0
    begin
        -- Missing implementation
        null;
    end DB_Entry;

begin
    DB_Entry ("John", 30);
    -- Precondition will fail!
    DB_Entry (",", 21);
end Show_Simple_Precondition;
```

Listing 1: show_simple_precondition.adb
In this example, we want to prevent the name field in our database from containing an empty string. We implement this requirement by using a precondition requiring that the length of the string used for the Name parameter of the DB_Entry procedure is greater than zero. If the DB_Entry procedure is called with an empty string for the Name parameter, the call will fail because the precondition is not met.

In the GNAT toolchain

GNAT handles pre- and postconditions by generating runtime assertions for them. By default, however, assertions aren't enabled. Therefore, in order to check pre- and postconditions at runtime, you need to enable assertions by using the -gnata switch.

Before we get to our next example, let's briefly discuss quantified expressions, which are quite useful in concisely writing pre- and postconditions. Quantified expressions return a Boolean value indicating whether elements of an array or container match the expected condition. They have the form: (for all I in A'Range => <condition on A(I)>), where A is an array and I is an index. Quantified expressions using for all check whether the condition is true for every element. For example:

(\texttt{for all I in A'Range} => A(I) = 0)

This quantified expression is only true when all elements of the array A have a value of zero.

Another kind of quantified expression uses 	exttt{for some}. The form looks similar: (for some I in A'Range => <condition on A(I)>). However, in this case the qualified expression tests whether the condition is true only on some elements (hence the name) instead of all elements.

We illustrate postconditions using the following example:

Listing 2: show_simple_postcondition.adb

1. with Ada.Text_IO; use Ada.Text_IO;
2. procedure Show_Simple_Postcondition is
3.   type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;
4.   type Int_8_Array is
5.     array (Integer range <>) of Int_8;
6.   function Square (A : Int_8) return Int_8 is
7.     (A * A)
8.       with Post => (if abs A in 0 | 1
9.         then Square'Result = abs A
10.        else Square'Result > A);
11. procedure Square (A : in out Int_8_Array)
12.   with Post => (for all I in A'Range =>
13.       A(I) = A'Old(I) *
We declare a signed 8-bit type Int_8 and an array of that type (Int_8_Array). We want to ensure each element of the array is squared after calling the procedure Square for an object of the Int_8_Array type. We do this with a postcondition using a for all expression. This postcondition also uses the 'Old attribute to refer to the original value of the parameter (before the call).

We also want to ensure that the result of calls to the Square function for the Int_8 type are greater than the input to that call. To do that, we write a postcondition using the 'Result attribute of the function and comparing it to the input value.

We can use both pre- and postconditions in the declaration of a single subprogram. For example:

Listing 3: show_simple_contract.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
```
procedure Show_Simple_Contract is

    type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;

    function Square (A : Int_8) return Int_8 is
        (A * A)
    with
    Pre => (Integer'Size >= Int_8'Size * 2
        and Integer (A) *
        Integer (A) <=
        Integer (Int_8'Last)),
    Post => (if abs A in 0 | 1
        then Square'Result = abs A
        else Square'Result > A);

    V : Int_8;

    begin
        V := Square (11);
        Put_Line ("Square of 11 is 
            & Int_8'Image (V));

        -- Precondition will fail...
        V := Square (12);
        Put_Line ("Square of 12 is 
            & Int_8'Image (V));
        end Show_Simple_Contract;

In this example, we want to ensure that the input value of calls to the Square function for the Int_8 type won't cause overflow in that function. We do this by converting the input value to the Integer type, which is used for the temporary calculation, and check if the result is in the appropriate range for the Int_8 type. We have the same postcondition in this example as in the previous one.

16.2 Predicates

Predicates specify expectations regarding types. They're similar to pre- and postconditions, but apply to types instead of subprograms. Their conditions are checked for each object of a given type, which allows verifying that an object of type T is conformant to the requirements of its type.

There are two kinds of predicates: static and dynamic. In simple terms, static predicates are used to check objects at compile-time, while dynamic predicates are used for checks at run time. Normally, static predicates are used for scalar types and dynamic predicates for the more complex types.

Static and dynamic predicates are specified using the following clauses, respectively:
• with Static_Predicate => <property>
• with Dynamic_Predicate => <property>

Let's use the following example to illustrate dynamic predicates:

Listing 4: show_dynamic_predicate_courses.adb

```ada
with Ada.Calendar; use Ada.Calendar;
with Ada.Containers.Vectors;
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

procedure Show_Dynamic_Predicate_Courses is

package Courses is
  type Course_Container is private;

type Course is record
    Name : Unbounded_String;
    Start_Date : Time;
    End_Date : Time;
  end record
  with Dynamic_Predicate =>
    Course.Start_Date <= Course.End_Date;

  procedure Add (CC : in out Course_Container;
                C : Course);
private
  package Course_Vectors is new
    Ada.Containers.Vectors
      (Index_Type => Natural,
       Element_Type => Course);

  type Course_Container is record
    V : Course_Vectors.Vector;
  end record;
end Courses;

package body Courses is
  procedure Add (CC : in out Course_Container;
                 C : Course) is
    begin
      CC.V.Append (C);
    end Add;
end Courses;

use Courses;

CC : Course_Container;
begin
  Add (CC,
       Course'(
         Name =>
           To_Unbounded_String
             ("Intro to Photography"),
         Start_Date =>
           Time_Of (2018, 5, 1),
         End_Date =>
           Time_Of (2018, 5, 10)));
```

(continues on next page)
In this example, the package Courses defines a type Course and a type Course_Container, an object of which contains all courses. We want to ensure that the dates of each course are consistent, specifically that the start date is no later than the end date. To enforce this rule, we declare a dynamic predicate for the Course type that performs the check for each object. The predicate uses the type name where a variable of that type would normally be used: this is a reference to the instance of the object being tested.

Note that the example above makes use of unbounded strings and dates. Both types are available in Ada's standard library. Please refer to the following sections for more information about:

- the unbounded string type (Unbounded_String): Unbounded Strings (page 246) section;
- dates and times: Dates & Times (page 231) section.

Static predicates, as mentioned above, are mostly used for scalar types and checked during compilation. They're particularly useful for representing non-contiguous elements of an enumeration. A classic example is a list of week days:

```ada
type Week is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

We can easily create a sub-list of work days in the week by specifying a subtype with a range based on Week. For example:

```ada
subtype Work_Week is Week range Mon .. Fri;
```

Ranges in Ada can only be specified as contiguous lists: they don't allow us to pick specific days. However, we may want to create a list containing just the first, middle and last day of the work week. To do that, we use a static predicate:

```ada
subtype Check_Days is Work_Week
  with Static_Predicate =>
    Check_Days in Mon | Wed | Fri;
```

Let's look at a complete example:
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Predicates is

  type Week is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  subtype Work_Week is Week range Mon .. Fri;

  subtype Test_Days is Work_Week
  with Static_Predicate =>
    Test_Days in Mon | Wed | Fri;

  type Tests_Week is array (Week) of Natural
  with Dynamic_Predicate =>
    (for all I in Tests_Week'Range =>
      case I is
      when Test_Days =>
        Tests_Week (I) > 0,
      when others =>
        Tests_Week (I) = 0);

  Num_Tests : Tests_Week :=
    (Mon => 3, Tue => 0,
     Wed => 4, Thu => 0,
     Fri => 2, Sat => 0,
     Sun => 0);

procedure Display_Tests (N : Tests_Week) is
begin
  for I in Test_Days loop
    Put_Line ("# tests on 
      & Test_Days'Image (I)
      & " => "
      & Integer'Image (N (I)));
  end loop;
end Display_Tests;

begin
  Display_Tests (Num_Tests);

  -- Assigning non-conformant values to
  -- individual elements of the Tests_Week
  -- type does not trigger a predicate
  -- check:
  Num_Tests (Tue) := 2;

  -- However, assignments with the "complete"
  -- Tests_Week type trigger a predicate
  -- check. For example:
  -- Num_Tests := (others => 0);

  -- Also, calling any subprogram with
  -- parameters of Tests_Week type
  -- triggers a predicate check. Therefore,
  -- the following line will fail:
  Display_Tests (Num_Tests);
end Show_Predicates;
Here we have an application that wants to perform tests only on three days of the work week. These days are specified in the `Test_Days` subtype. We want to track the number of tests that occur each day. We declare the type `Tests_Week` as an array, an object of which will contain the number of tests done each day. According to our requirements, these tests should happen only in the aforementioned three days; on other days, no tests should be performed. This requirement is implemented with a dynamic predicate of the type `Tests_Week`. Finally, the actual information about these tests is stored in the array `Num_Tests`, which is an instance of the `Tests_Week` type.

The dynamic predicate of the `Tests_Week` type is verified during the initialization of `Num_Tests`. If we have a non-conformant value there, the check will fail. However, as we can see in our example, individual assignments to elements of the array do not trigger a check. We can't check for consistency at this point because the initialization of the a complex data structure (such as arrays or records) may not be performed with a single assignment. However, as soon as the object is passed as an argument to a subprogram, the dynamic predicate is checked because the subprogram requires the object to be consistent. This happens in the last call to `Display_Tests` in our example. Here, the predicate check fails because the previous assignment has a non-conformant value.

**16.3 Type invariants**

Type invariants are another way of specifying expectations regarding types. While predicates are used for non-private types, type invariants are used exclusively to define expectations about private types. If a type `T` from a package `P` has a type invariant, the results of operations on objects of type `T` are always consistent with that invariant.

Type invariants are specified with a `with Type_Invariant => <property>` clause. Like predicates, the `property` defines a condition that allows us to check if an object of type `T` is conformant to its requirements. In this sense, type invariants can be viewed as a sort of predicate for private types. However, there are some differences in terms of checks. The following table summarizes the differences:

<table>
<thead>
<tr>
<th>Element</th>
<th>Subprogram parameter checks</th>
<th>Assignment checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicates</td>
<td>On all <code>in</code> and <code>out</code> parameters</td>
<td>On assignments and explicit initializations</td>
</tr>
<tr>
<td>Type invariants</td>
<td>On <code>out</code> parameters returned from subprograms declared in the same public scope</td>
<td>On all initializations</td>
</tr>
</tbody>
</table>

We could rewrite our previous example and replace dynamic predicates by type invariants. It would look like this:
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;
with Ada.Containers.Vectors;
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

procedure Show_Type_Invariant is

package Courses is
  type Course is private
    with Type_Invariant => Check (Course);

type Course_Container is private;

procedure Add (CC : in out Course_Container;
  C : Course);

function Init (Name : String;
  Start_Date, End_Date : Time)
  return Course;

function Check (C : Course)
  return Boolean;

private
  type Course is record
    Name : Unbounded_String;
    Start_Date : Time;
    End_Date : Time;
  end record;

  function Check (C : Course)
    return Boolean is
      (C.Start_Date <= C.End_Date);

package Course_Vectors is new
  Ada.Containers.Vectors
    (Index_Type => Natural,
     Element_Type => Course);

type Course_Container is record
  V : Course_Vectors.Vector;
end record;
end Courses;

package body Courses is
  procedure Add (CC : in out Course_Container;
    C : Course) is
  begin
    CC.V.Append (C);
  end Add;

  function Init (Name : String;
    Start_Date, End_Date : Time)
    return Course is

(continues on next page)
begin
  return
    Course'(Name =>
      To_Unbounded_String (Name),
      Start_Date => Start_Date,
      End_Date  => End_Date);
end Init;
end Courses;

use Courses;

CC : Course_Container;
begin
  Add (CC,
    Init (Name =>
      "Intro to Photography",
      Start_Date =>
        Time_Of (2018, 5, 1),
      End_Date  =>
        Time_Of (2018, 5, 10)));

  -- This should trigger an error in the
  -- type-invariant check
  Add (CC,
    Init (Name =>
      "Intro to Video Recording",
      Start_Date =>
        Time_Of (2019, 5, 1),
      End_Date  =>
        Time_Of (2018, 5, 10)));
end Show_Type_Invariant;

The major difference is that the Course type was a visible (public) type of the Courses package in the previous example, but in this example is a private type.
Ada allows us to interface with code in many languages, including C and C++. This section discusses how to interface with C.

### 17.1 Multi-language project

By default, when using `gprbuild` we only compile Ada source files. To compile C files as well, we need to modify the project file used by `gprbuild`. We use the `Languages` entry, as in the following example:

```ada
project Multilang is
  for Languages use ("ada", "c");
  for Source_Dirs use ("src");
  for Main use ("main.adb");
  for Object_Dir use "obj";
end Multilang;
```

### 17.2 Type convention

To interface with data types declared in a C application, you specify the `Convention` aspect on the corresponding Ada type declaration. In the following example, we interface with the `C_Enum` enumeration declared in a C source file:

```ada
procedure Show_C_Enum is
  type C_Enum is (A, B, C)
    with Convention => C;
  -- Use C convention for C_Enum
begin
  null;
end Show_C_Enum;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Enum
MD5: a14d7d981fd7d6d806cf3c55f35e19c8
To interface with C's built-in types, we use the Interfaces.C package, which contains most of the type definitions we need. For example:

```ada
with Interfaces.C; use Interfaces.C;
procedure Show_C_Struct is
  type c_struct is record
    a : int;
    b : long;
    c : unsigned;
    d : double;
  end record
  with Convention => C;
begin
  null;
end Show_C_Struct;
```

Here, we're interfacing with a C struct (C_Struct) and using the corresponding data types in C (int, long, unsigned and double). This is the declaration in C:

```c
struct c_struct{
  int a;
  long b;
  unsigned c;
  double d;
};
```

### 17.3 Foreign subprograms

#### 17.3.1 Calling C subprograms in Ada

We use a similar approach when interfacing with subprograms written in C. Consider the following declaration in the C header file:

```c
int my_func (int a);
```
Here's the corresponding C definition:

```c
#include "my_func.h"

int my_func (int a)
{
    return a * 2;
}
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_FUNC
MD5: 284b1639cb393fc14ed196d78429f3ba

We can interface this code in Ada using the Import aspect. For example:

```ada
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_C_Func is

    function my_func (a : int) return int
        with
        Import => True,
        Convention => C;

    -- Imports function 'my_func' from C.
    -- You can now call it from Ada.

    V : int;

begin
    V := my_func (2);
    Put_Line ("Result is " & int'Image (V));
end Show_C_Func;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_FUNC
MD5: 6c5d85c1debdeaa642946eacf413dfd2

If you want, you can use a different subprogram name in the Ada code. For example, we could call the C function Get_Value:

```ada
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_C_Func is

    function Get_Value (a : int) return int
        with
        Import => True,
        Convention => C,
        External_Name => "my_func";

    -- Imports function 'my_func' from C and
    -- renames it to 'Get_Value'
```

(continues on next page)

**17.3. Foreign subprograms**
Learning Ada

V : int;
begin
  V := Get_Value (2);
  Put_Line ("Result is " & int'Image (V));
end Show_C_Func;

17.3.2 Calling Ada subprograms in C

You can also call Ada subprograms from C applications. You do this with the Export aspect. For example:

Listing 8: c_api.ads

```ada
with Interfaces.C; use Interfaces.C;
package C_API is
  function My_Func (a : int) return int
    with
    Export => True,
    Convention => C,
    External_Name => "my_func";
end C_API;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Func
MD5: 856b4d99d6aa6946f4597f254fd2f97

This is the corresponding body that implements that function:

Listing 9: c_api.adb

```ada
package body C_API is
  function My_Func (a : int) return int
    begin
      return a * 2;
  end My_Func;
end C_API;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Func
MD5: 00aa4ec29fc551e710900e2ee7d96bc9

On the C side, we do the same as we would if the function were written in C: simply declare it using the extern keyword. For example:
Listing 10: main.c

```c
#include <stdio.h>

extern int my_func (int a);

int main (int argc, char **argv) {
    int v = my_func(2);
    printf("Result is %d\n", v);
    return 0;
}
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Ada_Func
MD5: 69301036be9be16ed45895c2a86bc352

17.4 Foreign variables

17.4.1 Using C global variables in Ada

To use global variables from C code, we use the same method as subprograms: we specify the Import and Convention aspects for each variable we want to import.

Let's reuse an example from the previous section. We'll add a global variable (func_cnt) to count the number of times the function (my_func) is called:

Listing 11: test.h

```c
extern int func_cnt;

int my_func (int a);
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Vars
MD5: 11ba8f7a72ce7058571994870a02b052

The variable is declared in the C file and incremented in my_func:

Listing 12: test.c

```c
#include "test.h"

int func_cnt = 0;

int my_func (int a)
{
    func_cnt++;
    return a * 2;
}
```

**Code block metadata**
In the Ada application, we just reference the foreign variable:

```
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_C_Func is
  function my_func (a : int) return int
    with
      Import => True,
      Convention => C;
  V : int;
  func_cnt : int
    with
      Import => True,
      Convention => C;
  -- We can access the func_cnt variable
  -- from test.c

begin
  V := my_func (1);
  V := my_func (2);
  V := my_func (3);

  Put_Line ("Result is 
            & int’Image (V));

  Put_Line ("Function was called 
            & int’Image (func_cnt)
            & " times");
end Show_C_Func;
```

As we see by running the application, the value of the counter is the number of times my_func was called.

We can use the External_Name aspect to give a different name for the variable in the Ada application in the same way we do for subprograms.
17.4.2 Using Ada variables in C

You can also use variables declared in Ada files in C applications. In the same way as we did for subprograms, you do this with the Export aspect.

Let's reuse a past example and add a counter, as in the previous example, but this time have the counter incremented in Ada code:

Listing 14: c_api.ads

```ada
with Interfaces.C; use Interfaces.C;
package C_API is
  func_cnt : int := 0
  with
    Export => True,
    Convention => C;
  function My_Func (a : int) return int
    with
      Export => True,
      Convention => C,
      External_Name => "my_func";
end C_API;
```

The variable is then incremented in My_Func:

Listing 15: c_api.adb

```ada
package body C_API is
  function My_Func (a : int) return int is
    begin
      func_cnt := func_cnt + 1;
      return a * 2;
    end My_Func;
end C_API;
```

In the C application, we just need to declare the variable and use it:

Listing 16: main.c

```c
#include <stdio.h>
extern int my_func (int a);
extern int func_cnt;
int main (int argc, char **argv) {
  (continues on next page)
```
int v;

v = my_func(1);
v = my_func(2);
v = my_func(3);

printf("Result is %d\n", v);
printf("Function was called %d times\n", func_cnt);

return 0;
}

182 Chapter 17. Interfacing with C
Learning Ada

(continued from previous page)

```ada
func_cnt : aliased int; -- ./test.h:3
pragma Import (C, func_cnt, "func_cnt");

function my_func (arg1 : int) return int; -- ./test.h:5
pragma Import (C, my_func, "my_func");

end test_h;
```

### Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds
MD5: 8d18aae72da3a9ab4f9f3943fab839

Now we simply refer to this test_h package in our Ada application:

Listing 19: show_c_func.adb

```ada
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO; use Ada.Text_IO;
with test_h; use test_h;

procedure Show_C_Func is
  V : int;
begin
  V := my_func (1);
  V := my_func (2);
  V := my_func (3);
  Put_Line ("Result is 
            & int'Image (V));
  Put_Line ("Function was called 
            & int'Image (func_cnt)
            & " times");
end Show_C_Func;
```

### Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds
MD5: 8a07aae87b9f36c3fce84b75e8388933

You can specify the name of the parent unit for the bindings you’re creating as the operand to fdump-ada-spec:

```bash
gcc -c -fdump-ada-spec -fada-spec-parent=Ext_C_Code -C ./test.h
```

This creates the file ext_c_code-test_h.ads:

Listing 20: ext_c_code-test_h.ads

```ada
package Ext_C_Code.test_h is
  -- automatic generated bindings...
end Ext_C_Code.test_h;
```

### Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_2
MD5: 3bd4087edff145a70d2a6db8543859ad

17.5. Generating bindings
17.5.1 Adapting bindings

The compiler does the best it can when creating bindings for a C header file. However, sometimes it has to guess about the translation and the generated bindings don’t always match our expectations. For example, this can happen when creating bindings for functions that have pointers as arguments. In this case, the compiler may use System.Address as the type of one or more pointers. Although this approach works fine (as we’ll see later), this is usually not how a human would interpret the C header file. The following example illustrates this issue.

Let’s start with this C header file:

Listing 21: test.h

```c
struct test;

struct test * test_create(void);

void test_destroy(struct test *t);

void test_reset(struct test *t);

void test_set_name(struct test *t, char *name);

void test_set_address(struct test *t, char *address);

void test_display(const struct test *t);
```

And the corresponding C implementation:

Listing 22: test.c

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "test.h"

struct test {
    char name[80];
    char address[120];
};

static size_t strlcpy_stat(char *dst, const char *src, size_t dstsize)
{
    size_t len = strlen(src);
    if (dstsize) {
        size_t bl = (len < dstsize-1 ?
                     len : dstsize-1);
        ((char*)memcpys(dst, src, bl))[bl] = 0;
    }
    return len;
```

(continues on next page)
struct test *
test_create(void)
{
    return malloc(sizeof(struct test));
}

void test_destroy(struct test *t)
{
    if (t != NULL) {
        free(t);
    }
}

void test_reset(struct test *t)
{
    t->name[0] = '\0';
    t->address[0] = '\0';
}

void test_set_name(struct test *t,
                    char      *name)
{
    strlcpy_stat(t->name,
                  name,
                  sizeof(t->name));
}

void test_set_address(struct test *t,
                       char      *address)
{
    strlcpy_stat(t->address,
                  address,
                  sizeof(t->address));
}

void test_display(const struct test *t)
{
    printf("Name: %s\n", t->name);
    printf("Address: %s\n", t->address);
}

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_3
MD5: 32652eb76ad92212609680d64e5687d3

Next, we'll create our bindings:

    gcc -c -fdump-ada-spec -C ./test.h

This creates the following specification in test_h.ads:

Listing 23: test_h.ads

pragma Ada_2005;
pragma Style_Checks (Off);
with Interfaces.C; use Interfaces.C;
with System;
with Interfaces.C.Strings;

(continues on next page)
As we can see, the binding generator completely ignores the declaration `struct test` and all references to the test struct are replaced by addresses (System.Address). Nevertheless, these bindings are good enough to allow us to create a test application in Ada:

Listing 24: show_automatic_c_struct_bindings.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Interfaces.C;
use Interfaces.C;
with Interfaces.C.Strings;
use Interfaces.C.Strings;
with test_h; use test_h;
with System;

procedure Show_Automatic_C_Struct_Bindings is
  Name : constant chars_ptr :=
    New_String ("John Doe");
  Address : constant chars_ptr :=
    New_String ("Small Town");
  T : System.Address := test_create;
begin
  test_reset (T);
```

(continues on next page)
We can successfully bind our C code with Ada using the automatically-generated bindings, but they aren't ideal. Instead, we would prefer Ada bindings that match our (human) interpretation of the C header file. This requires manual analysis of the header file. The good news is that we can use the automatic generated bindings as a starting point and adapt them to our needs. For example, we can:

1. Define a Test type based on System.Address and use it in all relevant functions.
2. Remove the test_ prefix in all operations on the Test type.

This is the resulting specification:

```
with System;
with Interfaces.C; use Interfaces.C;
with Interfaces.C.Strings;

package adapted_test_h is
  type Test is new System.Address;

  function Create return Test;
  pragma Import (C, Create, "test_create");

  procedure Destroy (T : Test);
  pragma Import (C, Destroy, "test_destroy");

  procedure Reset (T : Test);
  pragma Import (C, Reset, "test_reset");

  procedure Set_Name (T : Test;
                       Name : Interfaces.C.Strings.chars_ptr);
  pragma Import (C, Set_Name, "test_set_name");

  procedure Set_Address (T : Test;
                        Address : Interfaces.C.Strings.chars_ptr);
  pragma Import (C, Set_Address, "test_set_address");

  procedure Display (T : Test);
  pragma Import (C, Display, "test_display");

end adapted_test_h;
```

And this is the corresponding Ada body:
Learning Ada

Listing 26: show_adapted_c_struct_bindings.adb

```ada
with Interfaces.C;
use Interfaces.C;

with Interfaces.C.Strings;
use Interfaces.C.Strings;

with adapted_test_h; use adapted_test_h;

with System;

procedure Show_Adapted_C_Struct_Bindings is
  Name : constant chars_ptr := New_String ("John Doe");
  Address : constant chars_ptr := New_String ("Small Town");
  T : Test := Create;

begin
  Reset (T);
  Set_Name (T, Name);
  Set_Address (T, Address);
  Display (T);
  Destroy (T);
end Show_Adapted_C_Struct_Bindings;
```

Now we can use the Test type and its operations in a clean, readable way.
CHAPTER
EIGHTEEN

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a large and ill-defined concept in programming languages and one that tends to encompass many different meanings because different languages often implement their own vision of it, with similarities and differences from the implementations in other languages.

However, one model mostly "won" the battle of what object-oriented means, if only by sheer popularity. It's the model used in the Java programming language, which is very similar to the one used by C++. Here are some defining characteristics:

• Type derivation and extension: Most object oriented languages allow the user to add fields to derived types.

• Subtyping: Objects of a type derived from a base type can, in some instances, be substituted for objects of the base type.

• Runtime polymorphism: Calling a subprogram, usually called a method, attached to an object type can dispatch at runtime depending on the exact type of the object.

• Encapsulation: Objects can hide some of their data.

• Extensibility: People from the "outside" of your package, or even your whole library, can derive from your object types and define their own behaviors.

Ada dates from before object-oriented programming was as popular as it is today. Some of the mechanisms and concepts from the above list were in the earliest version of Ada even before what we would call OOP was added:

• As we saw, encapsulation is not implemented at the type level in Ada, but instead at the package level.

• Subtyping can be implemented using, well, subtypes, which have a full and permissive static substitutability model. The substitution will fail at runtime if the dynamic constraints of the subtype are not fulfilled.

• Runtime polymorphism can be implemented using variant records.

However, this lists leaves out type extensions, if you don't consider variant records, and extensibility.

The 1995 revision of Ada added a feature filling the gaps, which allowed people to program following the object-oriented paradigm in an easier fashion. This feature is called tagged types.

Note: It's possible to program in Ada without ever creating tagged types. If that's your preferred style of programming or you have no specific use for tagged types, feel free to not use them, as is the case for many features of Ada.

However, they can be the best way to express solutions to certain problems and they may be the best way to solve your problem. If that's the case, read on!
18.1 Derived types

Before presenting tagged types, we should discuss a topic we have brushed on, but not really covered, up to now:

You can create one or more new types from every type in Ada. Type derivation is built into the language.

Listing 1: newtypes.ads

```
package Newtypes is
  type Point is record
    X, Y : Integer;
  end record;
  type New_Point is new Point;
end Newtypes;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Newtypes
MD5: 0d45096755b4bfb08ba8db19ecba3f57

Type derivation is useful to enforce strong typing because the type system treats the two types as incompatible.

But the benefits are not limited to that: you can inherit things from the type you derive from. You not only inherit the representation of the data, but you can also inherit behavior.

When you inherit a type you also inherit what are called primitive operations. A primitive operation (or just a primitive) is a subprogram attached to a type. Ada defines primitives as subprograms defined in the same scope as the type.

Attention: A subprogram will only become a primitive of the type if:
1. The subprogram is declared in the same scope as the type and
2. The type and the subprogram are declared in a package

Listing 2: primitives.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Primitives is
  package Week is
    type Days is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
    -- Print_Day is a primitive
    -- of the type Days
    procedure Print_Day (D : Days);
  end Week;

  package body Week is
    procedure Print_Day (D : Days) is
      begin
        Put_Line (Days'Image (D));
      end Print_Day;
  end Week;
```

(continues on next page)
use Week;
type Weekend_Days is new
   Days range Saturday .. Sunday;

-- A procedure Print_Day is automatically
-- inherited here. It is as if the procedure
-- has been declared with the same body
begin
   Sat : Weekend_Days := Saturday;
   Print_Day (Sat);
end Primitives;

This kind of inheritance can be very useful, and is not limited to record types (you can use it on discrete types, as in the example above), but it's only superficially similar to object-oriented inheritance:

- Records can't be extended using this mechanism alone. You also can't specify a new representation for the new type: it will always have the same representation as the base type.
- There's no facility for dynamic dispatch or polymorphism. Objects are of a fixed, static type.

There are other differences, but it's not useful to list them all here. Just remember that this is a kind of inheritance you can use if you only want to statically inherit behavior without duplicating code or using composition, but a kind you can't use if you want any dynamic features that are usually associated with OOP.

18.2 Tagged types

The 1995 revision of the Ada language introduced tagged types to fulfill the need for an unified solution that allows programming in an object-oriented style similar to the one described at the beginning of this chapter.

Tagged types are very similar to normal records except that some functionality is added:

- Types have a tag, stored inside each object, that identifies the runtime type\(^{20}\) of that object.
- Primitives can dispatch. A primitive on a tagged type is what you would call a method in Java or C++. If you derive a base type and override a primitive of it, you can often call it on an object with the result that which primitive is called depends on the exact runtime type of the object.

\(^{20}\) https://en.wikipedia.org/wiki/Run-time_type_information
Learning Ada

- Subtyping rules are introduced allowing a tagged type derived from a base type to be statically compatible with the base type.

Let's see our first tagged type declarations:

Listing 3: p.ads

```ada
package P is
  type My_Class is tagged null record;
  -- Just like a regular record, but
  -- with tagged qualifier
  -- Methods are outside of the type
  -- definition:
  procedure Foo (Self : in out My_Class);
  -- If you define a procedure taking a
  -- My_Class argument in the same package,
  -- it will be a method.
  -- Here's how you derive a tagged type:
  type Derived is new My_Class with record
    A : Integer;
    -- You can add fields in derived types.
  end record;
  overriding
  procedure Foo (Self : in out Derived);
  -- The "overriding" qualifier is optional,
  -- but if it is present, it must be valid.
end P;
```

Listing 4: p.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body P is
  procedure Foo (Self : in out My_Class) is
    begin
      Put_Line ("In My_Class.Foo");
      end Foo;
  procedure Foo (Self : in out Derived) is
    begin
      Put_Line ("In Derived.Foo, A = " & Integer'Image (Self.A));
    end Foo;
end P;
```
18.3 Classwide types

To remain consistent with the rest of the language, a new notation needed to be introduced to say "This object is of this type or any descendant derives tagged type".

In Ada, we call this the classwide type. It's used in OOP as soon as you need polymorphism. For example, you can't do the following:

Listing 5: main.adb

```ada
with P; use P;

procedure Main is
  01 : My_Class;
        -- Declaring an object of type My_Class

  02 : Derived := (A => 12);
        -- Declaring an object of type Derived

  03 : My_Class := 02;
        -- INVALID: Trying to assign a value
        -- of type derived to a variable of
        -- type My_Class.
begin
  null;
end Main;
```

This is because an object of a type T is exactly of the type T, whether T is tagged or not. What you want to say as a programmer is "I want O3 to be able to hold an object of type My_Class or any type descending from My_Class". Here's how you do that:

Listing 6: main.adb

```ada
with P; use P;

procedure Main is
  01 : My_Class;
        -- Declare an object of type My_Class

  02 : Derived := (A => 12);
        -- Declare an object of type Derived

  03 : My_Class'Class := 02;
        -- Now valid: My_Class'Class designates
        -- the classwide type for My_Class,
        -- which is the set of all types
        -- descending from My_Class (including
        -- My_Class).
begin
```

(continues on next page)
Attention: Because an object of a classwide type can be the size of any descendant of its base type, it has an unknown size. It's therefore an indefinite type, with the expected restrictions:

- It can't be stored as a field/component of a record
- An object of a classwide type needs to be initialized immediately (you can't specify the constraints of such a type in any way other than by initializing it).

### 18.4 Dispatching operations

We saw that you can override operations in types derived from another tagged type. The eventual goal of OOP is to make a dispatching call: a call to a primitive (method) that depends on the exact type of the object.

But, if you think carefully about it, a variable of type My_Class always contains an object of exactly that type. If you want to have a variable that can contain a My_Class or any derived type, it has to be of type My_Class'Class.

In other words, to make a dispatching call, you must first have an object that can be either of a type or any type derived from this type, namely an object of a classwide type.

Listing 7: main.adb

```ada
with P; use P;

procedure Main is
  01 : My_Class;
    -- Declare an object of type My_Class
  02 : Derived := (A => 12);
    -- Declare an object of type Derived
  03 : My_Class'Class := 02;
  04 : My_Class'Class := 01;

begin
  Foo (01);
    -- Non dispatching: Calls My_Class.Foo
  Foo (02);
    -- Non dispatching: Calls Derived.Foo
  Foo (03);
    -- Dispatching: Calls Derived.Foo
  Foo (04);
    -- Dispatching: Calls My_Class.Foo
end Main;
```
Attention

You can convert an object of type Derived to an object of type My_Class. This is called a view conversion in Ada parlance and is useful, for example, if you want to call a parent method.

In that case, the object really is converted to a My_Class object, which means its tag is changed. Since tagged objects are always passed by reference, you can use this kind of conversion to modify the state of an object: changes to converted object will affect the original one.

Listing 8: main.adb

```ada
with P; use P;

procedure Main is
  O1 : Derived := (A => 12);
  -- Declare an object of type Derived
  O2 : My_Class := My_Class (O1);
  O3 : My_Class'Class := O2;
begin
  Foo (O1);
  -- Non dispatching: Calls Derived.Foo
  Foo (O2);
  -- Non dispatching: Calls My_Class.Foo
  Foo (O3);
  -- Dispatching: Calls My_Class.Foo
end Main;
```

18.4. Dispatching operations
18.5 Dot notation

You can also call primitives of tagged types with a notation that's more familiar to object oriented programmers. Given the Foo primitive above, you can also write the above program this way:

Listing 9: main.adb

```ada
with P; use P;

procedure Main is
  O1 : My_Class;
  -- Declare an object of type My_Class
  02 : Derived := (A => 12);
  -- Declare an object of type Derived
  O3 : My_Class'Class := 02;
  O4 : My_Class'Class := 01;
begin
  O1.Foo;
  -- Non dispatching: Calls My_Class.Foo
  O2.Foo;
  -- Non dispatching: Calls Derived.Foo
  O3.Foo;
  -- Dispatching: Calls Derived.Foo
  O4.Foo;
  -- Dispatching: Calls My_Class.Foo
end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
MD5: 9c6ebdfec9ceeb986d92eb90ec9ff59b

Runtime output

In My_Class.Foo
In Derived.Foo, A = 12
In Derived.Foo, A = 12
In My_Class.Foo

If the dispatching parameter of a primitive is the first parameter, which is the case in our examples, you can call the primitive using the dot notation. Any remaining parameter are passed normally:

Listing 10: main.adb

```ada
with P; use P;

procedure Main is
  package Extend is
    type D2 is new Derived with null record;
    procedure Bar (Self : in out D2;
                    Val : Integer);
  end Extend;

  package body Extend is
    procedure Bar (Self : in out D2;
```
18.6 Private & Limited

We've seen previously (in the Privacy (page 113) chapter) that types can be declared limited or private. These encapsulation techniques can also be applied to tagged types, as we'll see in this section.

This is an example of a tagged private type:

```
package P is
  type T is tagged private;
private
  type T is tagged record
    E : Integer;
  end record;
end P;
```

This is an example of a tagged limited type:

```
package P is
  type T is tagged limited record
    E : Integer;
  end record;
end P;
```
Naturally, you can combine both *limited* and *private* types and declare a tagged limited private type:

### Listing 13: p.ads

```ada
class package P is
   type T is tagged limited private;
   procedure Init (A : in out T);
private
   type T is tagged limited record
      E : Integer;
   end record;
end P;
```

### Listing 14: p.adb

```ada
class package body P is
   procedure Init (A : in out T) is
   begin
      A.E := 0;
   end Init;
end P;
```

### Listing 15: main.adb

```ada
with P; use P;

class procedure Main is
   T1, T2 : T;
begin
   T1.Init;
   T2.Init;
   -- The following line doesn't work
   -- because type T is private:
   -- T1.E := 0;
   -- The following line doesn't work
   -- because type T is limited:
   -- T2 := T1;
end Main;
```

Note that the code in the Main procedure above presents two assignments that trigger compilation errors because type T is limited private. In fact, you cannot:

- assign to T1.E directly because type T is private;
- assign T1 to T2 because type T is limited.
In this case, there's no distinction between tagged and non-tagged types: these compilation errors would also occur for non-tagged types.

### 18.7 Classwide access types

In this section, we'll discuss an useful pattern for object-oriented programming in Ada: class-wide access type. Let's start with an example where we declare a tagged type T and a derived type T_New:

**Listing 16: p.ads**

```ada
package P is
  type T is tagged null record;
  procedure Show (Dummy : T);
  type T_New is new T with null record;
  procedure Show (Dummy : T_New);
end P;
```

**Listing 17: p.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body P is
  procedure Show (Dummy : T) is
    Put_Line ("Using type ", T'External_Tag);
  end Show;

  procedure Show (Dummy : T_New) is
    Put_Line ("Using type ", T_New'External_Tag);
  end Show;
end P;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Classwide_Error
MD5: fd5cb99925d3c88536546aa0be8104b7

Note that we're using null records for both types T and T_New. Although these types don't actually have any component, we can still use them to demonstrate dispatching. Also note that the example above makes use of the 'External_Tag attribute in the implementation of the Show procedure to get a string for the corresponding tagged type.

As we've seen before, we must use a classwide type to create objects that can make dispatching calls. In other words, objects of type T'Class will dispatch. For example:

**Listing 18: dispatching_example.adb**

```ada
with P; use P;
procedure Dispatching_Example is
```

(continues on next page)
Learning Ada

T2 : T_New;
T_Dispatch : constant T'Class := T2;
begin
T_Dispatch.Show;
end Dispatching_Example;

Code block metadata
Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Classwide_Error
MD5: f8957b31c9c62db23759baad7b867a57

Runtime output
Using type P.T_NEW

A more useful application is to declare an array of objects that can dispatch. For example, we'd like to declare an array T_Arr, loop over this array and dispatch according to the actual type of each individual element:

for I in T_Arr'Range loop
T_Arr (I).Show;
-- Call Show procedure according
-- to actual type of T_Arr (I)
end loop;

However, it's not possible to declare an array of type T'Class directly:

Listing 19: classwide_compilation_error.adb

with P; use P;
procedure ClasswideCompilation_Error is
T_Arr : array (1 .. 2) of T'Class;
--
--  Compilation Error!
begin
for I in T_Arr'Range loop
T_Arr (I).Show;
end loop;
end ClasswideCompilation_Error;

Code block metadata
Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Classwide_Error
MD5: e86f6c6ee35dced8f338bf6177d178fd

Build output
classwide_compilation_error.adb:4:32: error: unconstrained element type in array
..declaration
gprbuild: *** compilation phase failed

In fact, it's impossible for the compiler to know which type would actually be used for each element of the array. However, if we use dynamic allocation via access types, we can allocate objects of different types for the individual elements of an array T_Arr. We do this by using classwide access types, which have the following format:

type T_Class is access T'Class;

We can rewrite the previous example using the T_Class type. In this case, dynamically allocated objects of this type will dispatch according to the actual type used during the
allocation. Also, let’s introduce an Init procedure that won’t be overridden for the derived T_New type. This is the adapted code:

Listing 20: p.ads

```ada
package P is
  type T is tagged record
    E : Integer;
  end record;
  type T_Class is access T'Class;
  procedure Init (A : in out T);
  procedure Show (Dummy : T);
  type T_New is new T with null record;
  procedure Show (Dummy : T_New);
end P;
```

Listing 21: p.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body P is
  procedure Init (A : in out T) is
    begin
      Put_Line ("Initializing type T...");
      A.E := 0;
    end Init;
  procedure Show (Dummy : T) is
    begin
      Put_Line ("Using type " & T'External_Tag);
    end Show;
  procedure Show (Dummy : T_New) is
    begin
      Put_Line ("Using type " & T_New'External_Tag);
    end Show;
end P;
```

Listing 22: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with P; use P;
procedure Main is
  T_Arr : array (1 .. 2) of T_Class;
begin
  T_Arr (1) := new T;
  T_Arr (2) := new T_New;
  for I in T_Arr'Range loop
    Put_Line ("Element # " & Integer'Image (I));
  end loop;
```

(continues on next page)
In this example, the first element (T_Arr (1)) is of type T, while the second element is of type T_New. When running the example, the Init procedure of type T is called for both elements of the T_Arr array, while the call to the Show procedure selects the corresponding procedure according to the type of each element of T_Arr.
In previous chapters, we’ve used arrays as the standard way to group multiple objects of a specific data type. In many cases, arrays are good enough for manipulating those objects. However, there are situations that require more flexibility and more advanced operations. For those cases, Ada provides support for containers — such as vectors and sets — in its standard library.

We present an introduction to containers here. For a list of all containers available in Ada, see Appendix B (page 271).

19.1 Vectors

In the following sections, we present a general overview of vectors, including instantiation, initialization, and operations on vector elements and vectors.

19.1.1 Instantiation

Here’s an example showing the instantiation and declaration of a vector V:

Listing 1: show_vector_inst.adb

```ada
with Ada.Containers.Vectors;

procedure Show_Vector_Inst is
  package Integer_Vectors is new
    Ada.Containers.Vectors
    (Index_Type => Natural,
     Element_Type => Integer);

  V : Integer_Vectors.Vector;

begin
  null;
end Show_Vector_Inst;
```

Containers are based on generic packages, so we can’t simply declare a vector as we would declare an array of a specific type:

```ada
A : array (1 .. 10) of Integer;
```
Learning Ada

Instead, we first need to instantiate one of those packages. We with the container package (Ada.Containers.Vectors in this case) and instantiate it to create an instance of the generic package for the desired type. Only then can we declare the vector using the type from the instantiated package. This instantiation needs to be done for any container type from the standard library.

In the instantiation of Integer_Vectors, we indicate that the vector contains elements of Integer type by specifying it as the Element_Type. By setting Index_Type to Natural, we specify that the allowed range includes all natural numbers. We could have used a more restrictive range if desired.

19.1.2 Initialization

One way to initialize a vector is from a concatenation of elements. We use the & operator, as shown in the following example:

Listing 2: show_vector_init.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Init is
  package Integer_Vectors is new
    Ada.Containers.Vectors
      (Index_Type => Natural,
       Element_Type => Integer);
  use Integer_Vectors;

  V : Vector := 20 & 10 & 0 & 13;
begin
  Put_Line ("Vector has "
            & Count_Type'Image (V.Length)
            & " elements");
end Show_Vector_Init;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Init
MD5: 0087b0a15e0c88b27ac36c3b27159a17

Runtime output

Vector has 4 elements

We specify use Integer_Vectors, so we have direct access to the types and operations from the instantiated package. Also, the example introduces another operation on the vector: Length, which retrieves the number of elements in the vector. We can use the dot notation because Vector is a tagged type, allowing us to write either V.Length or Length (V).
19.1.3 Appending and prepending elements

You add elements to a vector using the Prepend and Append operations. As the names suggest, these operations add elements to the beginning or end of a vector, respectively. For example:

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Append is

   package Integer_Vectors is new
      Ada.Containers.Vectors
      (Index_Type => Natural,
       Element_Type => Integer);
   use Integer_Vectors;

   V : Vector;

begin
   Put_Line ("Appending some elements "
             & "to the vector...");
   V.Append (20);
   V.Append (10);
   V.Append (0);
   V.Append (13);
   Put_Line ("Finished appending.");

   Put_Line ("Prepending some elements"
             & "to the vector...");
   V.Prepend (30);
   V.Prepend (40);
   V.Prepend (100);
   Put_Line ("Finished prepending.");

   Put_Line ("Vector has "
             & Count_Type'Image (V.Length)
             & " elements");
end Show_Vector_Append;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Append
MD5: f88d393ba96a7950f58d9f1c0c74a021

Runtime output

Appending some elements to the vector...
Finished appending.
Prepending some elements to the vector...
Finished prepending.
Vector has 7 elements

This example puts elements into the vector in the following sequence: (100, 40, 30, 20, 10, 0, 13).

The Reference Manual specifies that the worst-case complexity must be:

- $O(\log N)$ for the Append operation, and
Learning Ada

- O(N log N) for the Prepend operation.

19.1.4 Accessing first and last elements

We access the first and last elements of a vector using the First_Element and Last_Element functions. For example:

Listing 4: show_vector_first_last_element.adb

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_First_Last_Element is

package Integer_Vectors is new
Ada.Containers.Vectors
(Index_Type  => Natural,
 Element_Type => Integer);

use Integer_Vectors;

function Img (I : Integer)  return String
renames Integer'Image;
function Img (I : Count_Type) return String
renames Count_Type'Image;

V : Vector := 20 & 10 & 0 & 13;
begin
Put_Line ("Vector has 
 & Img (V.Length)
 & " elements");

-- Using V.First_Element to
-- retrieve first element
Put_Line ("First element is 
 & Img (V.First_Element));

-- Using V.Last_Element to
-- retrieve last element
Put_Line ("Last element is 
 & Img (V.Last_Element));
end Show_Vector_First_Last_Element;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_First_Last_Element
MD5: 6022557660d0017ced6b4115c845cd48d

Runtime output

Vector has 4 elements
First element is 20
Last element is 13

You can swap elements by calling the procedure Swap and retrieving a reference (a cursor) to the first and last elements of the vector by calling First and Last. A cursor allows us to iterate over a container and process individual elements from it.

With these operations, we're able to write code to swap the first and last elements of a vector:
Listing 5: show_vector_first_last_element.adb

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_First_Last_Element is

package Integer_Vectors is new
  Ada.Containers.Vectors
  (Index_Type    => Natural,
   Element_Type  => Integer);
use Integer_Vectors;

function Img (I : Integer) return String renames Integer'Image;

V : Vector := 20 & 10 & 0 & 13;
begn
  -- We use V.First and V.Last to retrieve
  -- cursor for first and last elements.
  -- We use V.Swap to swap elements.
  V.Swap (V.First, V.Last);
  Put_Line ("First element is now ",
             & Img (V.First_Element));
  Put_Line ("Last element is now ",
             & Img (V.Last_Element));
end Show_Vector_First_Last_Element;
```

Runtime output

First element is now 13
Last element is now 20

19.1.5 Iterating

The easiest way to iterate over a container is to use a `for E of` Our_Container loop. This gives us a reference (E) to the element at the current position. We can then use E directly. For example:

Listing 6: show_vector_iteration.adb

```ada
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Iteration is

package Integer_Vectors is new
  Ada.Containers.Vectors
  (Index_Type    => Natural,
   Element_Type  => Integer);
```

(continues on next page)
Element_Type => Integer);  
use Integer_Vectors;  
function Img (I : Integer) return String  
renames Integer’Image;  
V : Vector := 20 & 10 & 0 & 13;  
begin  
  Put_Line ("Vector elements are: ");  
  for E of V loop  
    Put_Line ("- " & Img (E));  
  end loop;  
end Show_Vector_Iteration;

Code block metadata
Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Iteration  
MD5: 4fc9a939aa822097d3a937646d3e2910

Runtime output
Vector elements are:  
- 20  
- 10  
- 0  
- 13  

This code displays each element from the vector V.

Because we’re given a reference, we can display not only the value of an element but also modify it. For example, we could easily write a loop to add one to each element of vector V:

for E of V loop  
  E := E + 1;  
end loop;  

We can also use indices to access vector elements. The format is similar to a loop over array elements: we use a for I in <range> loop. The range is provided by V.First_Index and V.Last_Index. We can access the current element by using it as an array index: V (I). For example:

Listing 7: show_vector_index_iteration.adb

with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Vector_Index_Iteration is  
  package Integer_Vectors is new  
    Ada.Containers.Vectors  
      (Index_Type => Natural,  
       Element_Type => Integer);
use Integer_Vectors;

V : Vector := 20 & 10 & 0 & 13;
begin
  Put_Line ("Vector elements are: ");

-- Using indices in a "for I in ..." loop to iterate:
for I in V.First_Index .. V.Last_Index loop
  Put ("- [ ",
       & Extended_Index'Image (I)
       & "] ");
  Put (Integer'Image (V (I))); New_Line;
end loop;
end Show_Vector_Index_Iteration;

Here, in addition to displaying the vector elements, we’re also displaying each index, I, just like what we can do for array indices. Also, we can access the element by using either the short form V (I) or the longer form V.Element (I) but not V.I.

As mentioned in the previous section, you can use cursors to iterate over containers. For this, use the function Iterate, which retrieves a cursor for each position in the vector. The corresponding loop has the format for C in V.Iterate loop. Like the previous example using indices, you can again access the current element by using the cursor as an array index: V (C). For example:
Instead of accessing an element in the loop using V (C), we could also have used the longer form Element (C). In this example, we’re using the function To_Index to retrieve the index corresponding to the current cursor.

As shown in the comments after the loop, we could also use a while ... loop to iterate over the vector. In this case, we would start with a cursor for the first element (retrieved by calling V.First) and then call Next (C) to retrieve a cursor for subsequent elements.
Next (C) returns No_Element when the cursor reaches the end of the vector.

You can directly modify the elements using a reference. This is what it looks like when using both indices and cursors:

```ada
-- Modify vector elements using index
for I in V.First_Index .. V.Last_Index loop
    V (I) := V (I) + 1;
end loop;

-- Modify vector elements using cursor
for C in V.Iterate loop
    V (C) := V (C) + 1;
end loop;
```

The Reference Manual requires that the worst-case complexity for accessing an element be $O(\log N)$.

Another way of modifying elements of a vector is using a process procedure, which takes an individual element and does some processing on it. You can call Update_Element and pass both a cursor and an access to the process procedure. For example:

```ada
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Update is
    package Integer_Vectors is new
        Ada.Containers.Vectors
            (Index_Type  => Natural,
             Element_Type => Integer);
    use Integer_Vectors;

    procedure Add_One (I : in out Integer) is
    begin
        I := I + 1;
    end Add_One;

    V : Vector := 20 & 10 & 12;
    begin
        -- Use V.Update_Element to process elements
        for C in V.Iterate loop
            V.Update_Element (C, Add_One'Access);
        end loop;
    end Show_Vector_Update;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Update
MD5: 5dcc3dd8020632a8ea2ce975ecd8f4da
19.1.6 Finding and changing elements

You can locate a specific element in a vector by retrieving its index. Find_Index retrieves the index of the first element matching the value you're looking for. Alternatively, you can use Find to retrieve a cursor referencing that element. For example:

Listing 10: show_find_vector_element.adb

```ada
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Find_Vector_Element is
  package Integer_Vectors is new Ada.Containers.Vectors
    (Index_Type => Natural,
     Element_Type => Integer);
  use Integer_Vectors;

  V : Vector := 20 & 10 & 0 & 13;
  Idx : Extended_Index;
  C : Cursor;

begin
  -- Using Find_Index to retrieve the index
  -- of element with value 10
  Idx := V.Find_Index (10);
  Put_Line ("Index of element with value 10 is "
            & Extended_Index'Image (Idx));

  -- Using Find to retrieve the cursor for
  -- the element with value 13
  C := V.Find (13);
  Idx := To_Index (C);
  Put_Line ("Index of element with value 13 is "
            & Extended_Index'Image (Idx));
end Show_Find_Vector_Element;
```

As we saw in the previous section, we can directly access vector elements by using either an index or cursor. However, an exception is raised if we try to access an element with an invalid index or cursor, so we must check whether the index or cursor is valid before using it to access an element. In our example, Find_Index or Find might not have found the element in the vector. We check for this possibility by comparing the index to No_Index or the cursor to No_Element. For example:

```ada
-- Modify vector element using index
if Idx /= No_Index then
  V (Idx) := 11;
end if;
```

(continues on next page)
Instead of writing \( V(C) := 14 \), we could use the longer form \( V.Replace\_Element(C, 14) \).

### 19.1.7 Inserting elements

In the previous sections, we’ve seen examples of how to add elements to a vector:

- using the concatenation operator (\&\) at the vector declaration, or
- calling the Prepend and Append procedures.

You may want to insert an element at a specific position, e.g. before a certain element in the vector. You do this by calling Insert. For example:

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Insert is
  package Integer_Vectors is new
    Ada.Containers.Vectors
      (Index_Type => Natural,
       Element_Type => Integer);
  use Integer_Vectors;

  procedure Show_Elements (V : Vector) is
  begin
    New_Line;
    Put_Line ("Vector has 
      & Count_Type'Image (V.Length)
      & " elements");
    if not V.Is_Empty then
      Put_Line ("Vector elements are: ");
      for E of V loop
        Put_Line ("- " & Integer'Image (E));
      end loop;
    end if;
    Show_Elements;
  end Show_Elements;

  V : Vector := 20 & 10 & 12;
  C : Cursor;
  begin
    Show_Elements (V);
    New_Line;
    Put_Line ("Adding element with value 9");
    Put_Line (" (before 10)...");
  end Show_Vector_Insert;
end Show_Vector_Insert;
```

(continues on next page)
-- Using V.Insert to insert the element 
-- into the vector

C := V.Find (10);
if C /= No_Element then
  V.Insert (C, 9);
end if;

Show_Elements (V);
end Show_Vector_Insert;

Code block metadata

Project: Courses.Intro To Ada.Standard_Library.Show_Vector_Insert
MD5: af49f39038896c51ab97541036fbcaf

Runtime output

Vector has 3 elements
Vector elements are:
- 20
- 10
- 12

Adding element with value 9
  (before 10)...

Vector has 4 elements
Vector elements are:
- 20
- 9
- 10
- 12

In this example, we're looking for an element with the value of 10. If we find it, we insert an element with the value of 9 before it.

19.1.8 Removing elements

You can remove elements from a vector by passing either a valid index or cursor to the Delete procedure. If we combine this with the functions Find_Index and Find from the previous section, we can write a program that searches for a specific element and deletes it, if found:

Listing 12: show_remove_vector_element.adb

with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Remove_Vector_Element is
  package Integer_Vectors is new
    Ada.Containers.Vectors
      (Index_Type => Natural,
       Element_Type => Integer);
  use Integer_Vectors;
V : Vector := 20 & 10 & 0 & 13 & 10 & 13;
Idx : Extended_Index;
C : Cursor;
begin
-- Use Find_Index to retrieve index of
-- the element with value 10
Idx := V.Find_Index (10);
-- Checking whether index is valid
if Idx /= No_Index then
-- Removing element using V.Delete
V.Delete (Idx);
end if;
-- Use Find to retrieve cursor for
-- the element with value 13
C := V.Find (13);
-- Check whether index is valid
if C /= No_Element then
-- Remove element using V.Delete
V.Delete (C);
end if;
end Show_Remove_Vector_Element;

Code block metadata
Project: Courses.Intro_To_Ada.Standard_Library.Show_Remove_Vector_Element
MD5: 540d0dc5715e58926e9dc4600bd6ad5d

We can extend this approach to delete all elements matching a certain value. We just need to keep searching for the element in a loop until we get an invalid index or cursor. For example:

Listing 13: show_remove_vector_elements.adb

with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Remove_Vector_Elements is
package Integer_Vectors is new
Ada.Containers.Vectors
(Index_Type => Natural,
Element_Type => Integer);
use Integer_Vectors;
procedure Show_Elements (V : Vector) is
begin
New_Line;
Put_Line ("Vector has 
& Count_Type'Image (V.Length)
& " elements");
if not V.Is_Empty then
Put_Line ("Vector elements are: ");
for E of V loop
  Put_Line ("- " & Integer’Image (E));
end loop;
end if;
end Show_Elements;

V : Vector := 20 & 10 & 0 & 13 & 10 & 14 & 13;
beg
  Show_Elements (V);
  --
  -- Remove elements using an index
  --
  declare
    E : constant Integer := 10;
    I : Extended_Index;
  begin
    New_Line;
    Put_Line ("Removing all elements with value of " & Integer’Image (E) & "...");
    loop
      I := V.Find_Index (E);
      exit when I = No_Index;
      V.Delete (I);
    end loop;
  end;
  --
  -- Remove elements using a cursor
  --
  declare
    E : constant Integer := 13;
    C : Cursor;
  begin
    New_Line;
    Put_Line ("Removing all elements with value of " & Integer’Image (E) & "...");
    loop
      C := V.Find (E);
      exit when C = No_Element;
      V.Delete (C);
    end loop;
  end;
  Show_Elements (V);
end Show_Remove_Vector_Elements;
Removing all elements with value of 10...
Removing all elements with value of 13...
Vector has 3 elements
Vector elements are:
- 20
- 0
- 14

In this example, we remove all elements with the value 10 from the vector by retrieving their index. Likewise, we remove all elements with the value 13 by retrieving their cursor.

19.1.9 Other Operations

We've seen some operations on vector elements. Here, we'll see operations on the vector as a whole. The most prominent is the concatenation of multiple vectors, but we'll also see operations on vectors, such as sorting and sorted merging operations, that view the vector as a sequence of elements and operate on the vector considering the element's relations to each other.

We do vector concatenation using the & operator on vectors. Let's consider two vectors V1 and V2. We can concatenate them by doing V := V1 & V2. V contains the resulting vector.

The generic package Generic_Sorting is a child package of Ada.Containers.Vectors. It contains sorting and merging operations. Because it's a generic package, you can't use it directly, but have to instantiate it. In order to use these operations on a vector of integer values (Integer_Vectors, in our example), you need to instantiate it directly as a child of Integer_Vectors. The next example makes it clear how to do this.

After instantiating Generic_Sorting, we make all the operations available to us with the use statement. We can then call Sort to sort the vector and Merge to merge one vector into another.

The following example presents code that manipulates three vectors (V1, V2, V3) using the concatenation, sorting and merging operations:

```ada
with Ada.Containers; use Ada.Containers;

procedure Show_Vector_Ops is

   package Integer_Vectors is new
      Ada.Containers.Vectors
      (Index_Type => Natural,
       Element_Type => Integer);

   package Integer_Vectors_Sorting is
      new Integer_Vectors.Generic_Sorting;
```

(continues on next page)
use Integer_Vectors;
use Integer_Vectors_Sorting;

procedure Show_Elements (V : Vector) is
begin
  New_Line;
  Put_Line ("Vector has 
    & Count_Type'Image (V.Length)
    & " elements");

  if not V.Is_Empty then
    Put_Line ("Vector elements are: ");
    for E of V loop
      Put_Line (- " & Integer'Image (E));
    end loop;
  end if;
end Show_Elements;

V, V1, V2, V3 : Vector;
begin
  V1 := 10 & 12 & 18;
  V2 := 11 & 13 & 19;
  V3 := 15 & 19;

  New_Line;
  Put_Line ("---- V1 ----");
  Show_Elements (V1);

  New_Line;
  Put_Line ("---- V2 ----");
  Show_Elements (V2);

  New_Line;
  Put_Line ("---- V3 ----");
  Show_Elements (V3);

  New_Line;
  Put_Line ("Concatenating V1, V2 and V3 into V:");
  V := V1 & V2 & V3;
  Show_Elements (V);

  New_Line;
  Put_Line ("Sorting V:");
  Sort (V);
  Show_Elements (V);

  New_Line;
  Put_Line ("Merging V2 into V1:");
  Merge (V1, V2);
  Show_Elements (V1);
end Show_Vector_Ops;
Runtime output

---- V1 ----
Vector has 3 elements
Vector elements are:
- 10
- 12
- 18

---- V2 ----
Vector has 3 elements
Vector elements are:
- 11
- 13
- 19

---- V3 ----
Vector has 2 elements
Vector elements are:
- 15
- 19

Concatenating V1, V2 and V3 into V:
Vector has 8 elements
Vector elements are:
- 10
- 11
- 12
- 13
- 15
- 18
- 19
- 19

Sorting V:
Vector has 8 elements
Vector elements are:
- 10
- 11
- 12
- 13
- 15
- 18
- 19
- 19

Merging V2 into V1:
Vector has 6 elements
Vector elements are:
- 10
- 11
- 12

(continues on next page)
The Reference Manual requires that the worst-case complexity of a call to \textit{Sort} be $O(N^2)$ and the average complexity be better than $O(N^2)$.

## 19.2 Sets

Sets are another class of containers. While vectors allow duplicated elements to be inserted, sets ensure that no duplicated elements exist.

In the following sections, we'll see operations you can perform on sets. However, since many of the operations on vectors are similar to the ones used for sets, we'll cover them more quickly here. Please refer back to the section on vectors for a more detailed discussion.

### 19.2.1 Initialization and iteration

To initialize a set, you can call the \textit{Insert} procedure. However, if you do, you need to ensure no duplicate elements are being inserted: if you try to insert a duplicate, you'll get an exception. If you have less control over the elements to be inserted so that there may be duplicates, you can use another option instead:

- a version of \textit{Insert} that returns a Boolean value indicating whether the insertion was successful;
- the \textit{Include} procedure, which silently ignores any attempt to insert a duplicated element.

To iterate over a set, you can use a \textit{for} \textit{E} of \textit{S} loop, as you saw for vectors. This gives you a reference to each element in the set.

Let's see an example:

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Init is
  use Integer_Sets;

  S : Set;
  -- Same as: S : Integer_Sets.Set;
  C : Cursor;
  Ins : Boolean;
begin
  S.Insert (20);
  S.Insert (10);
```

(continues on next page)
S.Insert (0);
S.Insert (13);

-- Calling S.Insert(0) now would raise
-- Constraint_Error because this element
-- is already in the set. We instead call a
-- version of Insert that doesn't raise an
-- exception but instead returns a Boolean
-- indicating the status

S.Insert (0, C, Ins);
if not Ins then
   Put_Line
       ("Error while inserting 0 into set");
end if;

-- We can also call S.Include instead
-- If the element is already present,
-- the set remains unchanged
S.Include (0);
S.Include (13);
S.Include (14);

Put_Line ("Set has "
         & Count_Type'Image (S.Length)
         & " elements");

-- Iterate over set using for .. of loop
Put_Line ("Elements:");
for E of S loop
   Put_Line ("- " & Integer'Image (E));
end loop;
end Show_Set_Init;

---

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Set_Init
MD5: b87f6729feac78396347248b95a30cb6

Runtime output

Error while inserting 0 into set
Set has 5 elements
Elements:
- 0
- 10
- 13
- 14
- 20

---

19.2. Sets
19.2.2 Operations on elements

In this section, we briefly explore the following operations on sets:

- Delete and Exclude to remove elements;
- Contains and Find to verify the existence of elements.

To delete elements, you call the procedure Delete. However, analogously to the Insert procedure above, Delete raises an exception if the element to be deleted isn't present in the set. If you want to permit the case where an element might not exist, you can call Exclude, which silently ignores any attempt to delete a non-existent element.

Contains returns a Boolean value indicating whether a value is contained in the set. Find also looks for an element in a set, but returns a cursor to the element or No_Element if the element doesn't exist. You can use either function to search for elements in a set.

Let's look at an example that makes use of these operations:

Listing 16: show_set_element_ops.adb

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Element_Ops is

    package Integer_Sets is new
        Ada.Containers.Ordered_Sets
        (Element_Type => Integer);

    use Integer_Sets;

    procedure Show_Elements (S : Set) is
        begin
            New_Line;
            Put_Line ("Set has ", & Count_Type'Image (S.Length) & " elements");
            Put_Line ("Elements:");
            for E of S loop
                Put_Line ("- " & Integer'Image (E));
            end loop;
            end Show_Elements;

        S : Set;
        begin
            S.Insert (20);
            S.Insert (10);
            S.Insert (0);
            S.Insert (13);
            S.Delete (13);
            if S.Contains (20) then
                Put_Line ("Found element 20 in set");
            end if;

        end Show_Elements;
```

(continues on next page)
In addition to ordered sets used in the examples above, the standard library also offers hashed sets. The Reference Manual requires the following average complexity of each operation:

<table>
<thead>
<tr>
<th>Operations</th>
<th>Ordered_Sets</th>
<th>Hashed_Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O((\log N)^2)$ or better</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Include</td>
<td>$O((\log N)^2)$ or better</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Replace</td>
<td>$O((\log N)^2)$ or better</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O((\log N)^2)$ or better</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Exclude</td>
<td>$O((\log N)^2)$ or better</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Find</td>
<td>$O((\log N)^2)$ or better</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Subprogram using cursor</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**19.2.3 Other Operations**

The previous sections mostly dealt with operations on individual elements of a set. But Ada also provides typical set operations: union, intersection, difference and symmetric difference. In contrast to some vector operations we've seen before (e.g. Merge), here you can use built-in operators, such as `-`. The following table lists the operations and its associated operator:

<table>
<thead>
<tr>
<th>Set Operation</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td><code>or</code></td>
</tr>
<tr>
<td>Intersection</td>
<td><code>and</code></td>
</tr>
<tr>
<td>Difference</td>
<td><code>-</code></td>
</tr>
<tr>
<td>Symmetric difference</td>
<td><code>xor</code></td>
</tr>
</tbody>
</table>
The following example makes use of these operators:

Listing 17: show_set_ops.adb

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Ops is

package Integer_Sets is new
    Ada.Containers.Ordered_Sets
        (Element_Type => Integer);

use Integer_Sets;

procedure Show_Elements (S : Set) is
begin
    Put_Line ("Elements:");
    for E of S loop
        Put_Line ("- " & Integer'Image (E));
    end loop;
end Show_Elements;

procedure Show_Op (S : Set; Op_Name : String) is
begin
    New_Line;
    Put_Line (Op_Name & "(set #1, set #2) has " & Count_Type'Image (S.Length) & " elements");
end Show_Op;

S1, S2, S3 : Set;
begin
    S1.Insert (0);
    S1.Insert (10);
    S1.Insert (13);
    S2.Insert (0);
    S2.Insert (10);
    S2.Insert (14);
    S3.Insert (0);
    S3.Insert (10);
    New_Line;
    Put_Line ("----- Set #1 -----");
    Show_Elements (S1);
    New_Line;
    Put_Line ("----- Set #2 -----");
    Show_Elements (S2);
    New_Line;
    Put_Line ("----- Set #3 -----");
    Show_Elements (S3);
    New_Line;
    if S3.Is_Subset (S1) then
        Put_Line ("S3 is a subset of S1");
    end if;
```

(continues on next page)
else
   Put_Line ("S3 is not a subset of S1");
end if;

S3 := S1 and S2;
Show_Op (S3, "Intersection");
Show_Elements (S3);

S3 := S1 or S2;
Show_Op (S3, "Union");
Show_Elements (S3);

S3 := S1 - S2;
Show_Op (S3, "Difference");
Show_Elements (S3);

S3 := S1 xor S2;
Show_Op (S3, "Symmetric difference");
Show_Elements (S3);

end Show_Set_Ops;

--- Set #1 ----
Elements:
- 0
- 10
- 13

--- Set #2 ----
Elements:
- 0
- 10
- 14

--- Set #3 ----
Elements:
- 0
- 10

S3 is a subset of S1
Intersection(set #1, set #2) has 2 elements
Elements:
- 0
- 10

Union(set #1, set #2) has 4 elements
Elements:
- 0
- 10
- 13
- 14

(continues on next page)
19.3 Indefinite maps

The previous sections presented containers for elements of definite types. Although most examples in those sections presented Integer types as element type of the containers, containers can also be used with indefinite types, an example of which is the String type. However, indefinite types require a different kind of containers designed specially for them.

We'll also be exploring a different class of containers: maps. They associate a key with a specific value. An example of a map is the one-to-one association between a person and their age. If we consider a person's name to be the key, the value is the person's age.

19.3.1 Hashed maps

Hashed maps are maps that make use of a hash as a key. The hash itself is calculated by a function you provide.

In other languages

Hashed maps are similar to dictionaries in Python and hashes in Perl. One of the main differences is that these scripting languages allow using different types for the values contained in a single map, while in Ada, both the type of key and value are specified in the package instantiation and remains constant for that specific map. You can't have a map where two elements are of different types or two keys are of different types. If you want to use multiple types, you must create a different map for each and use only one type in each map.

When instantiating a hashed map from Ada.Containers.Indefinite_Hashed_Maps, we specify following elements:

- Key_Type: type of the key
- Element_Type: type of the element
- Hash: hash function for the Key_Type
- Equivalent_Keys: an equality operator (e.g. =) that indicates whether two keys are to be considered equal.
  - If the type specified in Key_Type has a standard operator, you can use it, which you do by specifying that operator as the value of Equivalent_Keys.

In the next example, we'll use a string as a key type. We'll use the Hash function provided by the standard library for strings (in the Ada.Strings package) and the standard equality operator.

You add elements to a hashed map by calling Insert. If an element is already contained in a map M, you can access it directly by using its key. For example, you can change the value of an element by calling M ("My_Key") := 10. If the key is not found, an exception
is raised. To verify if a key is available, use the function `Contains` (as we've seen above in the section on sets).

Let's see an example:

Listing 18: show_hashed_map.adb

```ada
with Ada.Containers.Indefinite_Hashed_Maps;
with Ada.Strings.Hash;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Hashed_Map is

  package Integer_Hashed_Maps is new
    Ada.Containers.Indefinite_Hashed_Maps
      (Key_Type => String,
       Element_Type => Integer,
       Hash => Ada.Strings.Hash,
       Equivalent_Keys => ":=");

  use Integer_Hashed_Maps;

  M : Map;
  -- Same as:
  -- M : Integer_Hashed_Maps.Map;

begin
  M.Include ("Alice", 24);
  M.Include ("John", 40);
  M.Include ("Bob", 28);

  if M.Contains ("Alice") then
    Put_Line ("Alice's age is ", Integer'(Image (M ("Alice")))�);
  end if;

  -- Update Alice's age
  -- Key must already exist in M.
  -- Otherwise an exception is raised.
  M ("Alice") := 25;

  New_Line; Put_Line ("Name & Age:");
  for C in M.Iterate loop
    Put_Line (Key (C) & ": ", Integer'(Image (M (C))));
  end loop;

end Show_Hashed_Map;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Hashed_Map
MD5: 6117775bd9ce2b1466f448b100117ded

Runtime output

Alice's age is 24

Name & Age:
John: 40
Bob: 28
Alice: 25

19.3. Indefinite maps
19.3.2 Ordered maps

Ordered maps share many features with hashed maps. The main differences are:

- A hash function isn’t needed. Instead, you must provide an ordering function (< operator), which the ordered map will use to order elements and allow fast access, \( O(\log N) \), using a binary search.

  - If the type specified in Key_Type has a standard < operator, you can use it in a similar way as we did for Equivalent_Keys above for hashed maps.

Let’s see an example:

Listing 19: show_ordered_map.adb

```ada
with Ada.Containers.Indefinite_Ordered_Maps;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Ordered_Map is
  package Integer_Ordered_Maps is new
    Ada.Containers.Indefinite_Ordered_Maps
      (Key_Type => String,
       Element_Type => Integer);

  use Integer_Ordered_Maps;

  M : Map;

begin
  M.Include ("Alice", 24);
  M.Include ("John", 40);
  M.Include ("Bob", 28);

  if M.Contains ("Alice") then
    Put_Line ("Alice's age is ",
              & Integer'Image (M ("Alice")));
  end if;

  -- Update Alice's age
  -- Key must already exist in M
  M ("Alice") := 25;

  New_Line; Put_Line ("Name & Age: ");
  for C in M.Iterate loop
    Put_Line (Key (C) & ": "
              & Integer'Image (M (C)));
  end loop;

end Show_Ordered_Map;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Ordered_Map
MD5: 3deb3c685e767cee271b06e87727b086

Runtime output

Alice's age is 24

Name & Age:
Alice: 25

(continues on next page)
You can see a great similarity between the examples above and from the previous section. In fact, since both kinds of maps share many operations, we didn't need to make extensive modifications when we changed our example to use ordered maps instead of hashed maps. The main difference is seen when we run the examples: the output of a hashed map is usually unordered, but the output of a ordered map is always ordered, as implied by its name.

### 19.3.3 Complexity

Hashed maps are generally the fastest data structure available to you in Ada if you need to associate heterogeneous keys to values and search for them quickly. In most cases, they are slightly faster than ordered maps. So if you don't need ordering, use hashed maps.

The Reference Manual requires the following average complexity of operations:

<table>
<thead>
<tr>
<th>Operations</th>
<th>Ordered_Maps</th>
<th>Hashed_Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O((log N)^2) or better</td>
<td>O(log N)</td>
</tr>
<tr>
<td>Include</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclude</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subprogram using cursor

<table>
<thead>
<tr>
<th></th>
<th>Ordered_Maps</th>
<th>Hashed_Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
STANDARD LIBRARY: DATES & TIMES

The standard library supports processing of dates and times using two approaches:

- *Calendar* approach, which is suitable for handling dates and times in general;
- *Real-time* approach, which is better suited for real-time applications that require enhanced precision — for example, by having access to an absolute clock and handling time spans. Note that this approach only supports times, not dates.

The following sections present these two approaches.

20.1 Date and time handling

The Ada.Calendar package supports handling of dates and times. Let's look at a simple example:

Listing 1: display_current_time.adb

``` ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;
with Ada.Calendar.Formatting;
use Ada.Calendar.Formatting;

procedure Display_Current_Time is
    Now : Time := Clock;
begin
    Put_Line ("Current time: " & Image (Now));
end Display_Current_Time;
```

This example displays the current date and time, which is retrieved by a call to the Clock function. We call the function Image from the Ada.Calendar.Formatting package to get a String for the current date and time. We could instead retrieve each component using the Split function. For example:
Listing 2: display_current_year.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;

procedure Display_Current_Year is
  Now : Time := Clock;
  Now_Year : Year_Number;
  Now_Month : Month_Number;
  Now_Day : Day_Number;
  Now_Seconds : Day_Duration;
begin
  Split (Now,
        Now_Year,
        Now_Month,
        Now_Day,
        Now_Seconds);
  Put_Line ("Current year is: ",
             & Year_Number'Image (Now_Year));
  Put_Line ("Current month is: ",
             & Month_Number'Image (Now_Month));
  Put_Line ("Current day is: ",
             & Day_Number'Image (Now_Day));
end Display_Current_Year;
```

20.1.1 Delaying using date

You can delay an application so that it restarts at a specific date and time. We saw something similar in the chapter on tasking. You do this using a `delay until` statement. For example:

Listing 3: display_delay_next_specific_time.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;
with Ada.Calendar.Formatting;
use Ada.Calendar.Formatting;
with Ada.Calendar.Time_Zones;
use Ada.Calendar.Time_Zones;

procedure Display_Delay_Next_Specific_Time is
  TZ : Time_Offset := UTC_Time_Offset;
  Next : Time :=
  (continues on next page)
```
In this example, we specify the date and time by initializing Next using a call to Time_Of, a function taking the various components of a date (year, month, etc) and returning an element of the Time type. Because the date specified is in the past, the delay until statement won’t produce any noticeable effect. However, if we passed a date in the future, the program would wait until that specific date and time arrived.

Here we’re converting the time to the local timezone. If we don’t specify a timezone, Coordinated Universal Time (abbreviated to UTC) is used by default. By retrieving the time offset to UTC with a call to UTC_Time_Offset from the Ada.Calendar.Time_Zones package, we can initialize TZ and use it in the call to Time_Of. This is all we need do to make the information provided to Time_Of relative to the local time zone.

We could achieve a similar result by initializing Next with a String. We can do this with a call to Value from the Ada.Calendar.Formatting package. This is the modified code:

```ada
procedure Display_Delay_Next_Specific_Time is
  TZ : Time_Offset := UTC_Time_Offset;
```

Listing 4: display_delay_next_specific_time.adb
In this example, we're again using TZ in the call to Value to adjust the input time to the current time zone.

In the examples above, we were delaying to a specific date and time. Just like we saw in the tasking chapter, we could instead specify the delay relative to the current time. For example, we could delay by 5 seconds, using the current time:

```
Listing 5: display_delay_next.adb

with Ada.Calendar; use Ada.Calendar;
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_Delay_Next is
    D : Duration := 5.0;
    -- ^ seconds
    Now : Time := Clock;
    Next : Time := Now + D;
    -- ^ use duration to
    -- specify next
    -- point in time
begin
    Put_Line ("Let's wait 
        & Duration'Image (D)
        & " seconds...");
    delay until Next;
    Put_Line ("Enough waiting!");
end Display_Delay_Next;
```

Runtime output

```
Let's wait until... 2018-05-01 15:00:00.00
Enough waiting!
```
Let's wait 5.000000000 seconds...
Enough waiting!

Here, we’re specifying a duration of 5 seconds in D, adding it to the current time from Now, and storing the sum in Next. We then use it in the delay until statement.

### 20.2 Real-time

In addition to Ada.Calendar, the standard library also supports time operations for real-time applications. These are included in the Ada.Real_Time package. This package also includes a Time type. However, in the Ada.Real_Time package, the Time type is used to represent an absolute clock and handle a time span. This contrasts with the Ada.Calendar, which uses the Time type to represent dates and times.

In the previous section, we used the Time type from the Ada.Calendar and the delay until statement to delay an application by 5 seconds. We could have used the Ada.Real_Time package instead. Let’s modify that example:

Listing 6: display_delay_next_real_time.adb

```ada
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;

procedure Display_Delay_Next_Real_Time is
  D : Time_Span := Seconds (5);
  Next : Time := Clock + D;
begin
  Put_Line ("Let's wait 
            & Duration'Image (To_Duration (D))
            & " seconds...");
  delay until Next;
  Put_Line ("Enough waiting!");
end Display_Delay_Next_Real_Time;
```

The main difference is that D is now a variable of type Time_Span, defined in the Ada.Real_Time package. We call the function Seconds to initialize D, but could have gotten a finer granularity by calling Nanoseconds instead. Also, we need to first convert D to the Duration type using To_Duration before we can display it.
20.2.1 Benchmarking

One interesting application using the Ada.Real_Time package is benchmarking. We've used that package before in a previous section when discussing tasking. Let's look at an example of benchmarking:

Listing 7: display_benchmarking.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;

procedure Display_Benchmarking is
  procedure Computational_Intensive_App is
  begin
    delay 5.0;
  end Computational_Intensive_App;
  Start_Time, Stop_Time : Time;
  Elapsed_Time : Time_Span;

  begin
    Start_Time := Clock;
    Computational_Intensive_App;
    Stop_Time := Clock;
    Elapsed_Time := Stop_Time - Start_Time;
    Put_Line ("Elapsed time:
               & Duration'Image
               (To_Duration (Elapsed_Time))
               & " seconds");
  end Display_Benchmarking;
```

This example defines a dummy Computational_Intensive_App implemented using a simple delay statement. We initialize Start_Time and Stop_Time from the then-current clock and calculate the elapsed time. By running this program, we see that the time is roughly 5 seconds, which is expected due to the delay statement.

A similar application is benchmarking of CPU time. We can implement this using the Execution_Time package. Let's modify the previous example to measure CPU time:

Listing 8: display_benchmarking_cpu_time.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Execution_Time; use Ada.Execution_Time;

procedure Display_Benchmarking_CPU_Time is
  procedure Computational_Intensive_App is
  begin
    (continues on next page)
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Benchmarking
MD5: 4b20940cb613d3f634be5224f409efeb

Runtime output

Elapsed time: 5.024813163 seconds
In this example, Start_Time and Stop_Time are of type CPU_Time instead of Time. However, we still call the Clock function to initialize both variables and calculate the elapsed time in the same way as before. By running this program, we see that the CPU time is significantly lower than the 5 seconds we've seen before. This is because the delay statement doesn't require much CPU time. The results will be different if we change the implementation of Computational_Intensive_App to use a mathematical function in a long loop. For example:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Execution_Time; use Ada.Execution_Time;
with Ada.Numerics.Generic_Elementary_Functions;

procedure Display_Benchmarking_Math is
  procedure Computational_Intensive_App is
    package Funcs is new
      Ada.Numerics.Generic_Elementary_Functions
      (Float_Type => Long_Long_Float);
    use Funcs;
    X : Long_Long_Float;
  begin
    for I in 0 .. 1_000_000 loop
      X := Tan (Arctan
      (Tan (Arctan
        (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
          (Tan (Arctan
```

(continues on next page)
Learning Ada

(continued from previous page)

(Tan (Arctan (0.577)))))))))))));

end loop;
end Computational_Intensive_App;

procedure Benchm Elapsed_Time is
Start_Time, Stop_Time : Time;
Elapsed_Time : Time_Span;

begin
Start_Time := Clock;
Computational_Intensive_App;
Stop_Time := Clock;
Elapsed_Time := Stop_Time - Start_Time;
Put_Line ("Elapsed time: 
  & Duration'Image
  (To_Duration (Elapsed_Time))
  & " seconds");
end Benchm Elapsed_Time;

procedure Benchm_CPU_Time is
Start_Time, Stop_Time : CPU_Time;
Elapsed_Time : Time_Span;

begin
Start_Time := Clock;
Computational_Intensive_App;
Stop_Time := Clock;
Elapsed_Time := Stop_Time - Start_Time;
Put_Line ("CPU time: 
  & Duration'Image
  (To_Duration (Elapsed_Time))
  & " seconds");
end Benchm_CPU_Time;

begin
Benchm Elapsed_Time;
Benchm_CPU_Time;
end Display_Benchmarking_Math;

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Benchmarking_Math
MD5: 06fe96bf03321c248dd1ed843648cf0b

Runtime output

Elapsed time: 1.365845193 seconds
CPU time: 1.276072943 seconds

Now that our dummy Computational_Intensive_App involves mathematical operations requiring significant CPU time, the measured elapsed and CPU time are much closer to each other than before.
In previous chapters, we've seen source-code examples using the String type, which is a fixed-length string type — essentially, it's an array of characters. In many cases, this data type is good enough to deal with textual information. However, there are situations that require more advanced text processing. Ada offers alternative approaches for these cases:

- **Bounded strings**: similar to fixed-length strings, bounded strings have a maximum length, which is set at its instantiation. However, bounded strings are not arrays of characters. At any time, they can contain a string of varied length — provided this length is below or equal to the maximum length.

- **Unbounded strings**: similar to bounded strings, unbounded strings can contain strings of varied length. However, in addition to that, they don't have a maximum length. In this sense, they are very flexible.

The following sections present an overview of the different string types and common operations for string types.

### 21.1 String operations

Operations on standard (fixed-length) strings are available in the Ada.Strings.Fixed package. As mentioned previously, standard strings are arrays of elements of Character type with a fixed-length. That's why this child package is called Fixed.

One of the simplest operations provided is counting the number of substrings available in a string (Count) and finding their corresponding indices (Index). Let's look at an example:

Listing 1: show_find_substring.adb

```ada
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;  use Ada.Text_IO;

procedure Show_Find_Substring is

   S : String := "Hello" & 3 * " World";
   P : constant String := "World";
   Idx : Natural;
   Cnt : Natural;

begin
   Cnt := Ada.Strings.Fixed.Count (Source => S,
                                     Pattern => P);
   Put_Line ("String: " & S);
   Put_Line ("Count for '", P & "': "
             & Natural'Image (Cnt));

   for I in 0 .. Cnt - 1 loop
      Idx := Ada.Strings.Fixed.Index (Source => S,
                                      Pattern => P);
      Put_Line ("Index for '", P & "': "
                & Natural'Image (Idx));

      while Idx < Cnt loop
         Idx := Ada.Strings.Fixed.Index (Source => S,
                                         Pattern => P,
                                         Start => Idx);
         if Idx < Cnt then
            Put_Line ("Index for '", P & "': "
                      & Natural'Image (Idx));
         else
            Put_Line ("Pattern not found");
         end if;
      end loop;
   end loop;

end Show_Find_Substring;
```

(continues on next page)
Idx := 0;
for I in 1 .. Cnt loop
    Idx := Index (Source => S, Pattern => P, From => Idx + 1);
end loop;
end Show_Find_Substring;

Code block metadata

Project: Courses.Intro To Ada.Standard_Library.Show_Find_Substring
MD5: faa8373bf9a9f0f9f5507cf55590b0c0

Runtime output

String: Hello World World World
Count for 'World': 3
Found instance of 'World' at position: 7
Found instance of 'World' at position: 13
Found instance of 'World' at position: 19

We initialize the string S using a multiplication. Writing "Hello" & 3 * " World" creates the string Hello World World World. We then call the function Count to get the number of instances of the word World in S. Next we call the function Index in a loop to find the index of each instance of World in S.

That example looked for instances of a specific substring. In the next example, we retrieve all the words in the string. We do this using Find_Token and specifying whitespaces as separators. For example:

Listing 2: show_find_words.adb

with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Find_Words is
    S : String := "Hello" & 3 * " World";
    F : Positive;
    L : Natural;
    I : Natural := 1;
    Whitespace : constant Character_Set := To_Set (' ');
begin
    Put_Line ("String: " & S);
    Put_Line ("String length: " & Integer'Image (S'Length));

    while I in S'Range loop
            Find_Token
                (Source => S, Set => Whitespace,
We pass a set of characters to be used as delimiters to the procedure Find_Token. This set is a member of the Character_Set type from the Ada.Strings.Maps package. We call the To_Set function (from the same package) to initialize the set to Whitespace and then call Find_Token to loop over each valid index and find the starting index of each word. We pass Outside to the Test parameter of the Find_Token procedure to indicate that we're looking for indices that are outside the Whitespace set, i.e. actual words. The First and Last parameters of Find_Token are output parameters that indicate the valid range of the substring. We use this information to display the string (S (F .. L)).

The operations we've looked at so far read strings, but don't modify them. We next discuss operations that change the content of strings:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Insert substring in a string</td>
</tr>
<tr>
<td>Overwrite</td>
<td>Overwrite a string with a substring</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete a substring</td>
</tr>
<tr>
<td>Trim</td>
<td>Remove whitespaces from a string</td>
</tr>
</tbody>
</table>

All these operations are available both as functions or procedures. Functions create a new string but procedures perform the operations in place. The procedure will raise an exception if the constraints of the string are not satisfied. For example, if we have a string S containing 10 characters, inserting a string with two characters (e.g. "!!") into it produces a string containing 12 characters. Since it has a fixed length, we can't increase its size. One possible solution in this case is to specify that truncation should be applied while inserting the substring. This keeps the length of S fixed. Let's see an example that makes use of both function and procedure versions of Insert, Overwrite, and Delete:
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Adapted_Strings is

  S : String := "Hello World";
  P : constant String := "World";
  N : constant String := "Beautiful"

procedure Display_Adapted_String
  (Source : String;
   Before : Positive;
   New_Item : String;
   Pattern : String)
is

  S_Ins_In : String := Source;
  S_Ovr_In : String := Source;
  S_Del_In : String := Source;

  S_Ins : String :=
  Insert (Source,
           Before,
           New_Item & " ");
  S_Ovr : String :=
  Overwrite (Source,
             Before,
             New_Item);
  S_Del : String :=
  Trim (Delete (Source,
                Before,
                Before + Pattern'Length - 1),
             Ada.Strings.Right);

begin
  Insert (S_Ins_In,
           Before,
           New_Item,
           Right);

  Overwrite (S_Ovr_In,
             Before,
             New_Item,
             Right);

  Delete (S_Del_In,
           Before,
           Before + Pattern'Length - 1);

  Put_Line ("Original: "
            & Source & "\n");
  Put_Line ("Insert: "
            & S_Ins & "\n");
  Put_Line ("Overwrite: "
            & S_Ovr & "\n");
  Put_Line ("Delete: "
            & S_Del & "\n");

  Put_Line ("Insert (in-place): ")

(continues on next page)
Using fixed-length strings is usually good enough for strings that are initialized when they are declared. However, as seen in the previous section, procedural operations on strings cause difficulties when done on fixed-length strings because fixed-length strings are arrays of characters. The following example shows how cumbersome the initialization of fixed-length strings can be when it’s not performed in the declaration:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Char_Array is
   S : String (1 .. 15);  -- Strings are arrays of Character
   begin
      S := "Hello ";
end Show_Char_Array;
```

In this example, we look for the index of the substring World and perform operations on this substring within the outer string. The procedure Display_Adapted_String uses both versions of the operations. For the procedural version of Insert and Overwrite, we apply truncation to the right side of the string (Right). For the Delete procedure, we specify the range of the substring, which is replaced by whitespaces. For the function version of Delete, we also call Trim which trims the trailing whitespace.

### 21.2 limitation of fixed-length strings

Using fixed-length strings is usually good enough for strings that are initialized when they are declared. However, as seen in the previous section, procedural operations on strings cause difficulties when done on fixed-length strings because fixed-length strings are arrays of characters. The following example shows how cumbersome the initialization of fixed-length strings can be when it’s not performed in the declaration:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Char_Array is
   S : String (1 .. 15);  -- Strings are arrays of Character
   begin
      S := "Hello ";
end Show_Char_Array;
```
-- Alternatively:
-- #1:
-- S (1 .. 5) := "Hello";
-- S (6 .. S'Last) := (others => ' ');
-- #2:
-- S := ('H', 'e', 'l', 'l', 'o',
-- others => ' ');

Put_Line ("String: " & S);
Put_Line ("String Length: "
& Integer'Image (S'Length));
end Show_Char_Array;

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Char_Array
MD5: 9f3df03c9c5336184139cf2a22f2cb7e

Runtime output

String: Hello
String Length: 15

In this case, we can't simply write S := "Hello" because the resulting array of characters
for the Hello constant has a different length than the S string. Therefore, we need to
include trailing whitespaces to match the length of S. As shown in the example, we could
use an exact range for the initialization (S (1 .. 5)) or use an explicit array of individual
characters.

When strings are initialized or manipulated at run-time, it's usually better to use bounded
or unbounded strings. An important feature of these types is that they aren't arrays, so the
difficulties presented above don't apply. Let's start with bounded strings.

21.3 Bounded strings

Bounded strings are defined in the Ada.Strings.Bounded.Generic_Bounded_Length pack-
age. Because this is a generic package, you need to instantiate it and set the maximum
length of the bounded string. You can then declare bounded strings of the Bounded_String
type.

Both bounded and fixed-length strings have a maximum length that they can hold. How-
ever, bounded strings are not arrays, so initializing them at run-time is much easier. For
example:

Listing 5: show_bounded_string.adb

with Ada.Strings; use Ada.Strings;
with Ada.Strings.Bounded; use Ada.Text_IO;

procedure Show_Bounded_String is
  package B_Str is new
    Ada.Strings.Bounded.Generic_Bounded_Length
      (Max => 15);
  use B_Str;
11 S1, S2 : Bounded_String;

procedure Display_String_Info
(S : Bounded_String)
is
begin
  Put_Line ("String: " & To_String (S));
  Put_Line ("String Length: "
            & Integer'Image (Length (S)));
  -- String:
  -- S'Length => ok
  -- Bounded_String:
  -- S'Length => compilation error:
  -- bounded strings are
  -- not arrays!
  Put_Line ("Max. Length: 
            & Integer'Image (Max_Length));
end Display_String_Info;

begin
  S1 := To_Bounded_String ("Hello");
  Display_String_Info (S1);

  S2 := To_Bounded_String ("Hello World");
  Display_String_Info (S2);

  S1 := To_Bounded_String ("Something longer to say here...",
                           Right);
  Display_String_Info (S1);
end Show_Bounded_String;

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Bounded_String
MD5: a51fdeacfd43923145e92bf5c72ecd6

Runtime output

String: Hello
String Length: 5
Max. Length: 15
String: Hello World
String Length: 11
Max. Length: 15
String: Something longe
String Length: 15
Max. Length: 15

By using bounded strings, we can easily assign to S1 and S2 multiple times during execution. We use the To_Bounded_String and To_String functions to convert, in the respective direction, between fixed-length and bounded strings. A call to To_Bounded_String raises an exception if the length of the input string is greater than the maximum capacity of the bounded string. To avoid this, we can use the truncation parameter (Right in our example).

Bounded strings are not arrays, so we can't use the 'Length attribute as we did for fixed-length strings. Instead, we call the Length function, which returns the length of the bounded string. The Max_Length constant represents the maximum length of the bounded string that we set when we instantiated the package.

After initializing a bounded string, we can manipulate it. For example, we can append...
Learning Ada

a string to a bounded string using Append or concatenate bounded strings using the & operator. Like so:

Listing 6: show_bound_string_op.adb

```ada
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Bounded; use Ada.Text_IO;

procedure Show_Bounded_String_Op is
  package B_Str is new
    Ada.Strings.Bounded.Generic_Bounded_Length
    (Max => 30);
  use B_Str;
  S1, S2 : Bounded_String;
begin
  S1 := To_Bounded_String ("Hello");
  -- Alternatively:
  -- A := Null_Bounded_String & "Hello";
  Append (S1, " World");
  -- Alternatively:
  -- Append (A, " World", Right);
  Put_Line ("String: ", To_String (S1));
  S2 := To_Bounded_String ("Hello!");
  S1 := S1 & " " & S2;
  Put_Line ("String: ", To_String (S1));
end Show_Bounded_String_Op;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Bounded_String_Op
MD5: c7c6a840c314a9cd9f75aac082a63159

Runtime output

String: Hello World
String: Hello World Hello!

We can initialize a bounded string with an empty string using the Null_Bounded_String constant. Also, we can use the Append procedure and specify the truncation mode like we do with the To_Bounded_String function.

21.4 Unbounded strings

Unbounded strings are defined in the Ada.Strings.Unbounded package. This is not a generic package, so we don't need to instantiate it before using the Unbounded_String type. As you may recall from the previous section, bounded strings require a package instantiation.

Unbounded strings are similar to bounded strings. The main difference is that they can hold strings of any size and adjust according to the input string: if we assign, e.g., a 10-character string to an unbounded string and later assign a 50-character string, internal operations in the container ensure that memory is allocated to store the new string. In most cases, developers don't need to worry about these operations. Also, no truncation is necessary.
Initialization of unbounded strings is very similar to bounded strings. Let's look at an example:

Listing 7: show_unbounded_string.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

procedure Show_Unbounded_String is
  S1, S2 : Unbounded_String;

  procedure Display_String_Info (S : Unbounded_String)
  is
  begin
    Put_Line ("String: " & To_String (S));
    Put_Line ("String Length: " & Integer'Image (Length (S)));
  end Display_String_Info;

begin
  S1 := To_Unbounded_String ("Hello");
  -- Alternatively:
  -- A := Null_Unbounded_String & "Hello"
  Display_String_Info (S1);
  S2 := To_Unbounded_String ("Hello World");
  Display_String_Info (S2);
  S1 := To_Unbounded_String ("Something longer to say here...");
  Display_String_Info (S1);
end Show_Unbounded_String;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Unbounded_String
MD5: 904402992c96eb393b875d1b7cf49c1b

Runtime output

String: Hello
String Length: 5
String: Hello World
String Length: 11
String: Something longer to say here...
String Length: 31

Like bounded strings, we can assign to S1 and S2 multiple times during execution and use the To_Unbounded_String and To_String functions to convert back-and-forth between fixed-length strings and unbounded strings. However, in this case, truncation is not needed.

And, just like for bounded strings, you can use the Append procedure and the & operator for unbounded strings. For example:

Listing 8: show_unbounded_string_op.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
(continues on next page)
```
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

procedure Show_Unbounded_String_Op is
  S1, S2 : Unbounded_String := Null_Unbounded_String;
begin
  S1 := S1 & "Hello";
  S2 := S2 & "Hello!";
  Append (S1, " World");
  Put_Line ("String: " & To_String (S1));
  S1 := S1 & " " & S2;
  Put_Line ("String: " & To_String (S1));
end Show_Unbounded_String_Op;

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Unbounded_String_Op
MD5: 806e24a6dd0bc87e76f73a22e42ba390

Runtime output

String: Hello World
String: Hello World Hello!

In this example, we're concatenating the unbounded S1 and S2 strings with the "Hello" and "Hello!" strings, respectively. Also, we're using the Append procedure, just like we did with bounded strings.
STANDARD LIBRARY: FILES AND STREAMS

Ada provides different approaches for file input/output (I/O):

- **Text I/O**, which supports file I/O in text format, including the display of information on the console.
- **Sequential I/O**, which supports file I/O in binary format written in a sequential fashion for a specific data type.
- **Direct I/O**, which supports file I/O in binary format for a specific data type, but also supporting access to any position of a file.
- **Stream I/O**, which supports I/O of information for multiple data types, including objects of unbounded types, using files in binary format.

This table presents a summary of the features we've just seen:

<table>
<thead>
<tr>
<th>File I/O option</th>
<th>Format</th>
<th>Random access</th>
<th>Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text I/O</td>
<td>text</td>
<td></td>
<td>string type</td>
</tr>
<tr>
<td>Sequential I/O</td>
<td>binary</td>
<td></td>
<td>single type</td>
</tr>
<tr>
<td>Direct I/O</td>
<td>binary</td>
<td>✓</td>
<td>single type</td>
</tr>
<tr>
<td>Stream I/O</td>
<td>binary</td>
<td>✓</td>
<td>multiple types</td>
</tr>
</tbody>
</table>

In the following sections, we discuss details about these I/O approaches.

### 22.1 Text I/O

In most parts of this course, we used the Put_Line procedure to display information on the console. However, this procedure also accepts a **File_Type** parameter. For example, you can select between standard output and standard error by setting this parameter explicitly:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Std_Text_Out is
begin
    Put_Line (Standard_Output, "Hello World #1");
    Put_Line (Standard_Error, "Hello World #2");
end Show_Std_Text_Out;
```

**Code block metadata**

Project: Courses.Intro_To_Ada.Standard_Library.Show_Std_Text_Out
MD5: 4d75bd2906226897244e3d2a611c9725
You can also use this parameter to write information to any text file. To create a new file for writing, use the Create procedure, which initializes a **File Type** element that you can later pass to Put_Line (instead of, e.g., Standard_Output). After you finish writing information, you can close the file by calling the Close procedure.

You use a similar method to read information from a text file. However, when opening the file, you must specify that it's an input file (In_File) instead of an output file. Also, instead of calling the Put_Line procedure, you call the Get_Line function to read information from the file.

Let's see an example that writes information into a new text file and then reads it back from the same file:

```
Listing 2: show_simple_text_file_io.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Text_File_IO is
    F : File_Type;
    File_Name : constant String := "simple.txt";
begin
    Create (F, Out_File, File_Name);
    Put_Line (F, "Hello World #1");
    Put_Line (F, "Hello World #2");
    Put_Line (F, "Hello World #3");
    Close (F);

    Open (F, In_File, File_Name);
    while not End_Of_File (F) loop
        Put_Line (Get_Line (F));
    end loop;
    Close (F);
end Show_Simple_Text_File_IO;
```

In addition to the Create and Close procedures, the standard library also includes a Reset procedure, which, as the name implies, resets (erases) all the information from the file. For example:

```
Listing 3: show_text_file_reset.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Text_File_Reset is
    F : File_Type;
    File_Name : constant String := "simple.txt";
begin
    (continues on next page)
```

In addition to the Create and Close procedures, the standard library also includes a Reset procedure, which, as the name implies, resets (erases) all the information from the file. For example:

```
Listing 3: show_text_file_reset.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Text_File_Reset is
    F : File_Type;
    File_Name : constant String := "simple.txt";
begin
```

(continues on next page)
Create (F, Out_File, File_Name);
Put_Line (F, "Hello World #1");
Reset (F);
Put_Line (F, "Hello World #2");
Close (F);

Open (F, In_File, File_Name);
while not End_Of_File (F) loop
  Put_Line (Get_Line (F));
end loop;
Close (F);
end Show_Text_File_Reset;

By running this program, we notice that, although we've written the first string ("Hello World #1") to the file, it has been erased because of the call to Reset.

In addition to opening a file for reading or writing, you can also open an existing file and append to it. Do this by calling the Open procedure with the Append_File option.

When calling the Open procedure, an exception is raised if the specified file isn't found. Therefore, you should handle exceptions in that context. The following example deletes a file and then tries to open the same file for reading:

Listing 4: show_text_file_input_except.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Text_File_Input_Except is
  F : File_Type;
  File_Name : constant String := "simple.txt";
begin
  -- Open output file and delete it
  Create (F, Out_File, File_Name);
  Delete (F);

  -- Try to open deleted file
  Open (F, In_File, File_Name);
  Close (F);
exception
  when Name_Error =>
    Put_Line ("File does not exist");
  when others =>
    Put_Line ("Error while processing input file");
end Show_Text_File_Input_Except;
Learning Ada

File does not exist

In this example, we create the file by calling Create and then delete it by calling Delete. After the call to Delete, we can no longer use the **File_Type** element. After deleting the file, we try to open the non-existent file, which raises a **Name_Error** exception.

### 22.2 Sequential I/O

The previous section presented details about text file I/O. Here, we discuss doing file I/O in binary format. The first package we'll explore is the **Ada.Sequential_IO** package. Because this package is a generic package, you need to instantiate it for the data type you want to use for file I/O. Once you've done that, you can use the same procedures we've seen in the previous section: Create, Open, Close, Reset and Delete. However, instead of calling the Get_Line and Put_Line procedures, you'd call the Read and Write procedures.

In the following example, we instantiate the **Ada.Sequential_IO** package for floating-point types:

``` ada
with Ada.Text_IO;  
with Ada.Sequential_IO; 

procedure Show_Seq_Float_IO is 
  package Float_IO is 
    new Ada.Sequential_IO (Float); 
    use Float_IO; 
  end Float_IO; 

  F : Float_IO.File_Type; 
  File_Name : constant String := 
    "float_file.bin"; 

  begin 
    Create (F, Out_File, File_Name); 
    Write (F, 1.5); 
    Write (F, 2.4); 
    Write (F, 6.7); 
    Close (F); 

    declare 
      Value : Float; 
    begin 
      Open (F, In_File, File_Name); 
      while not End_Of_File (F) loop 
        Read (F, Value); 
        Ada.Text_IO.Put_Line 
          (Float'Image (Value)); 
      end loop; 
      Close (F); 
    end; 
  end Show_Seq_Float_IO; 
```

**Runtime output**

Project: Courses.Intro_To_Ada.Standard_Library.Show_Seq_Float_IO
MD5: 27aa5daf92c5a5f23f5d5f5f53f578aa34
We use the same approach to read and write complex information. The following example uses a record that includes a Boolean and a floating-point value:

Listing 6: show_seq_rec_io.adb

```ada
with Ada.Text_IO;
with Ada.Sequential_IO;

procedure Show_Seq_Rec_IO is
  type Num_Info is record
    Valid : Boolean := False;
    Value : Float;
  end record;

  procedure Put_Line (N : Num_Info) is
  begin
    if N.Valid then
      Ada.Text_IO.Put_Line
      ("(ok, " & Float'Image (N.Value) & ")");
    else
      Ada.Text_IO.Put_Line
      ("(not ok, " & Float'Image (N.Value) & ")");
    end if;
  end Put_Line;

  package Num_Info_IO is new Ada.Sequential_IO (Num_Info);
  use Num_Info_IO;

  F : Num_Info_IO.File_Type;
  File_Name : constant String := "float_file.bin";
begin
  Create (F, Out_File, File_Name);
  Write (F, (True, 1.5));
  Write (F, (False, 2.4));
  Write (F, (True, 6.7));
  Close (F);

  declare
    Value : Num_Info;
  begin
    Open (F, In_File, File_Name);
    while not End_Of_File (F) loop
      Read (F, Value);
      Put_Line (Value);
    end loop;
    Close (F);
  end;
end Show_Seq_Rec_IO;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Seq_Rec_IO
MD5: a88b1428cc50745dce0509087e74adb7

Runtime output
As the example shows, we can use the same approach we used for floating-point types to perform file I/O for this record. Once we instantiate the Ada.Sequential_IO package for the record type, file I/O operations are performed the same way.

### 22.3 Direct I/O

Direct I/O is available in the Ada.Direct_IO package. This mechanism is similar to the sequential I/O approach just presented, but allows us to access any position in the file. The package instantiation and most operations are very similar to sequential I/O. To rewrite the Show_Seq_Float_IO application presented in the previous section to use the Ada.Direct_IO package, we just need to replace the instances of the Ada.Sequential_IO package by the Ada.Direct_IO package. This is the new source code:

```
with Ada.Text_IO;
with Ada.Direct_IO;

procedure Show_Dir_Float_IO is
  package Float_IO is new Ada.Direct_IO (Float);
  use Float_IO;

  F : Float_IO.File_Type;
  File_Name : constant String := "float_file.bin";

begin
  Create (F, Out_File, File_Name);
  Write (F, 1.5);
  Write (F, 2.4);
  Write (F, 6.7);
  Close (F);

  declare
    Value : Float;
  begin
    Open (F, In_File, File_Name);
    while not End_Of_File (F) loop
      Read (F, Value);
      Ada.Text_IO.Put_Line (Float'Image (Value));
    end loop;
    Close (F);
  end Show_Dir_Float_IO;
```

---

**Code block metadata**

Project: Courses.Intro_To_Ada.Standard_Library.Show_Dir_Float_IO

MD5: e4e5855976de44f53a821eb98dcb206

**Runtime output**

1.50000E+00
2.40000E+00
6.70000E+00
Unlike sequential I/O, direct I/O allows you to access any position in the file. However, it doesn't offer an option to append information to a file. Instead, it provides an Inout_File mode allowing reading and writing to a file via the same File_Type element.

To access any position in the file, call the Set_Index procedure to set the new position / index. You can use the Index function to retrieve the current index. Let's see an example:

```
with Ada.Text_IO;
with Ada.Direct_IO;

procedure Show_Dir_Float_In_Out_File is
   package Float_IO is new Ada.Direct_IO (Float);
   use Float_IO;

   F : Float_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   -- Open file for input / output
   Create (F, Inout_File, File_Name);
   Write (F, 1.5);
   Write (F, 2.4);
   Write (F, 6.7);

   -- Set index to previous position
   -- and overwrite value
   Set_Index (F, Index (F) - 1);
   Write (F, 7.7);

   declare
      Value : Float;
   begin
      -- Set index to start of file
      Set_Index (F, 1);

      while not End_Of_File (F) loop
         Read (F, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
   end;
end Show_Dir_Float_In_Out_File;
```

By running this example, we see that the file contains 7.7, rather than the previous 6.7 that we wrote. We overwrote the value by changing the index to the previous position before doing another write.

In this example we used the Inout_File mode. Using that mode, we just changed the index back to the initial position before reading from the file (Set_Index (F, 1)) instead
of closing the file and reopening it for reading.

## 22.4 Stream I/O

All the previous approaches for file I/O in binary format (sequential and direct I/O) are specific for a single data type (the one we instantiate them with). You can use these approaches to write objects of a single data type that may be an array or record (potentially with many fields), but if you need to create and process files that include different data types, or any objects of an unbounded type, these approaches are not sufficient. Instead, you should use stream I/O.

Stream I/O shares some similarities with the previous approaches. We still use the Create, Open and Close procedures. However, instead of accessing the file directly via a *File_Type* element, you use a *Stream_Access* element. To read and write information, you use the *'Read* or *'Write* attributes of the data types you're reading or writing.

Let's look at a version of the *Show_Dir_Float_I0* procedure from the previous section that makes use of stream I/O instead of direct I/O:

```ada
with Ada.Text_IO;
with Ada.Streams.Stream_IO;
use Ada.Streams.Stream_IO;

procedure Show_Float_Stream
is
   F : File_Type;
   S : Stream_Access;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   S := Stream (F);
   Float'Write (S, 1.5);
   Float'Write (S, 2.4);
   Float'Write (S, 6.7);
   Close (F);
   declare
      Value : Float;
   begin
      Open (F, In_File, File_Name);
      S := Stream (F);
      while not End_Of_File (F) loop
         Float'Read (S, Value);
         Ada.Text_IO.Put_Line (Float'I mage (Value));
      end loop;
      Close (F);
   end;
end Show_Float_Stream;
```

---

**Code block metadata**

- **Project:** Courses.Intro_To_Ada.Standard_Library.Show_Float_Stream
- **MD5:** 34ccf04b0821074a332019ac0e38bb3e

256 Chapter 22. Standard library: Files and streams
After the call to Create, we retrieve the corresponding Stream_Access element by calling the Stream function. We then use this stream to write information to the file via the 'Write attribute of the Float type. After closing the file and reopening it for reading, we again retrieve the corresponding Stream_Access element and processed to read information from the file via the 'Read attribute of the Float type.

You can use streams to create and process files containing different data types within the same file. You can also read and write unbounded data types such as strings. However, when using unbounded data types you must call the 'Input and 'Output attributes of the unbounded data type: these attributes write information about bounds or discriminants in addition to the object's actual data.

The following example shows file I/O that mixes both strings of different lengths and floating-point values:

```ada
with Ada.Text_IO;
with Ada.Streams.Stream_IO;
use Ada.Streams.Stream_IO;

procedure Show_String_Stream is
    F : File_Type;
    S : Stream_Access;
    File_Name : constant String := "float_file.bin";

    procedure Output (S : Stream_Access;
                      FV : Float;
                      SV : String) is
        begin
            String'Output (S, SV);
            Float'Output (S, FV);
        end Output;

    procedure Input_Display (S : Stream_Access) is
        SV : String := String'Input (S);
        FV : Float := Float'Input (S);
        begin
            Ada.Text_IO.Put_Line (Float'Image (FV)
                                  & " --- " & SV);
        end Input_Display;

    begin
        Create (F, Out_File, File_Name);
        S := Stream (F);
        Output (S, 1.5, "Hi!!");
        Output (S, 2.4, "Hello world!");
        Output (S, 6.7, "Something longer here...");
        Close (F);
        Open (F, In_File, File_Name);
        S := Stream (F);
    end Show_String_Stream;
```

(continues on next page)
#### Code block metadata

**Project**: Courses.Intro_To_Ada.Standard_Library.Show_String_Stream  
**MD5**: 3ae8276ada5f24cab49994e368e0fa34

#### Runtime output

<table>
<thead>
<tr>
<th>Time</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.50000E+00</td>
<td>--- Hi!!</td>
</tr>
<tr>
<td>2.40000E+00</td>
<td>--- Hello world!</td>
</tr>
<tr>
<td>6.70000E+00</td>
<td>--- Something longer here...</td>
</tr>
</tbody>
</table>

When you use Stream I/O, no information is written into the file indicating the type of the data that you wrote. If a file contains data from different types, you must reference types in the same order when reading a file as when you wrote it. If not, the information you get will be corrupted. Unfortunately, strong data typing doesn't help you in this case. Writing simple procedures for file I/O (as in the example above) may help ensuring that the file format is consistent.

Like direct I/O, stream I/O support also allows you to access any location in the file. However, when doing so, you need to be extremely careful that the position of the new index is consistent with the data types you're expecting.
The standard library provides support for common numeric operations on floating-point types as well as on complex types and matrices. In the sections below, we present a brief introduction to these numeric operations.

### 23.1 Elementary Functions

The Ada.Numerics.Elementary_Functions package provides common operations for floating-point types, such as square root, logarithm, and the trigonometric functions (e.g., sin, cos). For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;

procedure Show_Elem_Math is
    X : Float;
begin
    X := 2.0;
    Put_Line ("Square root of " & Float'Image (X) & " is " & Float'Image (Sqrt (X)));

    X := e;
    Put_Line ("Natural log of " & Float'Image (X) & " is " & Float'Image (Log (X)));

    X := 10.0 ** 6.0;
    Put_Line ("Log_10 of " & Float'Image (X) & " is " & Float'Image (Log (X, 10.0)));

    X := 2.0 ** 8.0;
    Put_Line ("Log_2 of " & Float'Image (X) & " is " & Float'Image (Log (X, 2.0)));

    (continues on next page)
```
Learning Ada

(continued from previous page)

```ada
X := Pi;
Put_Line ("Cos of "
& Float'Image (X)
& " is "
& Float'Image (Cos (X)));

X := -1.0;
Put_Line ("Arccos of "
& Float'Image (X)
& " is "
& Float'Image (Arccos (X)));
end Show_Elem_Math;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Elem_Math
MD5: 17511d7e17cd98d4b6e49ad302d6dcb6

Runtime output

Square root of 2.00000E+00 is 1.41421E+00
Natural log of 2.71828E+00 is 1.00000E+00
Log_10 of 1.00000E+06 is 6.00000E+00
Log_2 of 2.56000E+02 is 8.00000E+00
Cos of 3.14159E+00 is -1.00000E+00
Arccos of -1.00000E+00 is 3.14159E+00

Here we use the standard e and Pi constants from the Ada.Numerics package.

The Ada.Numerics.Elementary_Functions package provides operations for the Float type. Similar packages are available for Long_Float and Long_Long_Float types. For example, the Ada.Numerics.Long_Elementary_Functions package offers the same set of operations for the Long_Float type. In addition, the Ada.Numerics. Generic_Elementary_Functions package is a generic version of the package that you can instantiate for custom floating-point types. In fact, the Elementary_Functions package can be defined as follows:

```
package Elementary_Functions is new
    Ada.Numerics.Generic_Elementary_Functions (Float);
```

### 23.2 Random Number Generation

The Ada.Numerics.Float_Random package provides a simple random number generator for the range between 0.0 and 1.0. To use it, declare a generator G, which you pass to Random. For example:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Float_Random;
use Ada.Numerics.Float_Random;
procedure Show_Float_Random_Num is
    G : Generator;
    X : Uniformly_Distributed;
begin
```

(continues on next page)
Reset (G);

Put_Line ("Some random numbers between 
  & Float'Image 
  (Uniformly_Distributed'First) 
  & " and " 
  & Float'Image 
  (Uniformly_Distributed'Last) 
  & ":");
for I in 1 .. 15 loop
  X := Random (G);
  Put_Line (Float'Image (X));
end loop;
end Show_Float_Random_Num;

Code block metadata
Project: Courses.Intro_To_Ada.Standard_Library.Show_Float_Random_Num
MD5: cf38ab00e27bad4309010e678113dd36

Runtime output
Some random numbers between 0.00000E+00 and 1.00000E+00:
9.74383E-01
3.73925E-01
7.63879E-01
2.70624E-01
5.32957E-02
1.24468E-01
4.45997E-01
9.39235E-01
3.75469E-01
7.85886E-01
4.37336E-01
4.67083E-01
9.41378E-01
1.42369E-01
5.85918E-01

The standard library also includes a random number generator for discrete numbers, which is part of the Ada.Numerics.Discrete_Random package. Since it's a generic package, you have to instantiate it for the desired discrete type. This allows you to specify a range for the generator. In the following example, we create an application that displays random integers between 1 and 10:

Listing 3: show_discrete_random_num.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Discrete_Random;

procedure Show_Discrete_Random_Num is
  subtype Random_Range is Integer range 1 .. 10;
  package R is new
    Ada.Numerics.Discrete_Random (Random_Range);
  use R;
  G : Generator;
  X : Random_Range;
begin
  (continues on next page)
Learning Ada

(continued from previous page)

15 Reset (G);
16 17
18 Put_Line ("Some random numbers between "
19 & Integer'Image (Random_Range'First)
20 & " and "
21 & Integer'Image (Random_Range'Last)
22 & ":");
23
24 for I in 1 .. 15 loop
25 X := Random (G);
26 Put_Line (Integer'Image (X));
27 end loop;
28 end Show_Discrete_Random_Num;

Code block metadata

Project: Courses.Intro To Ada.Standard_Library.Show_Discrete_Random_Num
MD5: 892f6525477f9a2c56f88885de011fba

Runtime output

Some random numbers between 1 and 10:
9 9 1 2 3 7 1 3 8 4 8 1 9 8

Here, package R is instantiated with the Random_Range type, which has a constrained range between 1 and 10. This allows us to control the range used for the random numbers. We could easily modify the application to display random integers between 0 and 20 by changing the specification of the Random_Range type. We can also use floating-point or fixed-point types.

23.3 Complex Types

The Ada.Numerics.Complex_Types package provides support for complex number types and the Ada.Numerics.Complex_Elementary_Functions package provides support for common operations on complex number types, similar to the Ada.Numerics.Elementary_Functions package. Finally, you can use the Ada.Text_IO.Complex_IO package to perform I/O operations on complex numbers. In the following example, we declare variables of the Complex type and initialize them using an aggregate:

Listing 4: show_elem_math.adb

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Complex_Types;
use Ada.Numerics.Complex_Types;

with Ada.Numerics.Complex_Elementary_Functions;
use Ada.Numerics.Complex_Elementary_Functions;

with Ada.Text_IO.Complex_IO;

procedure Show_Elem_Math is

   package C_IO is new
      Ada.Text_IO.Complex_IO (Complex_Types);
   use C_IO;

   X, Y : Complex;
   R, Th : Float;

begin
   X := (2.0, -1.0);
   Y := (3.0, 4.0);

   Put (X);
   Put (" * ");
   Put (Y);
   Put (" is ");
   Put (X * Y);
   New_Line;
   New_Line;

   R := 3.0;
   Th := Pi / 2.0;
   X := Compose_From_Polar (R, Th);
   -- Alternatively:
   -- X := R * Exp ((0.0, Th));
   -- X := R * e ** Complex'(0.0, Th);

   Put ("Polar form: ">
      & Float'Image (R) & " * e**(i * ">
      & Float'Image (Th) & ")");
   New_Line;

   Put ("Modulus of ");
   Put (X);
   Put (" is ");
   Put (Float'Image (abs (X)));
   New_Line;

   Put ("Argument of ");
   Put (X);
   Put (" is ");
   Put (Float'Image (Argument (X)));
   New_Line;

   Put ("Sqrt of ");
   Put (X);
   Put (" is ");
   Put (Sqrt (X));
   New_Line;
end Show_Elem_Math;
As we can see from this example, all the common operators, such as * and +, are available for complex types. You also have typical operations on complex numbers, such as Argument and Exp. In addition to initializing complex numbers in the cartesian form using aggregates, you can do so from the polar form by calling the Compose_From_Polar function.

The Ada.Numerics.Complex_Types and Ada.Numerics.Complex_Elementary_Functions packages provide operations for the Float type. Similar packages are available for Long_Float and Long_Long_Float types. In addition, the Ada.Numerics.Generic_Complex_Types and Ada.Numerics.Generic_Complex_Elementary_Functions packages are generic versions that you can instantiate for custom or pre-defined floating-point types. For example:

with Ada.Numerics.Generic_Complex_Types;
with Ada.Numerics.Generic_Complex_Elementary_Functions;
with Ada.Text_IO.Complex_IO;

procedure Show_Elem_Math is
    package Complex_Types is new
        Ada.Numerics.Generic_Complex_Types (Float);
    use Complex_Types;

    package Elementary_Functions is new
        Ada.Numerics.Generic_Complex_Elementary_Functions
        (Complex_Types);
    use Elementary_Functions;

    package C_IO is new Ada.Text_IO.Complex_IO
        (Complex_Types);
    use C_IO;

    X, Y : Complex;
    R, Th : Float;

23.4 Vector and Matrix Manipulation

The Ada.Numerics.Real_Arrays package provides support for vectors and matrices. It includes common matrix operations such as inverse, determinant, eigenvalues in addition to simpler operators such as matrix addition and multiplication. You can declare vectors and matrices using the Real_Vector and Real_Matrix types, respectively.

The following example uses some of the operations from the Ada.Numerics.Real_Arrays package:
with Ada.Text_IO; use Ada.Text_IO;

with Ada.Numerics.Real_Arrays; use Ada.Numerics.Real_Arrays;

procedure Show_Matrix is

   procedure Put_Vector (V : Real_Vector) is
   begin
      Put (" (");
      for I in V'Range loop
         Put (Float'Image (V (I)) & " ");
      end loop;
      Put_Line (")");
   end Put_Vector;

   procedure Put_Matrix (M : Real_Matrix) is
   begin
      for I in M'Range (1) loop
         Put (" (");
         for J in M'Range (2) loop
            Put (Float'Image (M (I, J)) & " ");
         end loop;
         Put_Line (")");
      end loop;
   end Put_Matrix;

begin
   V1 : Real_Vector := (1.0, 3.0);
   V2 : Real_Vector := (75.0, 11.0);
   M1 : Real_Matrix :=
      ((1.0, 5.0, 1.0),
       (2.0, 2.0, 1.0));
   M2 : Real_Matrix :=
      ((31.0, 11.0, 10.0),
       (34.0, 16.0, 11.0),
       (32.0, 12.0, 10.0),
       (31.0, 13.0, 10.0));
   M3 : Real_Matrix :=
      ((1.0, 2.0),
       (2.0, 3.0));

begin
   Put_Line ("V1");
   Put_Vector (V1);
   Put_Line ("V2");
   Put_Vector (V2);
   Put_Line ("V1 * V2 =");
   Put_Line (" & Float'Image (V1 * V2));
   Put_Line ("V1 * V2 =");
   Put_Matrix (V1 * V2);
   New_Line;
   Put_Line ("M1");
   Put_Matrix (M1);
   Put_Line ("M2");
   Put_Matrix (M2);
   Put_Line ("M2 * Transpose(M1) =");
   Put_Matrix (M2 * Transpose (M1));
   New_Line;
end;
Put_Line ("M3");
Put_Matrix (M3);
Put_Line ("Inverse (M3) =");
Put_Matrix (Inverse (M3));
Put_Line ("abs Inverse (M3) =");
Put_Matrix (abs Inverse (M3));
Put_Line ("Determinant (M3) =");
Put_Line ("
& Float'Image (Determinant (M3));
Put_Line ("Solve (M3, V1) =");
Put_Vector (Solve (M3, V1));
Put_Line ("Eigenvalues (M3) =");
Put_Vector (Eigenvalues (M3));
New_Line;
end Show_Matrix;

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Matrix
MD5: c9df45a742a42bd47e03fbf2d0282238

Runtime output

V1
  ( 1.00000E+00 3.00000E+00 )
V2
  ( 7.50000E+01 1.10000E+01 )
V1 * V2 =
  1.08000E+02
V1 * V2 =
  ( 7.50000E+01 1.10000E+01 )
  ( 2.25000E+02 3.30000E+01 )
M1
  ( 1.00000E+00 5.00000E+00 1.00000E+00 )
  ( 2.00000E+00 2.00000E+00 1.00000E+00 )
M2
  ( 3.10000E+01 1.10000E+01 1.00000E+01 )
  ( 3.40000E+01 1.60000E+01 1.10000E+01 )
  ( 3.20000E+01 1.20000E+01 1.00000E+01 )
  ( 3.10000E+01 1.30000E+01 1.00000E+01 )
M2 * Transpose(M1) =
  ( 9.60000E+01 9.40000E+01 )
  ( 1.25000E+02 1.11000E+02 )
  ( 1.02000E+02 9.80000E+01 )
  ( 1.06000E+02 9.80000E+01 )
M3
  ( 1.00000E+00 2.00000E+00 )
  ( 2.00000E+00 3.00000E+00 )
Inverse (M3) =
  ( -3.00000E+00 2.00000E+00 )
  ( 2.00000E+00 -1.00000E+00 )
abs Inverse (M3) =
  ( 3.00000E+00 2.00000E+00 )
  ( 2.00000E+00 1.00000E+00 )
Determinant (M3) =
  -1.00000E+00
Solve (M3, V1) =
  ( 3.00000E+00 -1.00000E+00 )
Eigenvalues (M3) =
Matrix dimensions are automatically determined from the aggregate used for initialization when you don't specify them. You can, however, also use explicit ranges. For example:

M1 : Real_Matrix (1..2, 1..3) :=
    ((1.0, 5.0, 1.0),
     (2.0, 2.0, 1.0));

The Ada.Numerics.Real_Arrays package implements operations for the **Float** type. Similar packages are available for **Long_Float** and **Long_Long_Float** types. In addition, the Ada.Numerics.Generic_Real_Arrays package is a generic version that you can instantiate with custom floating-point types. For example, the Real_Arrays package can be defined as follows:

```ada
package Real_Arrays is new Ada.Numerics.Generic_Real_Arrays (Float);
```
## 24.1 Appendix A: Generic Formal Types

The following tables contain examples of available formal types for generics:

<table>
<thead>
<tr>
<th>Formal type</th>
<th>Actual type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incomplete type</td>
<td>Any type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T;</code></td>
<td></td>
</tr>
<tr>
<td>Discrete type</td>
<td>Any integer, modular or enumeration type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is (&lt;&gt;);</code></td>
<td></td>
</tr>
<tr>
<td>Range type</td>
<td>Any signed integer type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is range &lt;&gt;;</code></td>
<td></td>
</tr>
<tr>
<td>Modular type</td>
<td>Any modular type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is mod &lt;&gt;;</code></td>
<td></td>
</tr>
<tr>
<td>Floating-point type</td>
<td>Any floating-point type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is digits &lt;&gt;;</code></td>
<td></td>
</tr>
<tr>
<td>Binary fixed-point type</td>
<td>Any binary fixed-point type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is delta &lt;&gt;;</code></td>
<td></td>
</tr>
<tr>
<td>Decimal fixed-point type</td>
<td>Any decimal fixed-point type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is delta &lt;&gt; digits &lt;&gt;;</code></td>
<td></td>
</tr>
<tr>
<td>Definite nonlimited private type</td>
<td>Any nonlimited, definite type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is private;</code></td>
<td></td>
</tr>
<tr>
<td>Nonlimited Private type with discriminant</td>
<td>Any nonlimited type with discriminant</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T (D : DT) is private;</code></td>
<td></td>
</tr>
<tr>
<td>Access type</td>
<td>Any access type for type T</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type A is access T;</code></td>
<td></td>
</tr>
<tr>
<td>Definite derived type</td>
<td>Any concrete type derived from base type B</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is new B;</code></td>
<td></td>
</tr>
<tr>
<td>Limited private type</td>
<td>Any definite type, limited or not</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is limited private;</code></td>
<td></td>
</tr>
<tr>
<td>Incomplete tagged type</td>
<td>Any concrete, definite, tagged type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is tagged;</code></td>
<td></td>
</tr>
<tr>
<td>Definite tagged private type</td>
<td>Any concrete, definite, tagged type</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is tagged private;</code></td>
<td></td>
</tr>
<tr>
<td>Definite tagged limited private type</td>
<td>Any concrete definite tagged type, limited or not</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is tagged limited private;</code></td>
<td></td>
</tr>
<tr>
<td>Definite abstract tagged private type</td>
<td>Any nonlimited, definite tagged type, abstract or concrete</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is abstract tagged private;</code></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Formal type</th>
<th>Actual type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definite abstract tagged limited private type</td>
<td>Any definite tagged type, limited or not, abstract or concrete</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is abstract tagged limited private;</code></td>
<td></td>
</tr>
<tr>
<td>Definite derived tagged type</td>
<td>Any concrete tagged type derived from base type B</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is new B with private;</code></td>
<td></td>
</tr>
<tr>
<td>Definite abstract derived tagged type</td>
<td>Any tagged type derived from base type B abstract or concrete</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is abstract new B with private;</code></td>
<td></td>
</tr>
<tr>
<td>Array type</td>
<td>Any array type with range R containing elements of type T</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type A is array (R) of T;</code></td>
<td></td>
</tr>
<tr>
<td>Interface type</td>
<td>Any interface type T</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is interface;</code></td>
<td></td>
</tr>
<tr>
<td>Limited interface type</td>
<td>Any limited interface type T</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is limited interface;</code></td>
<td></td>
</tr>
<tr>
<td>Task interface type</td>
<td>Any task interface type T</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is task interface;</code></td>
<td></td>
</tr>
<tr>
<td>Synchronized interface type</td>
<td>Any synchronized interface type T</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is synchronized interface;</code></td>
<td></td>
</tr>
<tr>
<td>Protected interface type</td>
<td>Any protected interface type T</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is protected interface;</code></td>
<td></td>
</tr>
<tr>
<td>Derived interface type</td>
<td>Any type T derived from base type B and interface I</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is new B and I with private;</code></td>
<td></td>
</tr>
<tr>
<td>Derived type with multiple interfaces</td>
<td>Any type T derived from base type B and interfaces I1 and I2</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is new B and I1 and I2 with private;</code></td>
<td></td>
</tr>
<tr>
<td>Abstract derived interface type</td>
<td>Any type T derived from abstract base type B and interface I</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is abstract new B and I with private;</code></td>
<td></td>
</tr>
<tr>
<td>Limited derived interface type</td>
<td>Any type T derived from limited base type B and limited interface I</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is limited new B and I with private;</code></td>
<td></td>
</tr>
<tr>
<td>Abstract limited derived interface type</td>
<td>Any type T derived from abstract limited base type B and limited interface I</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is abstract limited new B and I with private;</code></td>
<td></td>
</tr>
<tr>
<td>Synchronized interface type</td>
<td>Any type T derived from synchronized interface SI</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is synchronized new SI with private;</code></td>
<td></td>
</tr>
<tr>
<td>Abstract synchronized interface type</td>
<td>Any type T derived from synchronized interface SI</td>
</tr>
<tr>
<td><strong>Format</strong>: <code>type T is abstract synchronized new SI with private;</code></td>
<td></td>
</tr>
</tbody>
</table>
24.1.1 Indefinite version

Many of the examples above can be used for formal indefinite types:

<table>
<thead>
<tr>
<th>Formal type</th>
<th>Actual type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indefinite incomplete type</td>
<td>Any type</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;);</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite nonlimited private type</td>
<td>Any nonlimited type indefinite or definite</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is private;</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite limited private type</td>
<td>Any type, limited or not, indefinite or definite</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is limited private;</code></td>
<td></td>
</tr>
<tr>
<td>Incomplete indefinite tagged private type</td>
<td>Any concrete tagged type, indefinite or definite</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is tagged;</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite tagged private type</td>
<td>Any concrete, nonlimited tagged type, indefinite or definite</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is tagged private;</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite tagged limited private type</td>
<td>Any concrete tagged type, limited or not, indefinite or definite</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is tagged limited private;</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite abstract tagged private type</td>
<td>Any nonlimited tagged type, indefinite or definite, abstract or concrete</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is abstract tagged private;</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite abstract tagged limited private type</td>
<td>Any tagged type, limited or not, indefinite or definite abstract or concrete</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is abstract tagged limited private;</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite derived tagged type</td>
<td>Any tagged type derived from base type B, indefinite or definite</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is new B with private;</code></td>
<td></td>
</tr>
<tr>
<td>Indefinite abstract derived tagged type</td>
<td>Any tagged type derived from base type B, indefinite or definite abstract or concrete</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>type T (&lt;&gt;) is abstract new B with private;</code></td>
<td></td>
</tr>
</tbody>
</table>

The same examples could also contain discriminants. In this case, `(<>)` is replaced by a list of discriminants, e.g.: `(D: DT)`.

24.2 Appendix B: Containers

The following table shows all containers available in Ada, including their versions (standard, bounded, unbounded, indefinite):

<table>
<thead>
<tr>
<th>Category</th>
<th>Container</th>
<th>Std</th>
<th>Bounded</th>
<th>Unbounded</th>
<th>Indefinite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>Vectors</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>List</td>
<td>Doubly Linked Lists</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Map</td>
<td>Hashed Maps</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Map</td>
<td>Ordered Maps</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Set</td>
<td>Hashed Sets</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>Ordered Sets</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Tree</td>
<td>Multiway Trees</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Generic</td>
<td>Holders</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>Synchronized Queue Interfaces</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>Synchronized Queues</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>Priority Queues</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
Note: To get the correct container name, replace the whitespace by _ in the names above. (For example, Hashed Maps becomes Hashed_Maps.)

The following table presents the prefixing applied to the container name that depends on its version. As indicated in the table, the standard version does not have a prefix associated with it.

<table>
<thead>
<tr>
<th>Version</th>
<th>Naming prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Std</td>
<td></td>
</tr>
<tr>
<td>Bounded</td>
<td>Bounded_</td>
</tr>
<tr>
<td>Unbounded</td>
<td>Unbounded_</td>
</tr>
<tr>
<td>Indefinite</td>
<td>Indefinite_</td>
</tr>
</tbody>
</table>
Part II

Advanced Journey With Ada: A Flight In Progress
Warning: This is work in progress!
Information in this document is subject to change at any time without prior notification.

Note: The code examples in this course use a 50-column limit, which greatly improves the readability of the code on devices with a small screen size. This constraint, however, leads to an unusual coding style. For instance, instead of calling Put_Line in a single line, we have this:

```ada
Put_Line (
" is in the northeast quadrant"
);
```

or this:

```ada
Put_Line (" (X => ",
  & Integer'Image (P.X)
  & "))
```

Note that typical Ada code uses a limit of at least 79 columns. Therefore, please don't take the coding style from this course as a reference!

Note: Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/
  cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

21 http://creativecommons.org/licenses/by-sa/4.0
22 https://learn.adacore.com/zip/learning-ada_code.zip
1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

This course will teach you advanced topics of the Ada programming language. The *Introduction to Ada* (page 5) course is a prerequisite for this course.

This document was written by Gustavo A. Hoffmann, with major contributions from Robert A. Duff. The document also includes contributions from Franco Gasperoni, Gary Dismukes, Patrick Rogers, and Robert Dewar.

These contributions are clearly indicated in the document, together with the original publication source.

Special thanks to Patrick Rogers for all comments and suggestions. In particular, thanks for sharing the training slides on access types: many ideas from those slides were integrated into this course.

This document was reviewed by Patrick Rogers and Tucker Taft.

**CHANGELOG**

Changes are being tracked on the CHANGELOG page.
25.1 Types

25.1.1 Scalar Types

In general terms, scalar types are the most basic types that we can get. As we know, we can classify them as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Discrete</th>
<th>Numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Integer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Real</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Many attributes exist for scalar types. For example, we can use the Image and Value attributes to convert between a given type and a string type. The following table presents the main attributes for scalar types:

<table>
<thead>
<tr>
<th>Category</th>
<th>Attribute</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ranges</td>
<td>First</td>
<td>First value of the discrete subtype's range.</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>Last value of the discrete subtype's range.</td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td>Range of the discrete subtype (corresponds to Subtype'First .. Subtype'Last).</td>
</tr>
<tr>
<td>Iterators</td>
<td>Pred</td>
<td>Predecessor of the input value.</td>
</tr>
<tr>
<td></td>
<td>Succ</td>
<td>Successor of the input value.</td>
</tr>
<tr>
<td>Comparison</td>
<td>Min</td>
<td>Minimum of two values.</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>Maximum of two values.</td>
</tr>
<tr>
<td>String conversion</td>
<td>Image</td>
<td>String representation of the input value.</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Value of a subtype based on input string.</td>
</tr>
</tbody>
</table>

We already discussed some of these attributes in the Introduction to Ada course (in the sections about range and related attributes (page 76) and image attribute (page 13)). In this section, we'll discuss some aspects that have been left out of the previous course.

In the Ada Reference Manual

- 3.5 Scalar types

23 http://www.ada-auth.org/standards/22rm/html/RM-3-5.html
Learning Ada

Ranges

We've seen that the First and Last attributes can be used with discrete types. Those attributes are also available for real types. Here's an example using the Float type and a subtype of it:

Listing 1: show_first_last_real.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_First_Last_Real is
    subtype Norm is Float range 0.0 .. 1.0;
begin
    Put_Line ("Float'First: " & Float'First'Image);
    Put_Line ("Float'Last: " & Float'Last'Image);
    Put_Line ("Norm'First: " & Norm'First'Image);
    Put_Line ("Norm'Last: " & Norm'Last'Image);
end Show_First_Last_Real;
```

Code block metadata

- MD5: 89745a94fbd41a2880ba14e50401acb

Runtime output

- Float'First: -3.40282E+38
- Float'Last: 3.40282E+38
- Norm'First: 0.00000E+00
- Norm'Last: 1.00000E+00

This program displays the first and last values of both the Float type and the Norm subtype. In the case of the Float type, we see the full range, while for the Norm subtype, we get the values we used in the declaration of the subtype (i.e. 0.0 and 1.0).

Predecessor and Successor

We can use the Pred and Succ attributes to get the predecessor and successor of a specific value. For discrete types, this is simply the next discrete value. For example, Pred (2) is 1 and Succ (2) is 3. Let's look at a complete source-code example:

Listing 2: show_succ_pred_discrete.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Succ_Pred_Discrete is
    type State is (Idle, Started, Processing, Stopped);
    constant Machine_State := Started;
    constant Integer := 2;
begin
    Put_Line ("State : " & Machine_State'Image);
    Put_Line ("State'Pred (Machine_State): " & State'Pred (Machine_State)'Image);
    Put_Line ("State'Succ (Machine_State): " & State'Succ (Machine_State)'Image);
    Put_Line ("--------");
end Show_Succ_Pred_Discrete;
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
Put_Line ("I : ");
& I'Image);
Put_Line ("Integer'Pred (I) : ");
& Integer'Pred (I)'Image);
Put_Line ("Integer'Succ (I) : ");
& Integer'Succ (I)'Image);
end Show_Succ_Pred_Discrete;
```

**Code block metadata**

MD5: e11d0f50105864fdc1594b3bb72d927e

**Runtime output**

<table>
<thead>
<tr>
<th>State</th>
<th>: STARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>State'Pred (Machine_State):</td>
<td>IDLE</td>
</tr>
<tr>
<td>State&amp;Succ (Machine_State):</td>
<td>PROCESSING</td>
</tr>
<tr>
<td>I                :</td>
<td>2</td>
</tr>
<tr>
<td>Integer'Pred (I):</td>
<td>1</td>
</tr>
<tr>
<td>Integer'Succ (I):</td>
<td>3</td>
</tr>
</tbody>
</table>

In this example, we use the Pred and Succ attributes for a variable of enumeration type (State) and a variable of Integer type.

We can also use the Pred and Succ attributes with real types. In this case, however, the value we get depends on the actual type we’re using:

- for fixed-point types, the value is calculated using the smallest value (Small), which is derived from the declaration of the fixed-point type;
- for floating-point types, the value used in the calculation depends on representation constraints of the actual target machine.

Let’s look at this example with a decimal type (Decimal) and a floating-point type (My_Float):

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Succ_Pred_Real is
  subtype My_Float is
    Float range 0.0 .. 0.5;

type Decimal is
  delta 0.1 digits 2
  range 0.0 .. 0.5;

  D : Decimal;
  N : My_Float;
begin
  Put_Line ("---- DECIMAL -----");
  Put_Line ("Small: " & Decimal'Small'Image);
  Put_Line ("----- Succ -------");
  D := Decimal'First;
  loop
    Put_Line (D'Image);
    D := Decimal'Succ (D);
```

(continues on next page)
exit when D = Decimal'Last;
end loop;
Put_Line ("----- Pred -------");
D := Decimal'Last;
loop
  Put_Line (D'Image);
  D := Decimal'Pred (D);
  exit when D = Decimal'Last;
end loop;
Put_Line ("=============");

Put_Line ("----- MY_FLOAT ----");
Put_Line ("----- Succ ------");
N := My_Float'First;
for I in 1 .. 5 loop
  Put_Line (N'Image);
  N := My_Float'Succ (N);
end loop;
Put_Line ("----- Pred -------");
for I in 1 .. 5 loop
  Put_Line (N'Image);
  N := My_Float'Pred (N);
end loop;
end Show_Succ_Pred_Real;

---- DECIMAL ----
Small: 1.0000000000000000E-01
----- Succ ------
  0.0
  0.1
  0.2
  0.3
  0.4
----- Pred -------
  0.5
  0.4
  0.3
  0.2
  0.1
=============

----- MY_FLOAT ----
----- Succ ------
  0.00000E+00
  1.40130E-45
  2.80260E-45
  4.20390E-45
  5.60519E-45
----- Pred -------
  7.00649E-45
  5.60519E-45
  4.20390E-45
As the output of the program indicates, the smallest value (see Decimal'Small in the example) is used to calculate the previous and next values of Decimal type.

In the case of the My_Float type, the difference between the current and the previous or next values is 1.40130E-45 (or $2^{-149}$) on a standard PC.

**Scalar To String Conversion**

We've seen that we can use the Image and Value attributes to perform conversions between values of a given subtype and a string:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Image_Value_Attr is
  I : constant Integer := Integer'Value ("42");
begin
  Put_Line (I'Image);
end Show_Image_Value_Attr;
```

The Image and Value attributes are used for the String type specifically. In addition to them, there are also attributes for different string types — namely Wide_String and Wide_Wide_String. This is the complete list of available attributes:

<table>
<thead>
<tr>
<th>Conversion type</th>
<th>Attribute</th>
<th>String type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion to string</td>
<td>Image</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Wide_Image</td>
<td>Wide_String</td>
</tr>
<tr>
<td>Conversion to subtype</td>
<td>Wide_Wide_Image</td>
<td>Wide_Wide_String</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Wide_Value</td>
<td>Wide_String</td>
</tr>
<tr>
<td></td>
<td>Wide_Wide_Value</td>
<td>Wide_Wide_String</td>
</tr>
</tbody>
</table>

We discuss more about Wide_String and Wide_Wide_String in another section (page 495).
**Width attribute**

When converting a value to a string by using the `Image` attribute, we get a string with variable width. We can assess the maximum width of that string for a specific subtype by using the `Width` attribute. For example, `Integer'Width` gives us the maximum width returned by the `Image` attribute when converting a value of `Integer` type to a string of `String` type.

This attribute is useful when we're using bounded strings in our code to store the string returned by the `Image` attribute. For example:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Bounded;

procedure Show_Width_Attr is
  package B_Str is new
    Ada.Strings.Bounded.Generic_Bounded_Length
      (Max => Integer'Width);
  use B_Str;

  Str_I : Bounded_String;

  I : constant Integer := 42;
  J : constant Integer := 103;

begin
  Str_I := To_Bounded_String (I'Image);
  Put_Line ("Value: ", & To_String (Str_I));
  Put_Line ("String Length: ", & Length (Str_I)'Image);
  Put_Line ("----");

  Str_I := To_Bounded_String (J'Image);
  Put_Line ("Value: ", & To_String (Str_I));
  Put_Line ("String Length: ", & Length (Str_I)'Image);

end Show_Width_Attr;
```

In this example, we're storing the string returned by `Image` in the `Str_I` variable of `Bounded_String` type.

Similar to the `Image` and `Value` attributes, the `Width` attribute is also available for string types other than `String`. In fact, we can use:

- the `Wide_Width` attribute for strings returned by `Wide_Image`; and
- the `Wide_Wide_Width` attribute for strings returned by `Wide_Wide_Image`. 
The Base attribute gives us the unconstrained underlying hardware representation selected for a given numeric type. As an example, let's say we declared a subtype of the `Integer` type named `One_To_Ten`:

Listing 6: my_integers.ads
1. package My_Integers is
2. subtype One_To_Ten is Integer
3. range 1 .. 10;
4. end My_Integers;

For further reading...

The Ada standard defines that the minimum range of the `Integer` type is \(-2^{31} + 1 \ldots 2^{31} - 1\). In modern 64-bit systems — where wider types such as `Long_Integer` are defined — the range is at least \(-2^{31} + 1 \ldots 2^{31} - 1\). Therefore, we could think of the `Integer` type as having the following declaration:

\[
\text{type Integer is range } -2^{31} \ldots 2^{31} - 1;
\]

However, even though `Integer` is a predefined Ada type, it's actually a subtype of an anonymous type. That anonymous "type" is the hardware's representation for the numeric type as chosen by the compiler based on the requested range (for the signed integer types) or digits of precision (for floating-point types). In other words, these types are actually subtypes of something that does not have a specific name in Ada, and that is not constrained.

In effect,

```
type Integer is range -2 ** 31 .. 2 ** 31 - 1;
```

is really as if we said this:

```
subtype Integer is Some_Hardware_Type_With_Sufficient_Range
  range -2 ** 31 .. 2 ** 31 - 1;
```

Since the `Some_Hardware_Type_With_Sufficient_Range` type is anonymous and we therefore cannot refer to it in the code, we just say that `Integer` is a type rather than a subtype.

Let's focus on signed integers — as the other numerics work the same way. When we declare a signed integer type, we have to specify the required range, statically. If the compiler cannot find a hardware-defined or supported signed integer type with at least the
range requested, the compilation is rejected. For example, in current architectures, the code below most likely won't compile:

Listing 7: int_def.ads

```ada
package Int_Def is
  type Too_Big_To_Fail is
    range -2 ** 255 .. 2 ** 255 - 1;
end Int_Def;
```

The following example shows how the Base attribute affects the bounds of a variable:

Listing 8: show_base.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with My_Integers; use My_Integers;

procedure Show_Base is
  C : constant One_To_Ten := One_To_Ten'Last;
begin
  Using_Constrained_Subtype : declare
    V : One_To_Ten := C;
  begin
    Put_Line
      ("Increasing value for One_To_Ten...);
    V := One_To_Ten'Succ (V);
  exception
    when others =>
      Put_Line ("Exception raised!");
  end Using_Constrained_Subtype;

  Using_Base : declare
    V : One_To_Ten'Base := C;
  begin
    Put_Line
      ("Increasing value for One_To_Ten'Base...");
    V := One_To_Ten'Succ (V);
  exception
    when others =>
      Put_Line ("Exception raised!");
  end Using_Base;
  Put_Line ("One_To_Ten'Last: "
```

(continues on next page)
In the first block of the example (Using_Constrained_Subtype), we're asking for the next value after the last value of a range — in this case, One_To_Ten'\texttt{Succ} (One_To_Ten'\texttt{Last}). As expected, since the last value of the range doesn't have a successor, a constraint exception is raised.

In the Using_Base block, we're declaring a variable \texttt{V} of One_To_Ten'\texttt{Base} subtype. In this case, the next value exists — because the condition One_To_Ten'\texttt{Last} + 1 <= One_To_Ten'\texttt{Base}'\texttt{Last} is true —, so we can use the Succ attribute without having an exception being raised.

In the following example, we adjust the result of additions and subtractions to avoid constraint errors:

### Listing 9: my_integers.ads

```ada
package My_Integers is

  subtype One_To_Ten is Integer range 1 .. 10;

  function Sat_Add (V1, V2 : One_To_Ten'Base)
    return One_To_Ten;

  function Sat_Sub (V1, V2 : One_To_Ten'Base)
    return One_To_Ten;

end My_Integers;
```

### Listing 10: my_integers.adb

```ada
-- with Ada.Text_IO; use Ada.Text_IO;

package body My_Integers is

  function Saturate (V : One_To_Ten'Base)
    return One_To_Ten is
begin

(continues on next page)
```ada
-- Put_Line("SATURATE " & V'Image);

if V < One_To_Ten'First then
  return One_To_Ten'First;
elsif V > One_To_Ten'Last then
  return One_To_Ten'Last;
else
  return V;
end if;
end Saturate;

function Sat_Add (V1, V2 : One_To_Ten'Base)
return One_To_Ten is
begin
  return Saturate (V1 + V2);
end Sat_Add;

function Sat_Sub (V1, V2 : One_To_Ten'Base)
return One_To_Ten is
begin
  return Saturate (V1 - V2);
end Sat_Sub;
end My_Integers;
```

Listing 11: show_base.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with My_Integers; use My_Integers;

procedure Show_Base is
  type Display_Saturate_Op is (Add, Sub);

  procedure Display_Saturate
  (V1, V2 : One_To_Ten;
   Op   : Display_Saturate_Op)
  is
    Res : One_To_Ten;
  begin
    case Op is
      when Add =>
        Res := Sat_Add (V1, V2);
      when Sub =>
        Res := Sat_Sub (V1, V2);
    end case;
    Put_Line("SATURATE " & Op'Image
      & " (" & V1'Image
      & ", " & V2'Image
      & ") = " & Res'Image);
  end Display_Saturate;

begin
  Display_Saturate (1, 1, Add);
  Display_Saturate (10, 8, Add);
  Display_Saturate (1, 8, Sub);
end Show_Base;
```

Code block metadata
In this example, we're using the Base attribute to declare the parameters of the Sat_Add, Sat_Sub and Saturate functions. Note that the parameters of the Display_Saturate procedure are of One_To_Ten type, while the parameters of the Sat_Add, Sat_Sub and Saturate functions are of the (unconstrained) base subtype (One_To_Ten'Base). In those functions, we perform operations using the parameters of unconstrained subtype and adjust the result — in the Saturate function — before returning it as a constrained value of One_To_Ten subtype.

The code in the body of the My_Integers package contains lines that were commented out — to be more precise, a call to Put_Line call in the Saturate function. If you uncomment them, you'll see the value of the input parameter V (of One_To_Ten'Base type) in the runtime output of the program before it's adapted to fit the constraints of the One_To_Ten subtype.

### 25.1.2 Enumerations

We've introduced enumerations back in the *Introduction to Ada course* (page 51). In this section, we'll discuss a few useful features of enumerations, such as enumeration renaming, enumeration overloading and representation clauses.

**In the Ada Reference Manual**

- 3.5.1 Enumeration Types

#### Enumerations as functions

If you have used programming language such as C in the past, you're familiar with the concept of enumerations being constants with integer values. In Ada, however, enumerations are not integers. In fact, they're actually parameterless functions! Let's consider this example:

```
package Days is

    type Day is (Mon, Tue, Wed,
               Thu, Fri,
               Sat, Sun);

    -- Essentially, we're declaring these functions:
    --
    -- function Mon return Day;
    -- function Tue return Day;
    -- function Wed return Day;

Listing 12: days.ads
```

(continues on next page)

---

In the package Days, we're declaring the enumeration type Day. When we do this, we're essentially declaring seven parameterless functions, one for each enumeration. For example, the Mon enumeration corresponds to function Mon return Day. You can see all seven function declarations in the comments of the example above.

Note that this has no direct relation to how an Ada compiler generates machine code for enumeration. Even though enumerations are parameterless functions, a typical Ada compiler doesn't generate function calls for code that deals with enumerations.

**Enumeration renaming**

The idea that enumerations are parameterless functions can be used when we want to rename enumerations. For example, we could rename the enumerations of the Day type like this:

```ada
with Ada.Text_IO;
use Ada.Text_IO;
with Enumeration_Example;
use Enumeration_Example;

procedure Show_Renaming is
  D1 : constant Day := Mon;
end Show_Renaming;
```

Now, we can use both Monday or Mon to refer to Monday of the Day type:
D2 : constant Day := Monday;
begin
  if D1 = D2 then
    Put_Line ("D1 = D2");
    Put_Line (Day'Image (D1)
      & " = "
      & Day'Image (D2));
  end if;
end Show_Renaming;

Code block metadata
Project: Courses.Advanced_Ada.Data TYPES.TYPES.Enumerations.Enumeration_Renaming
MD5: 2d7177def2c9e9fb11c7dc5e036c3be3

Runtime output
D1 = D2
MON = MON

When running this application, we can confirm that D1 is equal to D2. Also, even though
we've assigned Monday to D2 (instead of Mon), the application displays Mon = Mon, since
Monday is just another name to refer to the actual enumeration (Mon).

Hint
If you just want to have a single (renamed) enumeration visible in your application — and
make the original enumeration invisible —, you can use a separate package. For example:

Listing 15: enumeration_example.ads
package Enumeration_Example is
  type Day is (Mon, Tue, Wed,
    Thu, Fri,
    Sat, Sun);
end Enumeration_Example;

Listing 16: enumeration_renaming.ads
with Enumeration_Example;
package Enumeration_Renaming is
  subtype Day is Enumeration_Example.Day;
  function Monday return Day renames
    Enumeration_Example.Mon;
  function Tuesday return Day renames
    Enumeration_Example.Tue;
  function Wednesday return Day renames
    Enumeration_Example.Wed;
  function Thursday return Day renames
    Enumeration_Example.Thu;
  function Friday return Day renames
    Enumeration_Example.Fri;
  function Saturday return Day renames
    Enumeration_Example.Sat;
  function Sunday return Day renames
(continues on next page)
Listing Ada

(continued from previous page)

Enumeration Example.Sun;

end Enumeration Renaming;

Listing 17: show_renaming.adb

with Ada.Text_IO; use Ada.Text_IO;

with Enumeration Renaming; use Enumeration Renaming;

procedure Show Renaming is
    D1 : constant Day := Monday;
begin
    Put_Line (Day'Image (D1));
end Show Renaming;

Code block metadata

MD5: 87fe75026f0fc118921eaee5fe55a8a

Runtime output

MON

Note that the call to Put_Line still display Mon instead of Monday.

Enumeration overloading

Enumerations can be overloaded. In simple terms, this means that the same name can be used to declare an enumeration of different types. A typical example is the declaration of colors:

Listing 18: colors.ads

package Colors is

    type Color is
        (Salmon,
         Firebrick,
         Red,
         Darkred,
         Lime,
         Forestgreen,
         Green,
         Darkgreen,
         Blue,
         Mediumblue,
         Darkblue);

    type Primary_Color is
        (Red,
         Green,
         Blue);

end Colors;

Code block metadata

290 Chapter 25. Data types
Note that we have Red as an enumeration of type Color and of type Primary_Color. The same applies to Green and Blue. Because Ada is a strongly-typed language, in most cases, the enumeration that we’re referring to is clear from the context. For example:

```
with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Red_Colors is
  C1 : constant Color := Red;
  -- Using Red from Color
  C2 : constant Primary_Color := Red;
  -- Using Red from Primary_Color
begin
  if C1 = Red then
    Put_Line ("C1 = Red");
  end if;
  if C2 = Red then
    Put_Line ("C2 = Red");
  end if;
end Red_Colors;
```

The same logic applies to comparisons such as the one in if C1 = Red: because the type of C1 is defined (Color), it's clear that the Red enumeration is the one of Color type.

### Enumeration subtypes

Note that enumeration overloading is not the same as enumeration subtypes. For example, we could define the following subtype:

```
package Colors.Shades is
  subtype Blue_Shades is
    Colors range Blue .. Darkblue;
end Colors.Shades;
```

## 25.1. Types
In this case, Blue of Blue_Shades and Blue of Colors are the same enumeration.

**Enumeration ambiguities**

A situation where enumeration overloading might lead to ambiguities is when we use them in ranges. For example:

```ada
package Colors is

  type Color is
    (Salmon,
     Firebrick,
     Red,
     Darkred,
     Lime,
     Forestgreen,
     Green,
     Darkgreen,
     Blue,
     Mediumblue,
     Darkblue);

  type Primary_Color is
    (Red,
     Green,
     Blue);

end Colors;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Color_Loop is
begin
  for C in Red .. Blue loop
    -- ERROR: range is ambiguous!
    Put_Line (Color'Image (C));
  end loop;
end Color_Loop;
```

Build output

```
color_loop.adb:6:17: error: ambiguous bounds in range of iteration
color_loop.adb:6:17: error: possible interpretations:
color_loop.adb:6:17: error: type "Primary_Color" defined at colors.ads:16
color_loop.adb:6:17: error: type "Color" defined at colors.ads:3
color_loop.adb:6:17: error: ambiguous bounds in discrete range
color_loop.adb:9:30: error: found type "Primary_Color" defined at colors.ads:16
gprbuild: *** compilation phase failed
```
Here, it's not clear whether the range in the loop is of Color type or of Primary_Color type. Therefore, we get a compilation error for this code example. The next line in the code example — the one with the call to Put_Line — gives us a hint about the developer's intention to refer to the Color type. In this case, we can use qualification — for example, Color'(Red) — to resolve the ambiguity:

**Listing 23: color_loop.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Color_Loop is
begin
  for C in Color'(Red) .. Color'(Blue) loop
    Put_Line (Color'Image (C));
  end loop;
end Color_Loop;
```

**Runtime output**

RED
DARKRED
LIME
FORESTGREEN
GREEN
DARKGREEN
BLUE

Note that, in the case of ranges, we can also rewrite the loop by using a range declaration:

**Listing 24: color_loop.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Color_Loop is
begin
  for C in Color range Red .. Blue loop
    Put_Line (Color'Image (C));
  end loop;
end Color_Loop;
```

**Runtime output**

RED
DARKRED
LIME
FORESTGREEN
GREEN
DARKGREEN
BLUE

25.1. Types
Alternatively, Color range Red .. Blue could be used in a subtype declaration, so we could rewrite the example above using a subtype (such as Red_To_Blue) in the loop:

Listing 25: color_loop.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Colors; use Colors;

procedure Color_Loop is
  subtype Red_To_Blue is Color range Red .. Blue;
begin
  for C in Red_To_Blue loop
    Put_Line (Color'Image (C));
  end loop;
end Color_Loop;
```

Position and Internal Code

As we've said above, a typical Ada compiler doesn't generate function calls for code that deals with enumerations. On the contrary, each enumeration has values associated with it, and the compiler uses those values instead.

Each enumeration has:

• a position value, which is a natural value indicating the position of the enumeration in the enumeration type; and

• an internal code, which, by default, in most cases, is the same as the position value.

Also, by default, the value of the first position is zero, the value of the second position is one, and so on. We can see this by listing each enumeration of the Day type and displaying the value of the corresponding position:

Listing 26: days.ads

```ada
package Days is
type Day is (Mon, Tue, Wed,
             Thu, Fri,
             Sat, Sun);
end Days;
```

Listing 27: show_days.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Days; use Days;

procedure Show_Days is
begin
  for D in Day loop
    Put_Line (Day'Image (D)
              & " position = "
              & Integer'Image (Day'Pos (D)));
    Put_Line (Day'Image (D)
              & " internal code = "
              & Integer'Image
                        (Day'Enum_Rep (D)));
  end loop;
end Show_Days;
```
25.1.3 Definite and Indefinite Subtypes

Indefinite types were mentioned back in the Introduction to Ada course (page 81). In this section, we’ll recapitulate and extend on both definite and indefinite types.

Definite types are the basic kind of types we commonly use when programming applications. For example, we can only declare variables of definite types; otherwise, we get a compilation error. Interestingly, however, to be able to explain what definite types are, we need to first discuss indefinite types.

Indefinite types include:

• unconstrained arrays;
• record types with unconstrained discriminants without defaults.

Let’s see some examples of indefinite types:

Listing 28: unconstrained_types.ads

```ada
package Unconstrained_Types is

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Simple_Record (Extended : Boolean) is
    record
      V : Integer;
      case Extended is
      when False =>
        null;
      when True  =>
        V_Float : Float;
      end case;
    end record;

```

(continues on next page)
In this example, both Integer_Array and Simple_Record are indefinite types.

Important

Note that we cannot use indefinite subtypes as discriminants. For example, the following code won’t compile:

Listing 29: unconstrained_types.ads

```ada
package Unconstrained_Types is
  type Integer_Array is
    array (Positive range <>) of Integer;
  type Simple_Record (Arr : Integer_Array) is
    record
      L : Natural := Arr'Length;
    end record;
end Unconstrained_Types;
```

Build output

unconstrained_types.ads:6:30: error: discriminants must have a discrete or access
__type
gprbuild: *** compilation phase failed

Integer_Array is a correct type declaration — although the type itself is indefinite after the declaration. However, we cannot use it as the discriminant in the declaration of Simple_Record. We could, however, have a correct declaration by using discriminants as access values:

Listing 30: unconstrained_types.ads

```ada
package Unconstrained_Types is
  type Integer_Array is
    array (Positive range <>) of Integer;
  type Integer_Array_Access is
    access Integer_Array;
  type Simple_Record (Arr : Integer_Array_Access) is
    record
      L : Natural := Arr'Length;
    end record;
end Unconstrained_Types;
```
By adding the Integer_Array_Access type and using it in Simple_Record's type declaration, we can indirectly use an indefinite type in the declaration of another indefinite type. We discuss this topic later in another chapter (page 745).

As we've just mentioned, we cannot declare variable of indefinite types:

Listing 31: using_unconstrained_type.adb

```ada
with Unconstrained_Types; use Unconstrained_Types;

procedure Using_Unconstrained_Type is
   A : Integer_Array;
   R : Simple_Record;
begin
   null;
end Using_Unconstrained_Type;
```

As we can see when we try to build this example, the compiler complains about the declaration of A and R because we're trying to use indefinite types to declare variables. The main reason we cannot use indefinite types here is that the compiler needs to know at this point how much memory it should allocate. Therefore, we need to provide the information that is missing. In other words, we need to change the declaration so the type becomes definite. We can do this by either declaring a definite type or providing constraints in the variable declaration. For example:
In this example, we declare the `Integer_Array_5` subtype, which is definite because we're constraining it to a range from 1 to 5, thereby defining the information that was missing in the indefinite type `Integer_Array`. Because we now have a definite type, we can use it to declare the A1 variable. Similarly, we can use the indefinite type `Integer_Array` directly in the declaration of A2 by specifying the previously unknown range.

Similarly, in this example, we declare the `Simple_Record_Ext` subtype, which is definite because we're initializing the record discriminant `Extended`. We can therefore use it in the declaration of the R1 variable. Alternatively, we can simply use the indefinite type `Simple_Record` and specify the information required for the discriminants. This is what we do in the declaration of the R2 variable.

Although we cannot use indefinite types directly in variable declarations, they're very useful to generalize algorithms. For example, we can use them as parameters of a subprogram:

```ada
with Unconstrained_Types; use Unconstrained_Types;
procedure Show_Integer_Array (A : Integer_Array);
```
Listing 34: show_integer_array.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Integer_Array (A : Integer_Array)
is
begin
for I in A'Range loop
   Put_Line (Positive'Image (I)
   & ": 
   & Integer'Image (A (I)));
end loop;
Put_Line ("--------");
end Show_Integer_Array;
```

Listing 35: using_unconstrained_type.adb

```ada
with Unconstrained_Types; use Unconstrained_Types;

with Show_Integer_Array;

procedure Using_Unconstrained_Type is
   A_5 : constant Integer_Array (1 .. 5) :=
      (1, 2, 3, 4, 5);
   A_10 : constant Integer_Array (1 .. 10) :=
      (1, 2, 3, 4, 5, others => 99);
begin
   Show_Integer_Array (A_5);
   Show_Integer_Array (A_10);
end Using_Unconstrained_Type;
```

Code block metadata

Indefinite Types
MD5: 3f744fa5921a55865bc5361ec4c6eb88

Runtime output

```
1: 1
2: 2
3: 3
4: 4
5: 5
--------
1: 1
2: 2
3: 3
4: 4
5: 5
6: 99
7: 99
8: 99
9: 99
10: 99
--------
```

In this particular example, the compiler doesn't know a priori which range is used for the A parameter of Show_Integer_Array. It could be a range from 1 to 5 as used for variable A_5 of the Using_Unconstrained_Type procedure, or it could be a range from 1 to 10 as used for variable A_10, or it could be anything else. Although the parameter A of Show_Integer_Array is unconstrained, both calls to Show_Integer_Array — in Us-
Learning Ada

ing_Unconstrained_Type procedure — use constrained objects.

Note that we could call the Show_Integer_Array procedure above with another unconstrained parameter. For example:

Listing 36: show_integer_array_header.ads

```ada
with Unconstrained_Types; use Unconstrained_Types;

procedure Show_Integer_Array_Header
(AA : Integer_Array;
HH : String);
```

Listing 37: show_integer_array_header.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Show_Integer_Array;

procedure Show_Integer_Array_Header
(AA : Integer_Array;
HH : String)
is
begin
  Put_Line (HH);
  Show_Integer_Array (AA);
end Show_Integer_Array_Header;
```

Listing 38: using_unconstrained_type.adb

```ada
with Unconstrained_Types; use Unconstrained_Types;

procedure Using_Unconstrained_Type is
  A_5 : constant Integer_Array (1 .. 5) :=
       (1, 2, 3, 4, 5);
  A_10 : constant Integer_Array (1 .. 10) :=
       (1, 2, 3, 4, 5, others => 99);
begin
  Show_Integer_Array_Header (A_5,
                           "First example");
  Show_Integer_Array_Header (A_10,
                           "Second example");
end Using_Unconstrained_Type;
```

Code block metadata

- Indefinite_Types
- MD5: dd09f8c4089c6ad4c18410879f80f731

Runtime output

First example
1: 1
2: 2
3: 3
4: 4
5: 5
--------
Second example
1: 1
(continues on next page)
In this case, we're calling the `Show_Integer_Array` procedure with another unconstrained parameter (the `AA` parameter). However, although we could have a long chain of procedure calls using indefinite types in their parameters, we still use a (definite) object at the beginning of this chain. For example, for the `A_5` object, we have this chain:

```
A_5
  ===> Show_Integer_Array_Header (AA => A_5, ...
      
  ===> Show_Integer_Array (A => AA);
```

Therefore, at this specific call to `Show_Integer_Array`, even though `A` is declared as a parameter of indefinite type, the actual argument is of definite type because `A_5` is constrained — and, thus, of definite type.

Note that we can declare variables based on parameters of indefinite type. For example:

**Listing 39: show_integer_array_plus.ads**

```ada
with Unconstrained_Types; use Unconstrained_Types;

procedure Show_Integer_Array_Plus
  (A : Integer_Array;
   V : Integer);
```

**Listing 40: show_integer_array_plus.adb**

```ada
with Show_Integer_Array;

procedure Show_Integer_Array_Plus
  (A : Integer_Array;
   V : Integer)
is
  A_Plus : Integer_Array (A'Range);
begin
  for I in A_Plus'Range loop
    A_Plus (I) := A (I) + V;
  end loop;
  Show_Integer_Array (A_Plus);
end Show_Integer_Array_Plus;
```

**Listing 41: using_unconstrained_type.adb**

```ada
with Unconstrained_Types; use Unconstrained_Types;

with Show_Integer_Array_Plus;

procedure Using_Unconstrained_Type is
```

(continues on next page)
In the `Show_Integer_Array_Plus` procedure, we're declaring `A_Plus` based on the range of `A`, which is itself of indefinite type. However, since the object passed as an argument to `Show_Integer_Array_Plus` must have a constraint, `A_Plus` will also be constrained. For example, in the call to `Show_Integer_Array_Plus` using `A_5` as an argument, the declaration of `A_Plus` becomes `A_Plus : Integer_Array (1 .. 5);`. Therefore, it becomes clear that the compiler needs to allocate five elements for `A_Plus`.

We'll see later how definite and indefinite types apply to formal parameters.

**In the Ada Reference Manual**

- 3.3 Objects and Named Numbers

**Constrained Attribute**

We can use the Constrained attribute to verify whether an object of discriminated type is constrained or not. Let's start our discussion by reusing the `Simple_Record` type from previous examples. In this version of the `Unconstrained_Types` package, we're adding a Reset procedure for the discriminated record type:

```ada
package Unconstrained_Types is

  type Simple_Record
    (Extended : Boolean := False) is
  record
    V : Integer;
    case Extended is
    when False =>
      null;
    when True =>
      V Float : Float;
  end case;
end record;
```

procedure Reset (R : in out Simple_Record);
end Unconstrained_Types;

Listing 43: unconstrained_types.adb

with Ada.Text_IO; use Ada.Text_IO;
package body Unconstrained_Types is

procedure Reset (R : in out Simple_Record) is

Zero_Not_Extended : constant
Simple_Record := (Extended => False, V => 0);

Zero_Extended : constant
Simple_Record := (Extended => True, V => 0, V_Float => 0.0);

begin
Put_Line ("---- Reset: R'Constrained => 
& R'Constrained'Image);
if not R'Constrained then
R := Zero_Extended;
else
if R.Extended then
R := Zero_Extended;
else
R := Zero_Not_Extended;
end if;
end if;
end Reset;
end Unconstrained_Types;

Code block metadata

As the name indicates, the Reset procedure initializes all record components with zero. Note that we use the Constrained attribute to verify whether objects are constrained before assigning to them. For objects that are not constrained, we can simply assign another object to it — as we do with the R := Zero_Extended statement. When an object is constrained, however, the discriminants must match. If we assign an object to R, the discriminant of that object must match the discriminant of R. This is the kind of verification that we do in the else part of that procedure: we check the state of the Extended discriminant before assigning an object to the R parameter.

The Using_Constrained_Attribute procedure below declares two objects of Simple_Record type: R1 and R2. Because the Simple_Record type has a default value for its discriminant, we can declare objects of this type without specifying a value for the discriminant. This is exactly what we do in the declaration of R1. Here, we don't specify any constraints, so that it takes the default value (Extended => False). In the declaration of R2, however, we explicitly set Extended to False:

25.1. Types
Listing 44: using_constrained_attribute.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Unconstrained_Types; use Unconstrained_Types;

procedure Using_Constrained_Attribute is
   R1 : Simple_Record;
   R2 : Simple_Record (Extended => False);

   procedure Show_Rs is
      begin
         Put_Line ("R1'Constrained => " & R1'Constrained'Image);
         Put_Line ("R1.Extended => " & R1.Extended'Image);
         Put_Line ("--");
         Put_Line ("R2'Constrained => " & R2.Constrained'Image);
         Put_Line ("R2.Extended => " & R2.Extended'Image);
         Put_Line ("-------------");
      end Show_Rs;

      begin
         Show_Rs;
         Reset (R1);
         Reset (R2);
         Put_Line ("-------------");
      end Show_Rs;

   end Using_Constrained_Attribute;
```

When we run this code, the user messages from Show_Rs indicate to us that R1 is not constrained, while R2 is constrained. Because we declare R1 without specifying a value for the Extended discriminant, R1 is not constrained. In the declaration of R2, on the other hand, the explicit value for the Extended discriminant makes this object constrained. Note that, for both R1 and R2, the value of Extended is **False** in the declarations.
As we were just discussing, the Reset procedure includes checks to avoid mismatches in
discriminants. When we don't have those checks, we might get exceptions at runtime. We
can force this situation by replacing the implementation of the Reset procedure with the
following lines:

```ada
begin
  Put_Line ("---- Reset: R'Constrained => 
    & R'Constrained'Image);
  R := Zero_Extended;
end Reset;
```

Running the code now generates a runtime exception:

```ada
raised CONSTRAINT_ERROR : unconstrained_types.adb:12 discriminant check failed
```

This exception is raised during the call to Reset (R2). As see in the code, R2 is con-
strained. Also, its Extended discriminant is set to **False**, which means that it doesn't have
the V_Float component. Therefore, R2 is not compatible with the constant Zero_Extended
object, so we cannot assign Zero_Extended to R2. Also, because R2 is constrained, its
Extended discriminant cannot be modified.

The behavior is different for the call to Reset (R1), which works fine. Here, when we
pass R1 as an argument to the Reset procedure, its Extended discriminant is **False** by
default. Thus, R1 is also not compatible with the Zero_Extended object. However, because
R1 is not constrained, the assignment modifies R1 (by changing the value of the Extended
discriminant). Therefore, with the call to Reset, the Extended discriminant of R1 changes
from **False** to **True**.

**In the Ada Reference Manual**

- 3.7.2 Operations of Discriminated Types

25.1.4 Incomplete types

Incomplete types — as the name suggests — are types that have missing information in
their declaration. This is a simple example:

```ada
type Incomplete;
```

Because this type declaration is incomplete, we need to provide the missing information at
some later point. Consider the incomplete type R in the following example:

```ada
package Incomplete_Type_Example is
  type R is record
    I : Integer;
  end record;
end Incomplete_Type_Example;
```

---

25.1. Types

---

http://www.ada-auth.org/standards/22rm/html/RM-3-7-2.html
The first declaration of type R is incomplete. However, in the second declaration of R, we specify that R is a record. By providing this missing information, we're completing the type declaration of R.

It's also possible to declare an incomplete type in the private part of a package specification and its complete form in the package body. Let's rewrite the example above accordingly:

```
package Incomplete_Type_Example
is
  private
    type R;  -- Incomplete type declaration!
end Incomplete_Type_Example;
```

```
package body Incomplete_Type_Example
is
  type R is record
    I : Integer;
  end record;  -- type R is now complete!
end Incomplete_Type_Example;
```

A typical application of incomplete types is to create linked lists using access types based on those incomplete types. This kind of type is called a recursive type. For example:

```
package Linked_List_Example
is
  type Integer_List;
  type Next is access Integer_List;

  type Integer_List is record
    I : Integer;
    N : Next;
  end record;
end Linked_List_Example;
```

Here, the N component of Integer_List is essentially giving us access to the next element.
of `Integer_List` type. Because the `Next` type is both referring to the `Integer_List` type and being used in the declaration of the `Integer_List` type, we need to start with an incomplete declaration of the `Integer_List` type and then complete it after the declaration of `Next`.

Incomplete types are useful to declare *mutually dependent types* (page 418), as we'll see later on. Also, we can also have formal incomplete types, as we'll discuss later.

**In the Ada Reference Manual**
- 3.10.1 Incomplete Type Declarations

### 25.1.5 Type view

Ada distinguishes between the partial and the full view of a type. The full view is a type declaration that contains all the information needed by the compiler. For example, the following declaration of type `R` represents the full view of this type:

```ada
package Full_View is
  -- Full view of the R type:
  type R is record
    I : Integer;
  end record;
end Full_View;
```

As soon as we start applying encapsulation and information hiding — via the `private` keyword — to a specific type, we are introducing a partial view and making only that view compile-time visible to clients. Doing so requires us to introduce the private part of the package (unless already present). For example:

```ada
package Partial_Full_Views is
  -- Partial view of the R type:
  type R is private;
private
  -- Full view of the R type:
  type R is record
    I : Integer;
  end record;
end Partial_Full_Views;
```

**25.1. Types**
As indicated in the example, the `type R is private` declaration is the partial view of the R type, while the `type R is record [...]` declaration in the private part of the package is the full view.

Although the partial view doesn’t contain the full type declaration, it contains very important information for the users of the package where it’s declared. In fact, the partial view of a private type is all that users actually need to know to effectively use this type, while the full view is only needed by the compiler.

In the previous example, the partial view indicates that R is a private type, which means that, even though users cannot directly access any information stored in this type — for example, read the value of the I component of R —, they can use the R type to declare objects. For example:

```ada
with Partial_Full_Views; use Partial_Full_Views;

procedure Main is
  -- Partial view of R indicates that
  -- R exists as a private type, so we
  -- can declare objects of this type:
  C : R;

begin
  -- But we cannot directly access any
  -- information declared in the full
  -- view of R:
  --
  -- C.I := 42;
  --
  null;
end Main;
```

In many cases, the restrictions applied to the partial and full views must match. For example, if we declare a limited type in the full view of a private type, its partial view must also be limited:

```ada
package Limited_Private_Example is
  -- Partial view must be limited,
  -- since the full view is limited.
  type R is limited private;

private
  type R is limited record
    I : Integer;
  end record;
```

(continues on next page)
There are, however, situations where the full view may contain additional requirements that aren’t mentioned in the partial view. For example, a type may be declared as non-tagged in the partial view, but, at the same time, be tagged in the full view:

```ada
package Tagged_Full_View_Example is
  -- Partial view using non-tagged type:
  type R is private;

private
  -- Full view using tagged type:
  type R is tagged record
    I : Integer;
  end record;

end Tagged_Full_View_Example;
```

In this case, from a user's perspective, the R type is non-tagged, so that users cannot use any object-oriented programming features for this type. In the package body of Tagged_Full_View_Example, however, this type is tagged, so that all object-oriented programming features are available for subprograms of the package body that make use of this type. Again, the partial view of the private type contains the most important information for users that want to declare objects of this type.

**In the Ada Reference Manual**

- 7.3 Private Types and Private Extensions

**Non-Record Private Types**

Although it’s very common to declare private types as record types, this is not the only option. In fact, we could declare any type in the full view — scalars, for example —, so we could declare a "private integer" type:

```ada
package Private_Integers is
  -- Partial view of private Integer type:

end Private_Integers;
```

[28](http://www.ada-auth.org/standards/22rm/html/RM-7-3.html)
type Private_Integer is private;

private

-- Full view of private Integer type:
type Private_Integer is new Integer;

end Private_Integers;

This code compiles as expected, but isn't very useful. We can improve it by adding operators to it, for example:

Listing 55: private_integers.ads

package Private_Integers is

-- Partial view of private Integer type:
type Private_Integer is private;

function "+" (Left, Right : Private_Integer)
    return Private_Integer;

private

-- Full view of private Integer type:
type Private_Integer is new Integer;

end Private_Integers;

Listing 56: private_integers.adb

package body Private_Integers is

function "+" (Left, Right : Private_Integer)
    return Private_Integer
is
    Res : constant Integer :=
        Integer (Left) + Integer (Right);
    -- Note that we're converting Left
    -- and Right to Integer, which calls
    -- the "+" operator of the Integer
    -- type. Writing "Left + Right" would
    -- have called the "+" operator of
    -- Private_Integer, which leads to
    -- recursive calls, as this is the
    -- operator we're currently in.

begin
    return Private_Integer (Res);
end "+";

end Private_Integers;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: f1fcbed95e0f66a6f67d1bfd9ba9df1c

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: ac161cb5debfe16465c45949cf682d7
Now, let’s use the new operator in a test application:

```
Listing 57: show_private_integers.adb

with Private_Integers; use Private_Integers;
procedure Show_Private_Integers is
  A, B : Private_Integer;
begin
  A := A + B;
end Show_Private_Integers;
```

In this example, we use the + operator as if we were adding two common integer variables of Integer type.

### Unconstrained Types

There are, however, some limitations: we cannot use unconstrained types such as arrays or even discriminants for arrays in the same way as we did for scalars. For example, the following declarations won’t work:

```
Listing 58: private_arrays.ads

package Private_Arrays is
  type Private_Unconstrained_Array is private;
  type Private_Constrained_Array (L : Positive) is private;

  type Integer_Array is
    array (Positive range <>) of Integer;
  type Private_Unconstrained_Array is
    array (Positive range <>) of Integer;
  type Private_Constrained_Array (L : Positive) is
    array (1 .. 2) of Integer;

end Private_Arrays;
```
Completing the private type with an unconstrained array type in the full view is not allowed because clients could expect, according to their view, to declare objects of the type. But doing so would not be allowed according to the full view. So this is another case of the partial view having to present clients with a sufficiently true view of the type’s capabilities.

One solution is to rewrite the declaration of Private_Constrained_Array using a record type:

```
Listing 59: private_arrays.ads

package Private_Arrays is

  type Private_Constrained_Array
    (L : Positive) is private;

private

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Private_Constrained_Array
    (L : Positive) is
    record
      Arr : Integer_Array (1 .. 2);
    end record;

end Private_Arrays;
```

```
Listing 60: declare_private_array.adb

with Private_Arrays; use Private_Arrays;

procedure Declare_Private_Array is
  Arr : Private_Constrained_Array (5);
begin
  null;
end Declare_Private_Array;
```

Now, the code compiles fine — but we had to use a record type in the full view to make it work.
Another solution is to make the private type indefinite. In this case, the client's partial view would be consistent with a completion as an indefinite type in the private part:

```
package Private_Arrays is
  type Private_Constrained_Array (<>) is private;
  function Init (L : Positive) return Private_Constrained_Array;
private
  type Private_Constrained_Array is array (Positive range <>) of Integer;
end Private_Arrays;
```

```
package body Private_Arrays is
  function Init (L : Positive) return Private_Constrained_Array is
    PCA : Private_Constrained_Array (1 .. L);
  begin
    return PCA;
  end Init;
end Private_Arrays;
```

```
with Private_Arrays; use Private_Arrays;
procedure Declare_Private_Array is
  Arr : Private_Constrained_Array := Init (5);
begin
  null;
end Declare_Private_Array;
```

The bounds for the object's declaration come from the required initial value when an object is declared. In this case, we initialize the object with a call to the Init function.
25.1.6 Type conversion

An important operation when dealing with objects of different types is type conversion, which we already discussed in the Introduction to Ada course (page 56). In fact, we can convert an object Obj_X of an operand type X to a similar, closely related target type Y by simply indicating the target type: Y (Obj_X). In this section, we discuss type conversions for different kinds of types.

Ada distinguishes between two kinds of conversion: value conversion and view conversion. The main difference is the way how the operand (argument) of the conversion is evaluated:

- in a value conversion, the operand is evaluated as an expression (page 581);
- in a view conversion, the operand is evaluated as a name.

In other words, we cannot use expressions such as 2 * A in a view conversion, but only A. In a value conversion, we could use both forms.

In the Ada Reference Manual

- 4.6 Type Conversions

Value conversion

Value conversions are possible for various types. In this section, we see some examples, starting with types derived from scalar types up to array conversions.

Root and derived types

Let’s start with the conversion between a scalar type and its derived types. For example, we can convert back-and-forth between the Integer type and the derived Int type:

```
package Custom_Integers is

    type Int is new Integer
        with Dynamic_Predicate => Int /= 0;

    function Double (I : Integer) return Integer is
        (I * 2);

end Custom_Integers;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Integers; use Custom_Integers;

procedure Show_Conversion is
    Int_Var : Int := 1;
    Integer_Var : Integer := 2;
begin
    -- Int to Integer conversion
    Integer_Var := Integer (Int_Var);
```

http://www.ada-auth.org/standards/22rm/html/RM-4-6.html
Put_Line ("Integer_Var : " & Integer_Var'Image);

-- Int to Integer conversion
-- as an actual parameter
Integer_Var := Double (Integer (Int_Var));

Put_Line ("Integer_Var : " & Integer_Var'Image);

-- Integer to Int conversion
-- using an expression
Int_Var := Int (Integer_Var * 2);

Put_Line ("Int_Var : " & Int_Var'Image);
end Show_Conversion;

Code block metadata
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Root_Derived_Type_Conversion
MD5: 7cd324f308edc34de3bc4bcccce63f1ee

Runtime output
Integer_Var : 1
Integer_Var : 2
Int_Var : 4

In the Show_Conversion procedure from this example, we first convert from Int to Integer. Then, we do the same conversion while providing the resulting value as an actual parameter for the Double function. Finally, we convert the Integer_Var * 2 expression from Integer to Int.

Note that the converted value must conform to any constraints that the target type might have. In the example above, Int has a predicate that dictates that its value cannot be zero. This (dynamic) predicate is checked at runtime, so an exception is raised if it fails:

Listing 66: show_conversion.adb

with Ada.Text_IO; use Ada.Text_IO;
with Custom_Integers; use Custom_Integers;

procedure Show_Conversion is
  Int_Var : Int;
  Integer_Var : Integer;
begin
  Integer_Var := 0;
  Int_Var := Int (Integer_Var);
  Put_Line ("Int_Var : 
                  " & Int_Var'Image);
end Show_Conversion;

Code block metadata
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Root_Derived_Type_Conversion
MD5: 4150cdfffd4c1fed39fa1728a77fa599f

25.1. Types
Learning Ada

Runtime output

raised ADAASSERTIONSASSERTION_ERROR DynamicPredicate failed at show_conversion.adb:9

In this case, the conversion from Integer to Int fails because, while zero is a valid integer value, it doesn't obey Int's predicate.

Numeric type conversion

A typical conversion is the one between integer and floating-point values. For example:

Listing 67: show_conversion.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Conversion is
  F : Float := 1.0;
  I : Integer := 2;
begin
  I := Integer (F);
  Put_Line ("I : 
          & I'Image);
  I := 4;
  F := Float (I);
  Put_Line ("F : 
          & F'Image);
end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Numeric_Type_Conversion
MD5: f64649c786377617b0bc9ff49475ba55

Runtime output

```
I : 1
F : 4.00000E+00
```

Also, we can convert between fixed-point types and floating-point or integer types:

Listing 68: fixed_point_defs.ads

```
package Fixed_Point_Defs is
  S : constant := 32;
  Exp : constant := 15;
  D : constant := 2.0 ** (S + Exp + 1);

  type TQ15_31 is delta D
    range -1.0 * 2.0 ** Exp ..
          1.0 * 2.0 ** Exp - D;

  pragma Assert (TQ15_31'Size = S);
end Fixed_Point_Defs;
```
Listing 69: show_conversion.adb

```ada
with Fixed_Point_Defs; use Fixed_Point_Defs;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Conversion is
    F : Float;
    FP : TQ15_31;
    I : Integer;

    begin
        FP := TQ15_31 (10.25);
        I := Integer (FP);

        Put_Line ("FP : " & FP'Image);
        Put_Line ("I : " & I'Image);

        I := 128;
        FP := TQ15_31 (I);
        F := Float (FP);

        Put_Line ("FP : " & FP'Image);
        Put_Line ("F : " & F'Image);
    end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Numeric_Type_Conversion
MD5: 70714ba396b03469397b982e0299561

Runtime output

FP : 10.25000
I : 10
FP : 128.00000
F : 1.280000E+02

As we can see in the examples above, converting between different numeric types works in all directions. (Of course, rounding is applied when converting from floating-point to integer types, but this is expected.)

Enumeration conversion

We can also convert between an enumeration type and a type derived from it:

Listing 70: custom_enumerations.ads

```ada
package Custom_Enumerations is

    type Priority is (Low, Mid, High);

    type Important_Priority is new
        Priority range Mid .. High;

end Custom_Enumerations;
```
Listing 71: show_conversion.adb

with Ada.Text_IO;  use Ada.Text_IO;
with Custom_Enumerations; use Custom_Enumerations;

procedure Show_Conversion is
  P : Priority := Low;
  IP : Important_Priority := High;
begin
  P := Priority (IP);
  Put_Line ("P: " & P'Image);
  P := Mid;
  IP := Important_Priority (P);
  Put_Line ("IP: " & IP'Image);
end Show_Conversion;

In this example, we have the Priority type and the derived type Important_Priority. As expected, the conversion works fine when the converted value is in the range of the target type. If not, an exception is raised:

Listing 72: show_conversion.adb

with Ada.Text_IO;  use Ada.Text_IO;
with Custom_Enumerations; use Custom_Enumerations;

procedure Show_Conversion is
  P : Priority;
  IP : Important_Priority;
begin
  P := Low;
  IP := Important_Priority (P);
  Put_Line ("IP: " & IP'Image);
end Show_Conversion;

Build output
Array conversion

Similarly, we can convert between array types. For example, if we have the array type `Integer_Array` and its derived type `Derived_Integer_Array`, we can convert between those array types:

```ada
package Custom_Arrays is
  type Integer_Array is
    array (Positive range <>) of Integer;
  type Derived_Integer_Array is new Integer_Array;
end Custom_Arrays;
```

```ada
procedure Show_Conversion is
  subtype Common_Range is Positive range 1 .. 3;
  AI : Integer_Array (Common_Range);
  AI_D : Derived_Integer_Array (Common_Range);
begin
  AI_D := [1, 2, 3];
  AI := Integer_Array (AI_D);
  Put_Line ("AI: 
            & AI'Image);
  AI := [4, 5, 6];
  AI_D := Derived_Integer_Array (AI);
  Put_Line ("AI D: 
            & AI_D'Image);
end Show_Conversion;
```

Code block metadata

25.1. Types
Learning Ada

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Array_Type_Conversion
MD5: e0a9fd519685b418a06dc7a3d0dab1c0

Runtime output

AI: [ 1, 2, 3]
AI_D: [ 4, 5, 6]

Note that both arrays must have the same number of components in order for the conversion to be successful. (Sliding is fine, though.) In this example, both arrays have the same range: Common_Range.

We can also convert between array types that aren't derived one from the other. As long as the components and the index subtypes are of the same type, the conversion between those types is possible. To be more precise, these are the requirements for the array conversion to be accepted:

- The component types must be the same type.
- The index types (or subtypes) must be the same or, at least, convertible.
- The dimensionality of the arrays must be the same.
- The bounds must be compatible (but not necessarily equal).

Converting between different array types can be very handy, especially when we're dealing with array types that were not declared in the same package. For example:

Listing 75: custom_arrays_1.ads

```
package Custom_Arrays_1 is
  type Integer_Array_1 is array (Positive range <> ) of Integer;
  type Float_Array_1 is array (Positive range <> ) of Float;
end Custom_Arrays_1;
```

Listing 76: custom_arrays_2.ads

```
package Custom_Arrays_2 is
  type Integer_Array_2 is array (Positive range <> ) of Integer;
  type Float_Array_2 is array (Positive range <> ) of Float;
end Custom_Arrays_2;
```

Listing 77: show_conversion.adb

```
pragma Ada_2022;
with Ada.Text_IO;   use Ada.Text_IO;
with Custom_Arrays_1; use Custom_Arrays_1;
with Custom_Arrays_2; use Custom_Arrays_2;
```
procedure Show_Conversion is
  subtype Common_Range is Positive range 1 .. 3;
  AI_1 : Integer_Array_1 (Common_Range);
  AI_2 : Integer_Array_2 (Common_Range);
  AF_1 : Float_Array_1 (Common_Range);
  AF_2 : Float_Array_2 (Common_Range);
begin
  AI_2 := [1, 2, 3];
  AI_1 := Integer_Array_1 (AI_2);
  Put_Line ("AI_1: "
            & AI_1'Image);
  AI_1 := [4, 5, 6];
  AI_2 := Integer_Array_2 (AI_1);
  Put_Line ("AI_2: "
            & AI_2'Image);

  -- ERROR: Cannot convert arrays whose
  -- components have different types:
  --
  -- AF_1 := Float_Array_1 (AI_1);
  --
  -- Instead, use array aggregate where each
  -- component is converted from integer to
  -- float:
  --
  AF_1 := [for I in AF_1'Range =>
            Float (AI_1 (I))];
  Put_Line ("AF_1: "
            & AF_1'Image);

  AF_2 := Float_Array_2 (AF_1);
  Put_Line ("AF_2: "
            & AF_2'Image);
end Show_Conversion;

As we can see in this example, the fact that Integer_Array_1 and Integer_Array_2 have
the same component type (Integer) allows us to convert between them. The same applies
to the Float_Array_1 and Float_Array_2 types.
A conversion is not possible when the component types don't match. Even though we can convert between integer and floating-point types, we cannot convert an array of integers to an array of floating-point directly. Therefore, we cannot write a statement such as \( \text{AF}_1 := \text{Float}_{\text{Array}}_1 (\text{AI}_1) \).

However, when the components don't match, we can of course implement the array conversion by converting the individual components. For the example above, we used an iterated component association in an array aggregate: \([\text{for } I \text{ in } \text{AF}_1'\text{Range} => \text{Float} (\text{AI}_1 (I))]\). (We discuss this topic later in another chapter (page 450).)

We may also encounter array types originating from the instantiation of generic packages. In this case as well, we can use array conversions. Consider the following generic package:

```
Listing 78: custom_arrays.ads

generic
    type T is private;
package Custom_Arrays is
    type T_Array is
        array (Positive range <>) of T;
end Custom_Arrays;
```

We could instantiate this generic package and reuse parts of the previous code example:

```
Listing 79: show_conversion.adb

pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Arrays;

procedure Show_Conversion is
    package CA_Int_1 is
        new Custom_Arrays (T => Integer);
    package CA_Int_2 is
        new Custom_Arrays (T => Integer);
    subtype Common_Range is Positive range 1 .. 3;
    AI_1 : CA_Int_1.T_Array (Common_Range);
    AI_2 : CA_Int_2.T_Array (Common_Range);
begin
    AI_2 := [1, 2, 3];
    AI_1 := CA_Int_1.T_Array (AI_2);
    Put_Line ("AI_1: "
              & AI_1'Image);
    AI_1 := [4, 5, 6];
    AI_2 := CA_Int_2.T_Array (AI_1);
    Put_Line ("AI_2: "
              & AI_2'Image);
end Show_Conversion;
```

We may also encounter array types originating from the instantiation of generic packages. In this case as well, we can use array conversions. Consider the following generic package:
As we can see in this example, each of the instantiated CA_Int_1 and CA_Int_2 packages has a T_Array type. Even though these T_Array types have the same name, they’re actually completely unrelated types. However, we can still convert between them in the same way as we did in the previous code examples.

**View conversion**

As mentioned before, view conversions just allow names to be converted. Thus, we cannot use expressions in this case.

Note that a view conversion never changes the value during the conversion. We could say that a view conversion is simply making us view an object from a different angle. The object itself is still the same for both the original and the target types.

For example, consider this package:

```ada
package Some_Tagged_Types is

  type T is tagged record
  A : Integer;
  end record;

  type T_Derived is new T with record
  B : Float;
  end record;

  Obj : T_Derived;

end Some_Tagged_Types;
```

Here, Obj is an object of type T_Derived. When we view this object, we notice that it has two components: A and B. However, we could view this object as being of type T. From that perspective, this object only has one component: A. (Note that changing the perspective doesn't change the object itself.) Therefore, a view conversion from T_Derived to T just makes us view the object Obj from a different angle.

In this sense, a view conversion changes the view of a given object to the target type's view, both in terms of components that exist and operations that are available. It doesn't really change anything at all in the value itself.

There are basically two kinds of view conversions: the ones using tagged types and the ones using untagged types. We discuss these kinds of conversion in this section.
View conversion of tagged types

A conversion between tagged types is a view conversion. Let's consider a typical code example that declares one, two and three-dimensional points:

Listing 81: points.ads

```ada
package Points is

  type Point_1D is tagged record
    X : Float;
  end record;

  procedure Display (P : Point_1D);

  type Point_2D is new Point_1D with record
    Y : Float;
  end record;

  procedure Display (P : Point_2D);

  type Point_3D is new Point_2D with record
    Z : Float;
  end record;

  procedure Display (P : Point_3D);

end Points;
```

Listing 82: points.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Points is

  procedure Display (P : Point_1D) is
  begin
    Put_Line ("(X => ", P.X'Image &, ")");
    end Display;

  procedure Display (P : Point_2D) is
  begin
    Put_Line ("(X => ", P.X'Image & ", Y => ", P.Y'Image & ")");
    end Display;

  procedure Display (P : Point_3D) is
  begin
    end Display;

end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Type_Conversion
MD5: 0acc05ae2310ab4ba038dfdb6bae0495

We can use the types from the Points package and convert between each other:
Listing 83: show_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Points; use Points;

procedure Show_Conversion is
  P_1D : Point_1D;
  P_3D : Point_3D;
begin
  P_3D := (X => 0.1, Y => 0.5, Z => 0.3);
  P_1D := Point_1D (P_3D);
  Put ("P_3D : ");
  Display (P_3D);
  Put ("P_1D : ");
  Display (P_1D);
end Show_Conversion;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Type_Conversion
MD5: fb8e07c8f2399cfae935179d8f413150

**Runtime output**

```
P_3D : (X => 1.00000E-01, Y => 5.00000E-01, Z => 3.00000E-01)
P_1D : (X => 1.00000E-01)
```

In this example, as expected, we're able to convert from the Point_3D type (which has three components) to the Point_1D type, which has only one component.

**View conversion of untagged types**

For untagged types, a view conversion is the one that happens when we have an object of an untagged type as an actual parameter for a formal in out or out parameter.

Let's see a code example. Consider the following simple procedure:

Listing 84: double.ads

```ada
procedure Double (X : in out Float);
```

Listing 85: double.adb

```ada
procedure Double (X : in out Float) is
begin
  X := X * 2.0;
end Double;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_Conversion
MD5: 31f4409d9faeaf213c5935179d8f413150

The Double procedure has an in out parameter of Float type. We can call this procedure using an integer variable I as the actual parameter. For example:
**Listing 86: show_conversion.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Double;

procedure Show_Conversion is
  I : Integer;
begin
  I := 2;
  Put_Line ("I : " & I'Image);

  -- Calling Double with
  -- Integer parameter:
  Double (Float (I));
  Put_Line ("I : " & I'Image);
end Show_Conversion;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_Conversion
MD5: 2256d3c120d569789dcd4c9959ed9d0f

**Runtime output**

I :  2
I :  4

In this case, the `Float (I)` conversion in the call to `Double` creates a temporary floating-point variable. This is the same as if we had written the following code:

**Listing 87: show_conversion.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Double;

procedure Show_Conversion is
  I : Integer;
begin
  I := 2;
  Put_Line ("I : " & I'Image);

  declare
  F : Float := Float (I);
  begin
  Double (F);
  I := Integer (F);
  end;
  Put_Line ("I : " & I'Image);
end Show_Conversion;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_Conversion
MD5: 3b90caf789952710ece42141a7b60968

**Runtime output**
In this sense, the view conversion that happens in Double (Float (I)) can be considered syntactic sugar, as it allows us to elegantly write two conversions in a single statement.

**Implicit conversions**

Implicit conversions are only possible when we have a type \( T \) and a subtype \( S \) related to the \( T \) type. For example:

Listing 88: custom_integers.ads

```ada
package Custom_Integers is

  type Int is new Integer
  with Dynamic_Predicate => Int /= 0;

  subtype Sub_Int_1 is Integer
  with Dynamic_Predicate => Sub_Int_1 /= 0;

  subtype Sub_Int_2 is Sub_Int_1
  with Dynamic_Predicate => Sub_Int_2 /= 1;

end Custom_Integers;
```

Listing 89: show_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Integers; use Custom_Integers;

procedure Show_Conversion is
  Int_Var : Int;
  Sub_Int_1_Var : Sub_Int_1;
  Sub_Int_2_Var : Sub_Int_2;
  Integer_Var : Integer;

begin
  Integer_Var := 5;
  Int_Var := Int (Integer_Var);

  Put_Line ("Int_Var : ",
            & Int_Var'Image);

  -- Implicit conversions:
  -- no explicit conversion required!
  Sub_Int_1_Var := Integer_Var;
  Sub_Int_2_Var := Integer_Var;

  Put_Line ("Sub_Int_1_Var : ",
            & Sub_Int_1_Var'Image);
  Put_Line ("Sub_Int_2_Var : ",
            & Sub_Int_2_Var'Image);

end Show_Conversion;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Implicit_Subtype_Conversion
MD5: dbbe498fa66701ca94f48119b1bc1a91
Learning Ada

Runtime output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Int_Var</td>
<td>5</td>
</tr>
<tr>
<td>Sub_Int_1_Var</td>
<td>5</td>
</tr>
<tr>
<td>Sub_Int_2_Var</td>
<td>5</td>
</tr>
</tbody>
</table>

In this example, we declare the Int type and the Sub_Int_1 and Sub_Int_2 subtypes:

- the Int type is derived from the Integer type,
- Sub_Int_1 is a subtype of the Integer type, and
- Sub_Int_2 is a subtype of the Sub_Int_1 subtype.

We need an explicit conversion when converting between the Integer and Int types. However, as the conversion is implicit for subtypes, we can simply write Sub_Int_1_Var := Integer_Var; (Of course, writing the explicit conversion Sub_Int_1(Integer_Var) in the assignment is possible as well.) Also, the same applies to the Sub_Int_2 subtype: we can write an implicit conversion in the Sub_Int_2_Var := Integer_Var; statement.

Conversion of other types

For other kinds of types, such as records, a direct conversion as we've seen so far isn't possible. In this case, we have to write a conversion function ourselves. A common convention in Ada is to name this function To_TypeName. For example, if we want to convert from any type to Integer or Float, we implement the To_Integer and To_Float functions, respectively. (Obviously, because Ada supports subprogram overloading, we can have multiple To_TypeName functions for different operand types.)

Let's see a code example:

Listing 90: custom_rec.ads

```ada
package Custom_Rec is
  type Rec is record
    X : Integer;
  end record;
  function To_Integer (R : Rec) return Integer is
    (R.X);
end Custom_Rec;
```

Listing 91: show_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Rec; use Custom_Rec;

procedure Show_Conversion is
  R : Rec;
  I : Integer;
begin
  R := (X => 2);
  I := To_Integer (R);
  Put_Line ("I : " & I'Image);
end Show_Conversion;
```

Code block metadata
In this example, we have the To_Integer function that converts from the Rec type to the Integer type.

In other languages

In C++, you can define conversion operators to cast between objects of different classes. Also, you can overload the = operator. Consider this example:

```cpp
#include <iostream>

class T1 {
public:
  T1 (float x) :
    x(x) {} // If class T3 is declared before class T1, we can overload the "=" operator.
  // void operator=(T3 v) {
  //   x = static_cast<float>(v);
  // }
  void display();
private:
  float x;
};

class T3 {
public:
  T3 (float x, float y, float z) :
    x(x), y(y), z(z) {}
  // implicit conversion
  operator float() const { return (x + y + z) / 3.0; }
  // implicit conversion
  // operator T1() const { return T1((x + y + z) / 3.0); }
  // explicit conversion (C++11)
  explicit operator T1() const {
    return T1(float(*this));
  }
  void display();
private:
  float x, y, z;
};
```

(continues on next page)
Here, we're using operator float() and operator T1() to cast from an object of class T3 to a floating-point value and an object of class T1, respectively. (If we switch the order and declare the T3 class before the T1 class, we could overload the = operator, as you can see in the commented-out lines.)

In Ada, this kind of conversions isn't available. Instead, we have to implement conversion functions such as the To_Integer function from the previous code example. This is the corresponding implementation:

Listing 92: custom_defs.ads

```ada
package Custom_Defs is

  type T1 is private;

  function Init (X : Float)

end Custom_Defs;
```
Learning Ada

Listing 93: custom_def.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Custom_Defs is

  procedure Display (Obj : T1) is
  begin
    Put_Line ("(X => " & Obj.X'Image & ")");
  end Display;

  function To_Float (Obj : T3)
  return Float is
    ((Obj.X + Obj.Y + Obj.Z) / 3.0);

  function To_T1 (Obj : T3)
  return T1 is
    (Init (To_Float (Obj)));

  procedure Display (Obj : T3) is
  begin
```

(continues on next page)

25.1. Types
In this example, we translate the casting operators from the C++ version by implementing the To_Float and To_T1 functions. (In addition to that, we replace the C++ constructors by Init functions.)

### 25.1.7 Qualified Expressions

We already saw qualified expressions in the *Introduction to Ada* (page 91) course. As mentioned there, a qualified expression specifies the exact type or subtype that the target expression will be resolved to, and it can be either any expression in parentheses, or an aggregate:

```ada
package Simple_Integers is

    type Int is new Integer;

end Simple_Integers;
```

(continues on next page)
Learning Ada

```ada
subtype Int_Not_Zero is Int
  with Dynamic_Predicate => Int_Not_Zero /= 0;
end Simple_Integers;
```

**Listing 96: showQualified_expressions.adb**

```ada
with Simple_Integers; use Simple_Integers;
procedure Show_Qualified_Expressions is
  I : Int;
begin
  -- Using qualified expression Int'(N)
  I := Int'(0);
end Show_Qualified_Expressions;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example

MD5: 0a83e10b51c72827e322984bd5c8009d

Here, the qualified expression `Int'(0)` indicates that the value zero is of `Int` type.

**In the Ada Reference Manual**

- 4.7 Qualified Expressions

---

**Verifying subtypes**

**Note:** This feature was introduced in Ada 2022.

We can use qualified expressions to verify a subtype's predicate:

```ada
with Simple_Integers; use Simple_Integers;
procedure Show_Qualified_Expressions is
  I : Int;
begin
  I := Int_Not_Zero'(0);
end Show_Qualified_Expressions;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example

MD5: 3ca4ab8ad7bf75ae029047f673a15d70

**Build output**

showQualified_expressions.adb:6:23: warning: expression fails predicate check on...
showQualified_expressions.adb:6:23: warning: check will fail at run time [-gnatw.]

---


---

25.1. Types 333
Learning Ada

Runtime output

```
raised ADAASSERTIONSASSERTION_ERROR DynamicPredicate failed at show_qualified_expressions.adb:6
```

Here, the qualified expression {Int_Not_Zero'}(0) checks the dynamic predicate of the subtype. (This predicate check fails at runtime.)

### 25.1.8 Default initial values

In the *Introduction to Ada course* (page 65), we’ve seen that record components can have default values. For example:

Listing 98: defaults.ads

```ada
package Defaults is

    type R is record
        X : Positive := 1;
        Y : Positive := 10;
    end record;

end Defaults;
```

#### Code block metadata

**Project:** Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_1
**MD5:** e230be602cbb24a854e71c8176c7148c

In this section, we’ll extend the concept of default values to other kinds of type declarations, such as scalar types and arrays.

To assign a default value for a scalar type declaration — such as an enumeration and a new integer —, we use the Default_Value aspect:

Listing 99: defaults.ads

```ada
package Defaults is

    type E is (E1, E2, E3)
        with Default_Value => E1;

    type T is new Integer
        with Default_Value => -1;

end Defaults;
```

#### Code block metadata

**Project:** Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_2
**MD5:** e6cd8261b099278ceeb5fa91d318f6e

Note that we cannot specify a default value for a subtype:

Listing 100: defaults.ads

```ada
package Defaults is

    subtype T is Integer
        with Default_Value => -1;
```

(continues on next page)
For array types, we use the `Default_Component_Value` aspect:

```
package Defaults is

  type Arr is
    array (Positive range <>) of Integer
    with Default_Component_Value => -1;

end Defaults;
```

This is a package containing the declarations we've just seen:

```
package Defaults is

  type E is (E1, E2, E3)
    with Default_Value => E1;

  type T is new Integer
    with Default_Value => -1;

  -- We cannot specify default
  -- values for subtypes:
  --
  -- subtype T is Integer
  -- with Default_Value => -1;

  type R is record
    X : Positive := 1;
    Y : Positive := 10;
  end record;

  type Arr is
    array (Positive range <>) of Integer
    with Default_Component_Value => -1;

end Defaults;
```

25.1. Types
In the example below, we declare variables of the types from the Defaults package:

```
with Ada.Text_IO; use Ada.Text_IO;
with Defaults; use Defaults;

procedure Use_Defaults is
  E1 : E;
  T1 : T;
  R1 : R;
  A1 : Arr (1 .. 5);
begin
  Put_Line ("Enumeration: ",& E’Image (E1));
  Put_Line ("Integer type: ",& T’Image (T1));
  Put_Line ("Record type: ",& Positive’Image (R1.X)
             & ", ",
             & Positive’Image (R1.Y));
  Put ("Array type: ");
  for V of A1 loop
    Put (Integer’Image (V) & ");
  end loop;
  New_Line;
end Use_Defaults;
```

As we see in the Use_Defaults procedure, all variables still have their default values, since we haven't assigned any value to them.

In the Ada Reference Manual

- 3.5 Scalar Types
- 3.6 Array Types

31 http://www.ada-auth.org/standards/22rm/html/RM-3-5.html
25.1.9 Deferred Constants

Deferred constants are declarations where the value of the constant is not specified immediately, but rather deferred to a later point. In that sense, if a constant declaration is deferred, it is actually declared twice:

1. in the deferred constant declaration, and
2. in the full constant declaration.

The simplest form of deferred constant is the one that has a full constant declaration in the private part of the package specification. For example:

Listing 104: deferred_constants.ads

```ada
package Deferred_Constants is
  type Speed is new Long_Float;
  Light : constant Speed;    
  -- ^ deferred constant declaration
private
  Light : constant Speed := 299_792_458.0;   
  -- ^ full constant declaration
end Deferred_Constants;
```

Code block metadata
MD5: f76e42326889f70fa7e1e216576f9771

Another form of deferred constant is the one that imports a constant from an external implementation — using the Import keyword. We can use this to import a constant declaration from an implementation in C. For example, we can declare the `light` constant in a C file:

Listing 105: constants.c

```c
double light = 299792458.0;
```

Code block metadata
MD5: 71194a329dc5adaac3e01aff143a9943

Then, we can import this constant in the `Deferred_Constants` package:

Listing 106: deferred_constants.ads

```ada
package Deferred_Constants is
  type Speed is new Long_Float;
  Light : constant Speed with
    Import, Convention => C;   
    -- ^^^^^ deferred constant
    -- declaration; imported
    -- from C file
```

(continues on next page)
In this case, we don't have a full declaration in the `Deferred_Constants` package, as the `Light` constant is imported from the `constants.c` file.

As a rule, the deferred and the full declarations should match — except, of course, for the actual value that is missing in the deferred declaration. For instance, we're not allowed to use different types in both declarations. However, we may use a subtype in the full declaration — as long as it's compatible with the type that was used in the deferred declaration. For example:

Listing 107: deferred_constants.ads

```ada
package Deferred_Constants is

  type Speed is new Long_Float;

  subtype Positive_Speed is
    Speed range 0.0 .. Speed'Last;

  Light : constant Speed;
  -- ^ deferred constant declaration

private

  Light : constant Positive_Speed :=
    299_792_458.0;
  -- ^ full constant declaration
  -- using a subtype

end Deferred_Constants;
```

Here, we're using the `Speed` type in the deferred declaration of the `Light` constant, but we're using the `Positive_Speed` subtype in the full declaration.

A useful application of deferred constants is when the value of the constant is calculated using entities not meant to be compile-time visible to clients. As such, these other entities are only visible in the private part of the package, so that's where the value of the deferred constant must be computed. For example, the full constant declaration may be computed by a call to an expression function:

Listing 108: deferred_constants.ads

```ada
package Deferred_Constants is

  type Speed is new Long_Float;

  Light : constant Speed;

end Deferred_Constants;
```

(continues on next page)
-- ^ deferred constant declaration

private

function Calculate_Light return Speed is
  (299_792_458.0);

Light : constant Speed := Calculate_Light;
  -- ^ full constant declaration
  -- calling a private function
end Deferred_Constants;

In the Ada Reference Manual

- 7.4 Deferred Constants

25.1.10 User-defined literals

Note: This feature was introduced in Ada 2022.

Any type definition has a kind of literal associated with it. For example, integer types are
associated with integer literals. Therefore, we can initialize an object of integer type with
an integer literal:

Listing 109: simple_integer_literal.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Simple_Integer_Literal is
  V : Integer;
begin
  V := 10;
  Put_Line (Integer'Image (V));
end Simple_Integer_Literal;

25.1. Types
Here, 10 is the integer literal that we use to initialize the integer variable V. Other examples of literals are real literals and string literals, as we'll see later.

When we declare an enumeration type, we limit the set of literals that we can use to initialize objects of that type:

```
Listing 110: simple Enumeration.adb
with Ada.Text_IO; use Ada.Text_IO;
procedure SimpleEnumeration is
  type Activation_State is (Unknown, Off, On);
  S : Activation_State;
begin
  S := On;
  Put_Line (Activation_State'Image (S));
end SimpleEnumeration;
```

For objects of `Activation_State` type, such as S, the only possible literals that we can use are Unknown, Off and On. In this sense, types have a constrained set of literals that can be used for objects of that type.

User-defined literals allow us to extend this set of literals. We could, for example, extend the type declaration of `Activation_State` and allow the use of integer literals for objects of that type. In this case, we need to use the `Integer_Literal` aspect and specify a function that implements the conversion from literals to the type we're declaring. For this conversion from integer literals to the `Activation_State` type, we could specify that 0 corresponds to Off, 1 corresponds to On and other values correspond to Unknown. We'll see the corresponding implementation later.

These are the three kinds of literals and their corresponding aspect:

<table>
<thead>
<tr>
<th>Literal</th>
<th>Example</th>
<th>Aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1</td>
<td>Integer_Literal</td>
</tr>
<tr>
<td>Real</td>
<td>1.0</td>
<td>Real_Literal</td>
</tr>
<tr>
<td>String</td>
<td>&quot;On&quot;</td>
<td>String_Literal</td>
</tr>
</tbody>
</table>

For our previous `Activation_States` type, we could declare a function `Integer_To_Activation_State` that converts integer literals to one of the enumeration literals that we've specified for the `Activation_States` type:

```
Listing 111: activation_States.ads
package Activation_States is
  type Activation_State is (Unknown, Off, On)
```

with Integer_Literal =>
  Integer_To_Activation_State;

function Integer_To_Activation_State
  (S : String) return Activation_State;
end Activation_States;

Based on this specification, we can now use an integer literal to initialize an object S of
Activation_State type:

S : Activation_State := 1;

Note that we have a string parameter in the declaration of the Integer_To_Activation_State function, even though the function itself is only used to convert integer literals (but not string literals) to the Activation_State type. It's our job to process that string parameter in the implementation of the Integer_To_Activation_State function and convert it to an integer value — using Integer'Value, for example:

Listing 112: activation_states.adb

package body Activation_States is

  function Integer_To_Activation_State
    (S : String) return Activation_State is
  begin
    case Integer'Value (S) is
      when 0 => return Off;
      when 1 => return On;
      when others => return Unknown;
    end case;
  end Integer_To_Activation_State;
end Activation_States;

Let's look at a complete example that makes use of all three kinds of literals:

Listing 113: activation_states.ads

package Activation_States is

type Activation_State is (Unknown, Off, On)
  with String_Literal =>
    To_Activation_State,
  Integer_Literal =>
    Integer_To_Activation_State,
  Real_Literal =>
Learning Ada

---

Real_To_Activation_State;

function To_Activation_State
(S : Wide_Wide_String)
return Activation_State;

function Integer_To_Activation_State
(S : String)
return Activation_State;

function Real_To_Activation_State
(S : String)
return Activation_State;
end Activation_States;

---

package body Activation_States is

function To_Activation_State
(S : Wide_Wide_String)
return Activation_State
is
begin
if S = "Off" then
return Off;
elsif S = "On" then
return On;
else
return Unknown;
end if;
end To_Activation_State;

function Integer_To_Activation_State
(S : String)
return Activation_State
is
begin
case Integer'Value (S) is
when 0 => return Off;
when 1 => return On;
when others => return Unknown;
end case;
end Integer_To_Activation_State;

function Real_To_Activation_State
(S : String)
return Activation_State
is
V : constant Float := Float'Value (S);
begin
if V < 0.0 then
return Unknown;
elsif V < 1.0 then
return Off;
else
return On;
end if;
end Real_To_Activation_State;

---
In this example, we're extending the declaration of the Activation_State type to include string and real literals. For string literals, we use the To_Activation_State function, which converts:

- the "Off" string to Off,
- the "On" string to On, and
- any other string to Unknown.

For real literals, we use the Real_To_Activation_State function, which converts:

- any negative number to Unknown,
- a value in the interval [0, 1) to Off, and
- a value equal or above 1.0 to On.

Note that the string parameter of To_Activation_State function — which converts string literals — is of Wide_Wide_String type, and not of String type, as it's the case for the other conversion functions.

In the Activation_Examples procedure, we show how we can initialize an object of Activation_State type with all kinds of literals (string, integer and real literals).

With the definition of the Activation_State type that we've seen in the complete example, we can initialize an object of this type with an enumeration literal or a string, as both forms are defined in the type specification:
Listing 116: using_string_literal.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Activation_States; use Activation_States;

procedure Using_String_Literal is
  S1 : constant Activation_State := On;
  S2 : constant Activation_State := "On";
begin
  Put_Line (Activation_State'Image (S1));
  Put_Line (Activation_State'Image (S2));
end Using_String_Literal;
```

**Code block metadata**

MD5: 6ca6aa79b88058801688fc2dfb186091

**Runtime output**

ON
ON

Note we need to be very careful when designing conversion functions. For example, the use of string literals may limit the kind of checks that we can do. Consider the following misspelling of the Off literal:

Listing 117: misspelling_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Activation_States; use Activation_States;

procedure Misspelling_Example is
  S : constant Activation_State := Offf;
  -- ^ Error: Off is misspelled.
begin
  Put_Line (Activation_State'Image (S));
end Misspelling_Example;
```

**Code block metadata**

MD5: ebc1036a58e460a9212106606461b014

**Build output**

misspelling_example.adb:6:10: error: "Offf" is undefined
misspelling_example.adb:6:10: error: possible misspelling of "Off"
gprbuild: *** compilation phase failed

As expected, the compiler detects this error. However, this error is accepted when using the corresponding string literal:

Listing 118: misspelling_example.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Activation_States; use Activation_States;

(continues on next page)```
procedure Misspelling_Example is
  S : constant Activation_State :=
    "Off";
-- ^ Error: Off is misspelled.
begin
  Put_Line (Activation_State'Image (S));
end Misspelling_Example;

Code block metadata
MD5: 99f74c67712a9b55c146b9d57405e47f

Runtime output
UNKNOWN

Here, our implementation of To_Activation_State simply returns Unknown. In some cases, this might be exactly the behavior that we want. However, let's assume that we'd prefer better error handling instead. In this case, we could change the implementation of To_Activation_State to check all literals that we want to allow, and indicate an error otherwise — by raising an exception, for example. Alternatively, we could specify this in the preconditions of the conversion function:

```ada
function To_Activation_State
  (S : Wide_Wide_String)
return Activation_State
  with Pre => S = "Off" or
               S = "On" or
               S = "Unknown";
```

In this case, the precondition explicitly indicates which string literals are allowed for the To_Activation_State type.

User-defined literals can also be used for more complex types, such as records. For example:

Listing 119: silly_records.ads

```ada
package Silly_Records is
  type Silly is record
    X : Integer;
    Y : Float;
  end record
  with String_Literal => To_Silly;

function To_Silly
  (S : Wide_Wide_String)
return Silly;
end Silly_Records;
```

Listing 120: silly_records.adb

```ada
package body Silly_Records is
  function To_Silly
    (S : Wide_Wide_String)
  return Silly
  is
    begin
```

(continues on next page)
if S = "Magic" then
    return (X => 42, Y => 42.0);
else
    return (X => 0, Y => 0.0);
end if;
end To_Silly;
end Silly_Records;

Listing 121: silly_magic.adb

with Ada.Text_IO; use Ada.Text_IO;
with Silly_Records; use Silly_Records;

procedure Silly_Magic is
R1 : Silly;
begin
    R1 := "Magic";
    Put_Line (R1.X'Image & ", " & R1.Y'Image);
end Silly_Magic;

In this example, when we initialize an object of Silly type with a string, its components are:

• set to 42 when using the "Magic" string; or
• simply set to zero when using any other string.

Obviously, this example isn't particularly useful. However, the goal is to show that this approach is useful for more complex types where a string literal (or a numeric literal) might simplify handling those types. Used-defined literals let you design types in ways that, otherwise, would only be possible when using a preprocessor or a domain-specific language.

In the Ada Reference Manual

• 4.2.1 User-Defined Literals\textsuperscript{34}

\textsuperscript{34} http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html
25.2 Types and Representation

25.2.1 Enumeration Representation Clauses

We have talked about the internal code of an enumeration in another section (page 294). We may change this internal code by using a representation clause, which has the following format:

```ada
for Primary_Color is (Red => 1, Green => 5, Blue => 1000);
```

The value of each code in a representation clause must be distinct. However, as you can see above, we don't need to use sequential values — the values must, however, increase for each enumeration.

We can rewrite the previous example using a representation clause:

```
package Days is
  type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  for Day use (Mon => 2#00000001#, Tue => 2#00000010#, Wed => 2#00000100#, Thu => 2#00001000#, Fri => 2#00010000#, Sat => 2#00100000#, Sun => 2#01000000#);
end Days;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Days; use Days;

procedure Show_Days is
begin
  for D in Day loop
    Put_Line (Day'Image (D) & " position = " & Integer'Image (Day'Pos (D)));
    Put_Line (Day'Image (D) & " internal code = " & Integer'Image (Day'Enum_Rep (D)));
  end loop;
end Show_Days;
```

Code block metadata

- Project: Courses.Advanced_Ada.Data_Types.Type_Representation Enumeration_Representation_Clauses Enumeration_Values
- MD5: a70c3f8a967c355a4bf8f2d669f9c541

Runtime output
Learning Ada

Now, the value of the internal code is the one that we've specified in the representation clause instead of being equivalent to the value of the enumeration position.

In the example above, we're using binary values for each enumeration — basically viewing the integer value as a bit-field and assigning one bit for each enumeration. As long as we maintain an increasing order, we can use totally arbitrary values as well. For example:

```ada
package Days is

  type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  for Day use (Mon => 5, Tue => 9, Wed => 42, Thu => 49, Fri => 50, Sat => 66, Sun => 99);

end Days;
```

### 25.2.2 Data Representation

This section provides a glimpse on attributes and aspects used for data representation. They are usually used for embedded applications because of strict requirements that are often found there. Therefore, unless you have very specific requirements for your application, in most cases, you won't need them. However, you should at least have a rudimentary understanding of them. To read a thorough overview on this topic, please refer to the *Introduction to Embedded Systems Programming* (page 1117) course.

In the Ada Reference Manual

- 13.2 Packed Types
- 13.3 Operational and Representation Attributes
- 13.5.3 Bit Ordering

---


348 Chapter 25. Data types
Sizes

Ada offers multiple attributes to retrieve the size of a type or an object:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Size of the representation of a subtype or an object (in bits).</td>
</tr>
<tr>
<td>Object_Size</td>
<td>Size of a component or an aliased object (in bits).</td>
</tr>
<tr>
<td>Component_Size</td>
<td>Size of a component of an array (in bits).</td>
</tr>
<tr>
<td>Storage_Size</td>
<td>Number of storage elements reserved for an access type or a task object.</td>
</tr>
</tbody>
</table>

For the first three attributes, the size is measured in bits. In the case of Storage_Size, the size is measured in storage elements. Note that the size information depends your target architecture. We’ll discuss some examples to better understand the differences among those attributes.

Important

A storage element is the smallest element we can use to store data in memory. As we'll see soon, a storage element corresponds to a byte in many architectures.

The size of a storage element is represented by the System.Storage_Unit constant. In other words, the storage unit corresponds to the number of bits used for a single storage element.

In typical architectures, System.Storage_Unit is 8 bits. In this specific case, a storage element is equal to a byte in memory. Note, however, that System.Storage_Unit might have a value different than eight in certain architectures.

Size attribute and aspect

Let's start with a code example using the Size attribute:

Listing 125: custom_types.ads

```ada
package Custom_Types is
   type UInt_7 is range 0 .. 127;
   type UInt_7_S32 is range 0 .. 127
      with Size => 32;
end Custom_Types;
```

Listing 126: show_sizes.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Types; use Custom_Types;
procedure Show_Sizes is
   V1 : UInt_7;
   V2 : UInt_7_S32;
begin
   Put_Line ("UInt_7'Size: ");
   Put_Line ("UInt_7_S32'Size: ");
end Show_Sizes;
```

Depending on your target architecture, you may see this output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>UInt_7</td>
<td>7</td>
</tr>
<tr>
<td>UInt_7 Object</td>
<td>8</td>
</tr>
<tr>
<td>V1</td>
<td>8</td>
</tr>
<tr>
<td>UInt_7_S32</td>
<td>32</td>
</tr>
<tr>
<td>UInt_7_S32 Object</td>
<td>32</td>
</tr>
<tr>
<td>V2</td>
<td>32</td>
</tr>
</tbody>
</table>

When we use the Size attribute for a type T, we're retrieving the minimum number of bits necessary to represent objects of that type. Note that this is not the same as the actual size of an object of type T because the compiler will select an object size that is appropriate for the target architecture.

In the example above, the size of the UInt_7 is 7 bits, while the most appropriate size to store objects of this type in the memory of our target architecture is 8 bits. To be more specific, the range of UInt_7 (0..127) can be perfectly represented in 7 bits. However, most target architectures don't offer 7-bit registers or 7-bit memory storage, so 8 bits is the most appropriate size in this case.

We can retrieve the size of an object of type T by using the Object_Size. Alternatively, we can use the Size attribute directly on objects of type T to retrieve their actual size — in our example, we write V1'Size to retrieve the size of V1.

In the example above, we've used both the Size attribute (for example, UInt_7'Size) and
the Size aspect (with Size => 32). While the size attribute is a function that returns the size, the size aspect is a request to the compiler to verify that the expected size can be used on the target platform. You can think of this attribute as a dialog between the developer and the compiler:

(Developer) "I think that UInt_7_S32 should be stored using at least 32 bits. Do you agree?"

(Ada compiler) "For the target platform that you selected, I can confirm that this is indeed the case."

Depending on the target platform, however, the conversation might play out like this:

(Developer) "I think that UInt_7_S32 should be stored using at least 32 bits. Do you agree?"

(Ada compiler) "For the target platform that you selected, I cannot possibly do it! COMPILATION ERROR!"

**Component size**

Let's continue our discussion on sizes with an example that makes use of the Component_Size attribute:

Listing 127: custom_types.ads

```ada
package Custom_Types is

  type UInt_7 is range 0 .. 127;

  type UInt_7_Array is
    array (Positive range <>) of UInt_7;

  type UInt_7_Array_Comp_32 is
    array (Positive range <>) of UInt_7
      with Component_Size => 32;

end Custom_Types;
```

Listing 128: show_sizes.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Custom_Types; use Custom_Types;

procedure Show_Sizes is
  Arr_1 : UInt_7_Array (1 .. 20);
  Arr_2 : UInt_7_Array_Comp_32 (1 .. 20);
begin
  Put_Line ("UInt_7_Array'Size: " & UInt_7_Array'Size'Image);
  Put_Line ("UInt_7_Array'Object_Size: " & UInt_7_Array'Object_Size'Image);
  Put_Line ("UInt_7_Array'Component_Size: " & UInt_7_Array'Component_Size'Image);
  Put_Line ("Arr_1'Component_Size: " & Arr_1'Component_Size'Image);
  Put_Line ("Arr_2'Component_Size: " & Arr_2'Component_Size'Image);
```

(continues on next page)
Put_Line
("Arr_1'Size: 
& Arr_1'Size'Image);
New_Line;
Put_Line
("UInt_7_Array_Comp_32'Object_Size: 
& UInt_7_Array_Comp_32'Size'Image);
Put_Line
("Arr_2'Component_Size: 
& Arr_2'Component_Size'Image);
Put_Line
("Arr_2'Size: 
& Arr_2'Size'Image);
New_Line;
end Show_Sizes;

Depending on your target architecture, you may see this output:

**Runtime output**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UInt_7_Array'Size</td>
<td>17179869176</td>
</tr>
<tr>
<td>UInt_7_Array'Object_Size</td>
<td>17179869176</td>
</tr>
<tr>
<td>UInt_7_Array'Component_Size</td>
<td>8</td>
</tr>
<tr>
<td>Arr_1'Component_Size</td>
<td>8</td>
</tr>
<tr>
<td>Arr_1'Size</td>
<td>160</td>
</tr>
<tr>
<td>UInt_7_Array_Comp_32'Object_Size</td>
<td>68719476704</td>
</tr>
<tr>
<td>UInt_7_Array_Comp_32'Object_Size</td>
<td>68719476704</td>
</tr>
<tr>
<td>UInt_7_Array_Comp_32'Component_Size</td>
<td>32</td>
</tr>
<tr>
<td>Arr_2'Component_Size</td>
<td>32</td>
</tr>
<tr>
<td>Arr_2'Size</td>
<td>640</td>
</tr>
</tbody>
</table>
Here, the value we get for Component_Size of the UInt_7_Array type is 8 bits, which matches the UInt_7'Object_Size — as we’ve seen in the previous subsection. In general, we expect the component size to match the object size of the underlying type.

However, we might have component sizes that aren’t equal to the object size of the component's type. For example, in the declaration of the UInt_7_Array_Comp_32 type, we're using the Component_Size aspect to query whether the size of each component can be 32 bits:

```
type UInt_7_Array_Comp_32 is
  array (Positive range <>) ofUInt_7
  with Component_Size => 32;
```

If the code compiles, we see this value when we use the Component_Size attribute. In this case, even though UInt_7'Object_Size is 8 bits, the component size of the array type (UInt_7_Array_Comp_32'Component_Size) is 32 bits.

Note that we can use the Component_Size attribute with data types, as well as with actual objects of that data type. Therefore, we can write UInt_7_Array'Component_Size and Arr_1'Component_Size, for example.

This big number (17179869176 bits) for UInt_7_Array'Size and UInt_7_Array'Object_Size might be surprising for you. This is due to the fact that Ada is reporting the size of the UInt_7_Array type for the case when the complete range is used. Considering that we specified a positive range in the declaration of the UInt_7_Array type, the maximum length on this machine is $2^{31} - 1$. The object size of an array type is calculated by multiplying the maximum length by the component size. Therefore, the object size of the UInt_7_Array type corresponds to the multiplication of $2^{31} - 1$ components (maximum length) by 8 bits (component size).

### Storage size

To complete our discussion on sizes, let’s look at this example of storage sizes:

#### Listing 129: custom_types.ads

```
package Custom_Types is
  type UInt_7 is range 0 .. 127;
  type UInt_7_Access is access UInt_7;
end Custom_Types;
```

#### Listing 130: show_sizes.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with System;
with Custom_Types; use Custom_Types;

procedure Show_Sizes is
  AV1, AV2 : UInt_7_Access;
begin
  Put_Line ("UInt_7_Access'Storage_Size: "
```

Learning Ada

(continued from previous page)

& Uint_7_Access'Storage_Size'Image);
Put_Line
("Uint_7_Access'Storage_Size (bits): "
& Integer'Image (Uint_7_Access'Storage_Size
* System.Storage_Unit));
Put_Line
("Uint_7'Size: "
& Uint_7'Size'Image);
Put_Line
("Uint_7_Access'Size: "
& Uint_7_Access'Size'Image);
Put_Line
("Uint_7_Access'Object_Size: "
& Uint_7_Access'Object_Size'Image);
Put_Line
("AV1'Size: "
& AV1'Size'Image);
New_Line;
Put_Line ("Allocating AV1...";
AV1 := new Uint_7;
Put_Line ("Allocating AV2...";
AV2 := new Uint_7;
New_Line;
Put_Line ("AV1.all'Size: "
& AV1.all'Size'Image);
New_Line;
end Show_Sizes;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation._Sizes
MD5: 5e652ee25b8550ac331f3ce98e24f7ba

Runtime output

Uint_7_Access'Storage_Size: 0
Uint_7_Access'Storage_Size (bits): 0
Uint_7'Size: 7
Uint_7_Access'Size: 64
Uint_7_Access'Object_Size: 64
AV1'Size: 64
Allocating AV1...
Allocating AV2...
AV1.all'Size: 8

Depending on your target architecture, you may see this output:

Uint_7_Access'Storage_Size: 0
Uint_7_Access'Storage_Size (bits): 0
Uint_7'Size: 7
Uint_7_Access'Size: 64
Uint_7_Access'Object_Size: 64

(continues on next page)
As we've mentioned earlier on, Storage_Size corresponds to the number of storage elements reserved for an access type or a task object. In this case, we see that the storage size of the UInt_7_Access type is zero. This is because we haven't indicated that memory should be reserved for this data type. Thus, the compiler doesn't reserve memory and simply sets the size to zero.

Because Storage_Size gives us the number of storage elements, we have to multiply this value by System.Storage_Unit to get the total storage size in bits. (In this particular example, however, the multiplication doesn't make any difference, as the number of storage elements is zero.)

Note that the size of our original data type UInt_7 is 7 bits, while the size of its corresponding access type UInt_7_Access (and the access object AV1) is 64 bits. This is due to the fact that the access type doesn't contain an object, but rather memory information about an object. You can retrieve the size of an object allocated via new by first dereferencing it — in our example, we do this by writing AV1.all'Size.

Now, let's use the Storage_Size aspect to actually reserve memory for this data type:

```ada
package Custom_Types is
  type UInt_7 is range 0 .. 127;
  type UInt_7_Reserved_Access is access UInt_7
    with Storage_Size => 8;
end Custom_Types;
```

```ada
procedure Show_Sizes is
  RAV1, RAV2 : UInt_7_Reserved_Access;
begin
  Put_Line
    ("UInt_7_Reserved_Access'Storage_Size: "
     & UInt_7_Reserved_Access'Storage_Size'Image);
  Put_Line
    ("UInt_7_Reserved_Access'Storage_Size (bits): "
     & Integer'Image
     (UInt_7_Reserved_Access'Storage_Size
      * System.Storage_Unit));
  Put_Line
    ("UInt_7_Reserved_Access'Size: "
     & UInt_7_Reserved_Access'Size'Image);
end Show_Sizes;
```

(continues on next page)
In this case, we're reserving 8 storage elements in the declaration of 

```
UInt_7 Reserved Access 'Storage Size: 8
```

Since each storage element corresponds to one byte (8 bits) in this architecture, we're reserving a maximum of 64 bits (or 8 bytes) for the UInt_7 Reserved Access type.

This example raises an exception at runtime — a storage error, to be more specific. This is because the maximum reserved size is 64 bits, and the size of a single access object is 64 bits as well. Therefore, after the first allocation, the reserved storage space is already consumed, so we cannot allocate a second access object.

This behavior might be quite limiting in many cases. However, for certain applications
where memory is very constrained, this might be exactly what we want to see. For example, having an exception being raised when the allocated memory for this data type has reached its limit might allow the application to have enough memory to at least handle the exception gracefully.

**Alignment**

For many algorithms, it's important to ensure that we're using the appropriate alignment. This can be done by using the Alignment attribute and the Alignment aspect. Let's look at this example:

```
package Custom_Types is

  type UInt_7 is range 0 .. 127;

  type Aligned_UInt_7 is new UInt_7
    with Alignment => 4;

end Custom_Types;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Custom_Types; use Custom_Types;

procedure Show_Alignment is
  V   : constant UInt_7 := 0;
  Aligned_V : constant Aligned_UInt_7 := 0;
begin
  Put_Line ("UInt_7'Alignment: "
            & UInt_7'Alignment'Image);
  Put_Line ("UInt_7'Size: 
            & UInt_7'Size'Image);
  Put_Line ("UInt_7'Object_Size: "
            & UInt_7'Object_Size'Image);
  Put_Line ("V'Alignment: 
            & V'Alignment'Image);
  Put_Line ("V'Size: 
            & V'Size'Image);
  New_Line;
  Put_Line ("Aligned_UInt_7'Alignment: "
            & Aligned_UInt_7'Alignment'Image);
  Put_Line ("Aligned_UInt_7'Size: 
            & Aligned_UInt_7'Size'Image);
  Put_Line ("Aligned_UInt_7'Object_Size: "
            & Aligned_UInt_7'Object_Size'Image);
  Put_Line ("Aligned_V'Alignment: 
```

(continues on next page)
Depending on your target architecture, you may see this output:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Alignment</th>
<th>Size</th>
<th>Object Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>UInt_7</td>
<td>1</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>V</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Aligned_UInt_7</td>
<td>4</td>
<td>7</td>
<td>32</td>
</tr>
<tr>
<td>Aligned_V</td>
<td>4</td>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>

In this example, we're reusing the UInt_7 type that we've already been using in previous examples. Because we haven't specified any alignment for the UInt_7 type, it has an alignment of 1 storage unit (or 8 bits). However, in the declaration of the Aligned_UInt_7 type, we're using the Alignment aspect to request an alignment of 4 storage units (or 32 bits):

```ada
type Aligned_UInt_7 is new UInt_7
  with Alignment => 4;
```

When using the Alignment attribute for the Aligned_UInt_7 type, we can confirm that its alignment is indeed 4 storage units (bytes).

Note that we can use the Alignment attribute for both data types and objects — in the code above, we're using UInt_7'Alignment and V'Alignment, for example.

Because of the alignment we're specifying for the Aligned_UInt_7 type, its size — indicated by the Object_Size attribute — is 32 bits instead of 8 bits as for the UInt_7 type.

Note that you can also retrieve the alignment associated with a class using S'Class'Alignment. For example:
Listing 135: show_class_alignment.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Class_Alignment is

  type Point_1D is tagged record
    X : Integer;
  end record;

  type Point_2D is new Point_1D with record
    Y : Integer;
  end record
    with Alignment => 16;

  type Point_3D is new Point_2D with record
    Z : Integer;
  end record;

begin
  Put_Line ("1D_Point'Alignment: ", 
            & Point_1D'Alignment'Image);
  Put_Line ("1D_Point'Class'Alignment: ", 
            & Point_1D'Class'Alignment'Image);
  Put_Line ("2D_Point'Alignment: ", 
            & Point_2D'Alignment'Image);
  Put_Line ("2D_Point'Class'Alignment: ", 
            & Point_2D'Class'Alignment'Image);
  Put_Line ("3D_Point'Alignment: ", 
            & Point_3D'Alignment'Image);
  Put_Line ("3D_Point'Class'Alignment: ", 
            & Point_3D'Class'Alignment'Image);

end Show_Class_Alignment;
```

**Code block metadata**

...Class_Alignment
MD5: 4eb28d59439d1eb86cd23fb08acd3493

**Runtime output**

1D_Point'Alignment: 8
1D_Point'Class'Alignment: 8
2D_Point'Alignment: 16
2D_Point'Class'Alignment: 16
3D_Point'Alignment: 16
3D_Point'Class'Alignment: 16

**Overlapping Storage**

Algorithms can be designed to perform in-place or out-of-place processing. In other words, they can take advantage of the fact that input and output arrays share the same storage space or not.

We can use the Has_Same_Storage and the Overlaps_Storage attributes to retrieve more information about how the storage space of two objects related to each other:

- the Has_Same_Storage attribute indicates whether two objects have the exact same storage.

25.2. Types and Representation 359
- A typical example is when both objects are exactly the same, so they obviously share the same storage. For example, for array A, A’Has_Same_Storage (A) is always True.

- the Overlaps_Storage attribute indicates whether two objects have at least one bit in common.

- Note that, if two objects have the same storage, this implies that their storage also overlaps. In other words, A’Has_Same_Storage (B) = True implies that A’Overlaps_Storage (B) = True.

Let's look at this example:

Listing 136: int_array_processing.ads

```ada
package Int_Array_Processing is

  type Int_Array is
    array (Positive range <>) of Integer;

  procedure Show_Storage (X : Int_Array; Y : Int_Array);

  procedure Process (X : Int_Array; Y : out Int_Array);

end Int_Array_Processing;
```

Listing 137: int_array_processing.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Int_Array_Processing is

  procedure Show_Storage (X : Int_Array; Y : Int_Array) is
    begin
      if X’Has_Same_Storage (Y) then
        Put_Line ("Info: X and Y have the same storage.");
      else
        Put_Line ("Info: X and Y don’t have “the same storage.");
      end if;
      if X’Overlaps_Storage (Y) then
        Put_Line ("Info: X and Y overlap.");
      else
        Put_Line ("Info: X and Y don’t overlap.");
      end if;
    end Show_Storage;

  procedure Process (X : Int_Array; Y : out Int_Array) is
    begin
      Put_Line ("==== PROCESS ====");
      Show_Storage (X, Y);
      if X’Has_Same_Storage (Y) then
        Put_Line ("In-place processing...");
      else
```

(continues on next page)
if not X’Overlaps_Storage (Y) then
   Put_Line
   ("Out-of-place processing...");
else
   Put_Line
   ("Cannot process 
& "overlapping arrays...");
end if;
end if;
New_Line;
end Process;
end Int_Array_Processing;

with Int_Array_Processing;
use Int_Array_Processing;

procedure Main is
   A : Int_Array (1 .. 20) := (others => 3);
   B : Int_Array (1 .. 20) := (others => 4);
begin
   Process (A, A);
   -- In-place processing:
   -- sharing the exact same storage
   Process (A (1 .. 10), A (10 .. 20));
   -- Overlapping one component: A (10)
   Process (A (1 .. 10), A (11 .. 20));
   -- Out-of-place processing:
   -- same array, but not sharing any storage
   Process (A, B);
   -- Out-of-place processing:
   -- two different arrays
end Main;

Code block metadata
   Overlapping_Storage
MD5: 0f599163c6f24c3ef46ec6577b501c21

Build output
int_array_processing.adb:29:24: warning: "Y" may be referenced before it has a
   value [enabled by default]

Runtime output
==== PROCESS ====
Info: X and Y have the same storage.
Info: X and Y overlap.
In-place processing...

==== PROCESS ====
Info: X and Y don’t have the same storage.
Info: X and Y overlap.
Cannot process overlapping arrays...
In this code example, we implement two procedures:

- **Show_Storage**, which shows storage information about two arrays by using the Has_Same_Storage and Overlaps_Storage attributes.

- **Process**, which are supposed to process an input array X and store the processed data in the output array Y.
  - Note that the implementation of this procedure is actually just a mock-up, so that no processing is actually taking place.

We have four different instances of how we can call the Process procedure:

- in the Process (A, A) call, we're using the same array for the input and output arrays. This is a perfect example of in-place processing. Because the input and the output arrays arguments are actually the same object, they obviously share the exact same storage.

- in the Process (A (1 .. 10), A (10 .. 20)) call, we're using two slices of the A array as input and output arguments. In this case, a single component of the A array is shared: A (10). Because the storage space is overlapping, but not exactly the same, neither in-place nor out-of-place processing can usually be used in this case.

- in the Process (A (1 .. 10), A (11 .. 20)) call, even though we're using the same array A for the input and output arguments, we're using slices that are completely independent from each other, so that the input and output arrays are not sharing any storage in this case. Therefore, we can use out-of-place processing.

- in the Process (A, B) call, we have two different arrays — which obviously don't share any storage space —, so we can use out-of-place processing.

### Packed Representation

As we've seen previously, the minimum number of bits required to represent a data type might be less than the actual number of bits used to store an object of that same type. We've seen an example where UInt_7'Size was 7 bits, while UInt_7'Object_Size was 8 bits. The most extreme case is the one for the **Boolean** type: in this case, **Boolean'Size** is 1 bit, while **Boolean'Object_Size** might be 8 bits (or even more on certain architectures). In such cases, we have 7 (or more) unused bits in memory for each object of **Boolean** type. In other words, we're wasting memory. On the other hand, we're gaining speed of access because we can directly access each element without having to first change its internal representation back and forth. We'll come back to this point later.

The situation is even worse when implementing bit-fields, which can be declared as an array of **Boolean** components. For example:
Listing 139: flag_definitions.ads

```ada
package Flag_Definitions is

  type Flags is
    array (Positive range <>) of Boolean;

end Flag_Definitions;
```

Listing 140: show_flags.adb

```ada
with Ada.Text_IO;    use Ada.Text_IO;
with Flag_Definitions; use Flag_Definitions;

procedure Show_Flags is
  Flags_1 : Flags (1 .. 8);
begin
  Put_Line ("Boolean'Size: ",
            & Boolean'Size'Image);
  Put_Line ("Boolean'Object_Size: ",
            & Boolean'Object_Size'Image);
  Put_Line ("Flags_1'Size: ",
            & Flags_1'Size'Image);
  Put_Line ("Flags_1'Component_Size: ",
            & Flags_1'Component_Size'Image);
end Show_Flags;
```

In this example, we're declaring the Flags type as an array of `Boolean` components. As we can see in this case, although the size of the `Boolean` type is just 1 bit, an object of this type has a size of 8 bits. Consequently, each component of the Flags type has a size of 8 bits. Moreover, an array with 8 components of `Boolean` type — such as the Flags_1 array — has a size of 64 bits.

Therefore, having a way to compact the representation — so that we can store multiple objects without wasting storage space — may help us improving memory usage. This is actually possible by using the Pack aspect. For example, we could extend the previous example and declare a Packed_Flags type that makes use of this aspect:
### Listing 141: flag_definitions.ads

```ada
package Flag_Definitions is
  type Flags is array (Positive range <>) of Boolean;
  type Packed_Flags is array (Positive range <>) of Boolean
    with Pack;
end Flag_Definitions;
```

### Listing 142: show_packed_flags.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Flag_Definitions; use Flag_Definitions;

procedure Show_Packed_Flags is
  Flags_1 : Flags (1 .. 8);
  Flags_2 : Packed_Flags (1 .. 8);
begin
  Put_Line ("Boolean'Size: " & Boolean'Size'Image);
  Put_Line ("Boolean'Object_Size: " & Boolean'Object_Size'Image);
  Put_Line ("Flags 1'Size: " & Flags_1'Size'Image);
  Put_Line ("Flags 1'Component_Size: " & Flags_1'Component_Size'Image);
  Put_Line ("Flags 2'Size: " & Flags_2'Size'Image);
  Put_Line ("Flags 2'Component_Size: " & Flags_2'Component_Size'Image);
end Show_Packed_Flags;
```

---

**Code block metadata**

- **Project:** Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.Packed_Flags
- **MD5:** c71cf68dc8bc41d0df2a5e3eb61b51fd

**Build output**

- show_packed_flags.adb:5:04: warning: variable "Flags_1" is read but never assigned. [-gnatwv]
- show_packed_flags.adb:6:04: warning: variable "Flags_2" is read but never assigned. [-gnatwv]

**Runtime output**

- Boolean'Size: 1
- Boolean'Object_Size: 8
- Flags 1'Size: 64
- Flags 1'Component_Size: 8
- Flags 2'Size: 8
- Flags 2'Component_Size: 1

Depending on your target architecture, you may see this output:
In this example, we're declaring the `Flags_2` array of `Packed_Flags` type. Its size is 8 bits — instead of the 64 bits required for the `Flags_1` array. Because the array type `Packed_Flags` is packed, we can now effectively use this type to store an object of `Boolean` type using just 1 bit of the memory, as indicated by the `Flags_2'Component_Size` attribute.

In many cases, we need to convert between a normal representation (such as the one used for the `Flags_1` array above) to a packed representation (such as the one for the `Flags_2` array). In many programming languages, this conversion may require writing custom code with manual bit-shifting and bit-masking to get the proper target representation. In Ada, however, we just need to indicate the actual type conversion, and the compiler takes care of generating code containing bit-shifting and bit-masking to perform the type conversion.

Let's modify the previous example and introduce this type conversion:

```
package Flag_Definitions is

  type Flags is
    array (Positive range <>) of Boolean;

  type Packed_Flags is
    array (Positive range <>) of Boolean
      with Pack;

  Default_Flags : constant Flags :=
    (True, True, False, True,
     False, False, True, True);

end Flag_Definitions;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Flag_Definitions; use Flag_Definitions;

procedure Show_Flag_Conversion is
  Flags_1 : Flags (1 .. 8);
  Flags_2 : Packed_Flags (1 .. 8);

begin
  Flags_1 := Default_Flags;
  Flags_2 := Packed_Flags (Flags_1);

  for I in Flags_2'Range loop
    Put_Line (I'Image & " : " &
      Flags_1 (I)'Image & ", " &
      Flags_2 (I)'Image);
  end loop;

end Show_Flag_Conversion;
```
Learning Ada

Runtime output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>TRUE, TRUE</td>
</tr>
<tr>
<td>2:</td>
<td>TRUE, TRUE</td>
</tr>
<tr>
<td>3:</td>
<td>FALSE, FALSE</td>
</tr>
<tr>
<td>4:</td>
<td>TRUE, TRUE</td>
</tr>
<tr>
<td>5:</td>
<td>FALSE, FALSE</td>
</tr>
<tr>
<td>6:</td>
<td>FALSE, FALSE</td>
</tr>
<tr>
<td>7:</td>
<td>TRUE, TRUE</td>
</tr>
<tr>
<td>8:</td>
<td>TRUE, TRUE</td>
</tr>
</tbody>
</table>

In this extended example, we're now declaring `Default_Flags` as an array of constant flags, which we use to initialize `Flags_1`.

The actual conversion happens with `Flags_2 := Packed_Flags (Flags_1)`. Here, the type conversion `Packed_Flags()` indicates that we're converting from the normal representation (used for the `Flags` type) to the packed representation (used for `Packed_Flags` type). We don't need to write more code than that to perform the correct type conversion.

Also, by using the same strategy, we could read information from a packed representation. For example:

```ada
Flags_1 := Flags (Flags_2);
```

In this case, we use `Flags()` to convert from a packed representation to the normal representation.

We elaborate on the topic of converting between data representations in the section on changing data representation (page 376).

**Trade-offs**

As indicated previously, when we're using a packed representation (vs. using a standard unpacked representation), we're trading off speed of access for less memory consumption. The following table summarizes this:

<table>
<thead>
<tr>
<th>Representation</th>
<th>More speed of access</th>
<th>Less memory consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpacked</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Packed</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

On one hand, we have better memory usage when we apply packed representations because we may save many bits for each object. On the other hand, there's a cost associated with accessing those packed objects because they need to be unpacked before we can actually access them. In fact, the compiler generates code — using bit-shifting and bit-masking — that converts a packed representation into an unpacked representation, which we can then access. Also, when storing a packed object, the compiler generates code that converts the unpacked representation of the object into the packed representation.

This packing and unpacking mechanism has a performance cost associated with it, which results in less speed of access for packed objects. As usual in those circumstances, before using packed representation, we should assess whether memory constraints are more important than speed in our target architecture.
25.2.3 Record Representation and storage clauses

In this section, we discuss how to use record representation clauses to specify how a record is represented in memory. Our goal is to provide a brief introduction into the topic. If you're interested in more details, you can find a thorough discussion about record representation clauses in the Introduction to Embedded Systems Programming (page 1117) course.

Let's start with the simple approach of declaring a record type without providing further information. In this case, we're basically asking the compiler to select a reasonable representation for that record in the memory of our target architecture.

Let's see a simple example:

Listing 145: p.ads

```ada
package P is
  type R is record
    A : Integer;
    B : Integer;
  end record;
end P;
```

Considering a typical 64-bit PC architecture with 8-bit storage units, and `Integer` defined as a 32-bit type, we get this memory representation:

Each storage unit is a position in memory. In the graph above, the numbers on the top (0, 1, 2, ...) represent those positions for record R.

In addition, we can show the bits that are used for components A and B:

The memory representation we see in the graph above can be described in Ada using representation clauses, as you can see in the code starting at the `for R use` record line in the code example below — we'll discuss the syntax and further details right after this example.
package P is

  type R is record
    A : Integer;
    B : Integer;
  end record;

  -- Representation clause for record R:
  for R use record
    A at 0 range 0 .. 31;
    -- ^ starting memory position
    B at 4 range 0 .. 31;
    -- ^ first bit .. last bit
  end record;

end P;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Rep_Clauses_2
MD5: b6be86ae7e1a5c2e7d981fe37bad49ed

Here, we're specifying that the A component is stored in the bits #0 up to #31 starting at position #0. Note that the position itself doesn't represent an absolute address in the device's memory; instead, it's relative to the memory space reserved for that record. The B component has the same 32-bit range, but starts at position #4.

This is a generalized view of the syntax:

```ada
for Record_Type use record
  Component_Name at Start_Position
    range First_Bit .. Last_Bit;
end record;
```

These are the elements we see above:

- Component_Name: name of the component (from the record type declaration);
- Start_Position: start position — in storage units — of the memory space reserved for that component;
- First_Bit: first bit (in the start position) of the component;
- Last_Bit: last bit of the component.

Note that the last bit of a component might be in a different storage unit. Since the Integer type has a larger width (32 bits) than the storage unit (8 bits), components of that type span over multiple storage units. Therefore, in our example, the first bit of component A is at position #0, while the last bit is at position #3.

Also note that the last eight bits of component A are bits #24 .. #31. If we think in terms of storage units, this corresponds to bits #0 .. #7 of position #3. However, when specifying the last bit in Ada, we always use the First_Bit value as a reference, not the position where those bits might end up. Therefore, we write range 0 .. 31, well knowing that those 32 bits span over four storage units (positions #0 .. #3).

In the Ada Reference Manual

- 13.5.1 Record Representation Clauses

---

Storage Place Attributes

We can retrieve information about the start position, and the first and last bits of a component by using the storage place attributes:

- **Position**, which retrieves the start position of a component;
- **First_Bit**, which retrieves the first bit of a component;
- **Last_Bit**, which retrieves the last bit of a component.

Note, however, that these attributes can only be used with actual records, and not with record types.

We can revisit the previous example and verify how the compiler represents the R type in memory:

Listing 147: p.ads

```ada
package P is
  type R is record
    A : Integer;
    B : Integer;
  end record;
end P;
```

Listing 148: show_storage.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System;
with P; use P;

procedure Show_Storage is
  R1 : R;
begin
  Put_Line ("R'Size: ", & R'Size'Image);
  Put_Line ("R'Object_Size: ", & R'Object_Size'Image);
  New_Line;
  Put_Line ("System.Storage_Unit: ", & System.Storage_Unit'Image);
  New_Line;
  Put_Line ("R1.A'Position : ", & R1.A'Position'Image);
  Put_Line ("R1.A'First_Bit : ", & R1.A'First_Bit'Image);
  Put_Line ("R1.A'Last_Bit : ", & R1.A'Last_Bit'Image);
  New_Line;
  Put_Line ("R1.B'First_Bit : ", & R1.B'First_Bit'Image);
end Show_Storage;
```

(continues on next page)
First of all, we see that the size of the $R$ type is 64 bits, which can be explained by those two 32-bit integer components. Then, we see that components $A$ and $B$ start at positions #0 and #4, and each one makes use of bits in the range from #0 to #31. This matches the graph we’ve seen above.

### Using Representation Clauses

We can use representation clauses to change the way the compiler handles memory for a record type. For example, let’s say we want to have an empty storage unit between components $A$ and $B$. We can use a representation clause where we specify that component $B$ starts at position #5 instead of #4, leaving an empty byte after component $A$ and before component $B$:

<table>
<thead>
<tr>
<th>position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>#0 .. 7</td>
<td>#8 .. #15</td>
<td>#16 .. #23</td>
<td>#24 .. #31</td>
<td>#0 .. 7</td>
<td>#8 .. #15</td>
<td>#16 .. #23</td>
<td>#24 .. #31</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>$A$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$B$</td>
</tr>
</tbody>
</table>

This is the code that implements that:

Learning Ada

Listing 149: p.ads

```ada
package P is

  type R is record
    A : Integer;
    B : Integer;
  end record;

  for R use record
    A at 0 range 0 .. 31;
    B at 5 range 0 .. 31;
  end record;

end P;
```

Listing 150: show_empty_byte.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with P; use P;

procedure Show_Empty_Byte is begin
  Put_Line ("R'Size: " & R'Size'Image);
  Put_Line ("R'Object_Size: " & R'Object_Size'Image);
end Show_Empty_Byte;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Rep_Clauses_Empty_Byte
MD5: c616e534e95a06f2e8b3052a3e8a9aab

Runtime output

R'Size: 72
R'Object_Size: 96

When running the application above, we see that, due to the extra byte in the record representation, the sizes increase. On a typical 64-bit PC, R'Size is now 76 bits, which reflects the additional eight bits that we introduced between components A and B. Depending on the target architecture, you may also see that R'Object_Size is now 96 bits, which is the size the compiler selects as the most appropriate for this record type. As we've mentioned in the previous section, we can use aspects to request a specific size to the compiler. In this case, we could use the Object_Size aspect:

Listing 151: p.ads

```ada
package P is

  type R is record
    A : Integer;
    B : Integer;
  end record
  with Object_Size => 72;
  end record;

  for R use record
    A at 0 range 0 .. 31;

end P;
```
Derived Types And Representation Clauses

In some cases, you might want to modify the memory representation of a record without impacting existing code. For example, you might want to use a record type that was declared in a package that you're not allowed to change. Also, you would like to modify its memory representation in your application. A nice strategy is to derive a type and use a representation clause for the derived type.

We can apply this strategy on our previous example. Let's say we would like to use record type R from package P in our application, but we're not allowed to modify package P — or the record type, for that matter. In this case, we could simply derive R as R_New and use a representation clause for R_New. This is exactly what we do in the specification of the child package P.Rep:

```ada
package P is
    type R is record
        A : Integer;
        B : Integer;
    end record;
end P;
```

If the code compiles, R'Size and R'Object_Size should now have the same value.
Learning Ada

Listing 154: p-rep.ads

```ada
package P.Rep is

  type R_New is new R
    with Object_Size => 72;

  for R_New use
  record
    A at 0 range 0 .. 31;
    B at 5 range 0 .. 31;
  end record;

end P.Rep;
```

Listing 155: show_empty_byte.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with P; use P;
with P.Rep; use P.Rep;

procedure Show_Empty(Byte is
begin
  Put_Line ("R'Size: " & R'Size'Image);
  Put_Line ("R'Object_Size: " & R'Object_Size'Image);
  Put_Line ("R_New'Size: " & R_New'Size'Image);
  Put_Line ("R_New'Object_Size: " & R_New'Object_Size'Image);
end Show_Empty_BYTE;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Derived_Rep_Clauses_Empty_Byte
MD5: 3a1e0837f8bd8250f20fc7b274b869d5

Runtime output

```
R'Size: 64
R'Object_Size: 64
R_New'Size: 72
R_New'Object_Size: 72
```

When running this example, we see that the R type retains the memory representation selected by the compiler for the target architecture, while the R_New has the memory representation that we specified.

25.2. Types and Representation 373
Representation on Bit Level

A very common application of representation clauses is to specify individual bits of a record. This is particularly useful, for example, when mapping registers or implementing protocols. Let's consider the following fictitious register as an example:

<table>
<thead>
<tr>
<th>bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>S</td>
<td>(reserved)</td>
<td>Error</td>
<td>V1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, S is the current status, Error is a flag, and V1 contains a value. Due to the fact that we can use representation clauses to describe individual bits of a register as records, the implementation becomes as simple as this:

```ada
package P is
  type Status is (Ready, Waiting, Processing, Done);
  type UInt_3 is range 0 .. 2 ** 3 - 1;
  type Simple_Reg is record
    S : Status;
    Error : Boolean;
    V1 : UInt_3;
  end record;
  for Simple_Reg use record
    S at 0 range 0 .. 1;
    -- Bit #2 and 3: reserved!
    Error at 0 range 4 .. 4;
    V1 at 0 range 5 .. 7;
  end record;
end P;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
with P; use P;
procedure Show_Simple_Reg is
begin
  Put_Line ("Simple_Reg'Size: " & Simple_Reg'Size'Image);
  Put_Line ("Simple_Reg'Object_Size: " & Simple_Reg'Object_Size'Image);
end Show_Simple_Reg;
```

Code block metadata
As we can see in the declaration of the Simple_Reg type, each component represents a field from our register, and it has a fixed location (which matches the register representation we see in the graph above). Any operation on the register is as simple as accessing the record component. For example:

```
with Ada.Text_IO; use Ada.Text_IO;
with P; use P;

procedure Show_Simple_Reg is
  Default : constant Simple_Reg :=
    (S => Ready,
     Error => False,
     V1 => 0);

  R : Simple_Reg := Default;

begin
  Put_Line ("R.S: " & R.S'Image);
  R.V1 := 4;
  Put_Line ("R.V1: " & R.V1'Image);
end Show_Simple_Reg;
```

As we can see in the example, to retrieve the current status of the register, we just have to write R.S. To update the V1 field of the register with the value 4, we just have to write R.V1 := 4. No extra code — such as bit-masking or bit-shifting — is needed here.

**In other languages**

Some programming languages require that developers use complicated, error-prone approaches — which may include manually bit-shifting and bit-masking variables — to retrieve information from or store information to individual bits or registers. In Ada, however, this is efficiently handled by the compiler, so that developers only need to correctly describe the register mapping using representation clauses.
25.2.4 Changing Data Representation

Note: This section was originally written by Robert Dewar and published as Gem #27: Changing Data Representation\(^{40}\) and Gem #28\(^{41}\).

A powerful feature of Ada is the ability to specify the exact data layout. This is particularly important when you have an external device or program that requires a very specific format. Some examples are:

Listing 159: communication.ads

```ada
package Communication is

    type Com_Packet is record
        Key : Boolean;
        Id : Character;
        Val : Integer range 100 .. 227;
    end record;

    for Com_Packet use record
        Key at 0 range 0 .. 0;
        Id at 0 range 1 .. 8;
        Val at 0 range 9 .. 15;
    end record;

end Communication;
```

Code block metadata

MD5: cbd7f5547c5b0458853ac21d03aa41f8

Build output

communication.ads:12:11: warning: component clause forces biased representation, for "Val" [-gнатw.b]

which lays out the fields of a record, and in the case of Val, forces a biased representation in which all zero bits represents 100. Another example is:

Listing 160: array_representation.ads

```ada
package Array_Representation is

    type Val is (A, B, C, D, E, F, G, H);

    type Arr is array (1 .. 16) of Val
    with Component_Size => 3;

end Array_Representation;
```

Code block metadata

MD5: 7eb17fc2cd415acb7c53a363fa336807

\(^{40}\) https://www.adacore.com/gems/gem-27

\(^{41}\) https://www.adacore.com/gems/gem-28
which forces the components to take only 3 bits, crossing byte boundaries as needed. A final example is:

Listing 161: enumeration_representation.ads

```ada
package Enumeration_Representation is
  type Status is (Off, On, Unknown);
  for Status use (Off => 2#001#,
                  On    => 2#010#,
                  Unknown => 2#100#);
end Enumeration_Representation;
```

which allows specified values for an enumeration type, instead of the efficient default values of 0, 1, 2.

In all these cases, we might use these representation clauses to match external specifications, which can be very useful. The disadvantage of such layouts is that they are inefficient, and accessing individual components, or, in the case of the enumeration type, looping through the values can increase space and time requirements for the program code.

One approach that is often effective is to read or write the data in question in this specified form, but internally in the program represent the data in the normal default layout, allowing efficient access, and do all internal computations with this more efficient form.

To follow this approach, you will need to convert between the efficient format and the specified format. Ada provides a very convenient method for doing this, as described in RM 13.6 "Change of Representation"42.

The idea is to use type derivation, where one type has the specified format and the other has the normal default format. For instance for the array case above, we would write:

Listing 162: array_representation.ads

```ada
package Array_Representation is
  type Val is (A, B, C, D, E, F, G, H);
  type Arr is array (1 .. 16) of Val;
  type External_Arr is new Arr
    with Component_Size => 3;
end Array_Representation;
```

Now we read and write the data using the External_Arr type. When we want to convert to the efficient form, Arr, we simply use a type conversion.

Listing 163: using_array_for_io.adb

```ada
with Array_Representation;
use Array_Representation;

procedure Using_Array_For_IO is
  Input_Data : External_Arr;
  Work_Data : Arr;
  Output_Data : External_Arr;
begin
  -- (read data into Input_Data)
  -- Now convert to internal form
  Work_Data := Arr (Input_Data);
  -- (computations using efficient
  -- Work_Data form)
  -- Convert back to external form
  Output_Data := External_Arr (Work_Data);
end Using_Array_For_IO;
```

Using this approach, the quite complex task of copying all the data of the array from one form to another, with all the necessary masking and shift operations, is completely automatic.

Similar code can be used in the record and enumeration type cases. It is even possible to specify two different representations for the two types, and convert from one form to the other, as in:

Listing 164: enumeration_representation.ads

```ada
package Enumeration_Representation is
  type Status_In is (Off, On, Unknown);
  type Status_Out is new Status_In;
  for Status_In use (Off => 2#001#,
                     On   => 2#010#,
                     Unknown => 2#100#);
  for Status_Out use (Off   => 103,
                      On    => 1045,
                      Unknown => 7700);
end Enumeration_Representation;
```

(continues on next page)
There are two restrictions that must be kept in mind when using this feature. First, you have to use a derived type. You can't put representation clauses on subtypes, which means that the conversion must always be explicit. Second, there is a rule RM 13.1\textsuperscript{43} (10) that restricts the placement of interesting representation clauses:

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

All the representation clauses that are interesting from the point of view of change of representation are "type related", so for example, the following sequence would be illegal:

```
Listing 165: array_representation.ads

package Array_Representation is

  type Val is (A, B, C, D, E, F, G, H);
  type Arr is array (1 .. 16) of Val;

  procedure Rearrange (Arg : in out Arr);

  type External_Arr is new Arr
  with Component_Size => 3;

end Array_Representation;
```

Why these restrictions? Well, the answer is a little complex, and has to do with efficiency considerations, which we will address below.

**Restrictions**

In the previous subsection, we discussed the use of derived types and representation clauses to achieve automatic change of representation. More accurately, this feature is not completely automatic, since it requires you to write an explicit conversion. In fact there is a principle behind the design here which says that a change of representation should never occur implicitly behind the back of the programmer without such an explicit request by means of a type conversion.

The reason for that is that the change of representation operation can be very expensive, since in general it can require component by component copying, changing the representation on each component.

\textsuperscript{43} http://www.ada-auth.org/standards/22rm/html/RM-13-1.html
Let's have a look at the -gnatG expanded code to see what is hidden under the covers here. For example, the conversion Arr (Input_Data) from the previous example generates the following expanded code:

```ada
B26b : declare
    [subtype p__TarrD1 is integer range 1 .. 16]
    R25b : p__TarrD1 := 1;
begin
    for L24b in 1 .. 16 loop
        [subtype p__arr___XP3 is
            system_unsigned_types__long_long_unsigned range 0 ..
            16#FFFF_FFFF_FFFF#
        ]
        work_data := p__arr___XP3!(work_data and not shift_left!(
            16#7#, 3 * (integer(L24b - 1)))) or shift_left!(p__arr___XP3!
            (input_data (R25b)), 3 * (integer(L24b - 1))));
        R25b := p__TarrD1'succ(R25b);
    end loop;
end B26b;
```

That's pretty horrible! In fact, we could have simplified it for this section, but we have left it in its original form, so that you can see why it is nice to let the compiler generate all this stuff so you don't have to worry about it yourself.

Given that the conversion can be pretty inefficient, you don't want to convert backwards and forwards more than you have to, and the whole approach is only worthwhile if we'll be doing extensive computations involving the value.

The expense of the conversion explains two aspects of this feature that are not obvious. First, why do we require derived types instead of just allowing subtypes to have different representations, avoiding the need for an explicit conversion?

The answer is precisely that the conversions are expensive, and you don't want them happening behind your back. So if you write the explicit conversion, you get all the gobbledegook listed above, but you can be sure that this never happens unless you explicitly ask for it.

This also explains the restriction we mentioned in previous subsection from RM 13.1\textsuperscript{44} (10):

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

It turns out this restriction is all about avoiding implicit changes of representation. Let's have a look at how type derivation works when there are primitive subprograms defined at the point of derivation. Consider this example:

```ada
package My_Ints is

    type My_Int_1 is range 1 .. 10;

    function Odd (Arg : My_Int_1) return Boolean;

    type My_Int_2 is new My_Int_1;

end My_Ints;
```

\textsuperscript{44} http://www.ada-auth.org/standards/22rm/html/RM-13-1.html
Now when we do the type derivation, we inherit the function Odd for My_Int_2. But where does this function come from? We haven't written it explicitly, so the compiler somehow materializes this new implicit function. How does it do that?

We might think that a complete new function is created including a body in which My_Int_2 replaces My_Int_1, but that would be impractical and expensive. The actual mechanism avoids the need to do this by use of implicit type conversions. Suppose after the above declarations, we write:

```
with My_Ints; use My_Ints;

procedure Using_My_Int is
  Var : My_Int_2;
begin
  if Odd (Var) then
    null;
  end if;
end Using_My_Int;
```

The compiler translates this as:

```
with My_Ints; use My_Ints;

procedure Using_My_Int is
  Var : My_Int_2;
begin
  if Odd (Var) then
    null;
  end if;
end Using_My_Int;
```

(continues on next page)
if Odd (My_Int_1 (Var)) then
   -- ^ Converting My_Int_2 to
   -- My_Int_1 type before
calling Odd function.
null;
end if;
end Using_My_Int;

This implicit conversion is a nice trick, it means that we can get the effect of inheriting a new operation without actually having to create it. Furthermore, in a case like this, the type conversion generates no code, since My_Int_1 and My_Int_2 have the same representation.

But the whole point is that they might not have the same representation if one of them had a representation clause that made the representations different, and in this case the implicit conversion inserted by the compiler could be expensive, perhaps generating the junk we quoted above for the Arr case. Since we never want that to happen implicitly, there is a rule to prevent it.

The business of forbidding by-reference types (which includes all tagged types) is also driven by this consideration. If the representations are the same, it is fine to pass by reference, even in the presence of the conversion, but if there was a change of representation, it would force a copy, which would violate the by-reference requirement.

So to summarize this section, on the one hand Ada gives you a very convenient way to trigger these complex conversions between different representations. On the other hand, Ada guarantees that you never get these potentially expensive conversions happening unless you explicitly ask for them.

### 25.2.5 Valid Attribute

When receiving data from external sources, we're subjected to problems such as transmission errors. If not handled properly, erroneous data can lead to major issues in an application.

One of those issues originates from the fact that transmission errors might lead to invalid information stored in memory. When proper checks are active, using invalid information is detected at runtime and an exception is raised at this point, which might then be handled by the application.

Instead of relying on exception handling, however, we could instead ensure that the information we're about to use is valid. We can do this by using the Valid attribute. For example, if we have a variable Var, we can verify that the value stored in Var is valid by writing Var’Valid, which returns a Boolean value. Therefore, if the value of Var isn't valid, Var’Valid returns False, so we can have code that handles this situation before we actually make use of Var. In other words, instead of handling a potential exception in other parts of the application, we can proactively verify that input information is correct and avoid that an exception is raised.
In the next example, we show an application that

- generates a file containing mock-up data, and then
- reads information from this file as state values.

The mock-up data includes valid and invalid states.

Listing 170: create_test_file.ads

```ada
procedure Create_Test_File (File_Name : String);
```

Listing 171: create_test_file.adb

```ada
with Ada.Sequential_IO;

procedure Create_Test_File (File_Name : String)
is
  package Integer_Sequential_IO is new Ada.Sequential_IO (Integer);
  use Integer_Sequential_IO;

  F : File_Type;
begin
  Create (F, Out_File, File_Name);
  Write (F, 1);
  Write (F, 2);
  Write (F, 4);
  Write (F, 3);
  Write (F, 2);
  Write (F, 10);
  Close (F);
end Create_Test_File;
```

Listing 172: states.ads

```ada
with Ada.Sequential_IO;

package States is

  type State is (Off, On, Waiting)
  with Size => Integer'Size;

  for State use (Off   => 1,
                 On    => 2,
                 Waiting => 4);

  package State_Sequential_IO is new Ada.Sequential_IO (State);

  procedure Read_Display_States
  (File_Name : String);

end States;
```

Listing 173: states.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body States is

  procedure Read_Display_States
```

(continues on next page)
Listing 174: show_states_from_file.adb

```ada
with States;  use States;
with Create_Test_File;

procedure Show_States_From_File is
  File_Name : constant String := "data.bin";
begin
  Create_Test_File (File_Name);
  Read_Display_States (File_Name);
end Show_States_From_File;
```

Let's start our discussion on this example with the States package, which contains the declaration of the State type. This type is a simple enumeration containing three states: Off, On and Waiting. We're assigning specific integer values for this type by declaring an
Learning Ada

enumeration representation clause. Note that we're using the Size aspect to request that objects of this type have the same size as the **Integer** type. This becomes important later on when parsing data from the file.

In the **Create_Test_File** procedure, we create a file containing integer values, which is parsed later by the **Read_Display_States** procedure. The **Create_Test_File** procedure doesn't contain any reference to the State type, so we're not constrained to just writing information that is valid for this type. On the contrary, this procedure makes use of the **Integer** type, so we can write any integer value to the file. We use this strategy to write both valid and invalid values of State to the file. This allows us to simulate an environment where transmission errors occur.

We call the **Read_Display_States** procedure to read information from the file and display each state stored in the file. In the main loop of this procedure, we call Read to read a state from the file and store it in the **S** variable. We then call the nested **Display_State** procedure to display the actual state stored in **S**. The most important line of code in the **Display_State** procedure is the one that uses the **Valid** attribute:

```
if S'Valid then
```

In this line, we're verifying that the **S** variable contains a valid state before displaying the actual information from **S**. If the value stored in **S** isn't valid, we can handle the issue accordingly. In this case, we're simply displaying a message indicating that an invalid value was detected. If we didn't have this check, the **Constraint_Error** exception would be raised when trying to use invalid data stored in **S** — this would happen, for example, after reading the integer value 3 from the input file.

In summary, using the **Valid** attribute is a good strategy we can employ when we know that information stored in memory might be corrupted.

**In the Ada Reference Manual**

- 13.9.2 The **Valid** Attribute

---

### 25.2.6 Unchecked Union

We've introduced variant records back in the *Introduction to Ada course* (page 104). In simple terms, a variant record is a record with discriminants that allows for changing its structure. Basically, it's a record containing a **case**.

The **State_Or_Integer** declaration in the **States** package below is an example of a variant record:

```
package States is

  type State is (Off, On, Waiting)
  with Size => Integer'Size;

  for State use (Off => 1,
                 On => 2,
                 Waiting => 4);

  type State_Or_Integer (Use_Enum : Boolean) is record

    case Use_Enum is

    end States;

Listing 175: states.ads
```

(continues on next page)
when False => I : Integer;
when True  => S : State;
end case;
end record;

procedure Display_State_Value
(V : State_Or_Integer);
end States;

Listing 176: states.adb

with Ada.Text_IO; use Ada.Text_IO;
package body States is
  procedure Display_State_Value
  (V : State_Or_Integer)
  is
    begin
      Put_Line ("State: " & V.S'Image);
      Put_Line ("Value: " & V.I'Image);
    end Display_State_Value;
end States;

Code block metadata
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_Or_Integer
MD5: fa72f52a4396a2e66931ff6932c567fc

As mentioned in the previous course, if you try to access a component that is not valid for your record, a Constraint_Error exception is raised. For example, in the implementation of the Display_State_Value procedure, we're trying to retrieve the value of the integer component (I) of the V record. When calling this procedure, the Constraint_Error exception is raised as expected because Use_Enum is set to True, so that the I component is invalid — only the S component is valid in this case.

Listing 177: show_variant_rec_error.adb

with States; use States;
procedure Show_Variant_Rec_Error is
  V : State_Or_Integer (Use_Enum => True);
begin
  V.S := On;
  Display_State_Value (V);
end Show_Variant_Rec_Error;

Code block metadata
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_Or_Integer
MD5: b8cf215dd55bfdec6950df35c7bc19b9

Runtime output
State: ON
raised CONSTRAINT_ERROR : states.adb:10 discriminant check failed
In addition to not being able to read the value of a component that isn't valid, assigning a value to a component that isn't valid also raises an exception at runtime. In this example, we cannot assign to V.I:

```ada
with States; use States;

procedure Show_Variant_Rec_Error is
  V : State_Or_Integer (Use_Enum => True);
begin
  V.I := 4;
  -- Error: V.I cannot be accessed because
  -- Use_Enum is set to True.
end Show_Variant_Rec_Error;
```

We may circumvent this limitation by using the Unchecked_Union aspect. For example, we can derive a new type from State_Or_Integer and use this aspect in its declaration. We do this in the declaration of the Unchecked_State_Or_Integer type below.

```ada
package States is
  type State is (Off, On, Waiting)
    with Size => Integer'Size;
  for State use (Off => 1, On => 2, Waiting => 4);

  type State_Or_Integer (Use_Enum : Boolean) is
    record
      case Use_Enum is
        when False => I : Integer;
        when True => S : State;
      end case;
    end record;

  type Unchecked_State_Or_Integer (Use_Enum : Boolean) is new
    State_Or_Integer (Use_Enum)
  new
    States.Unchecked_Union
end States;
```
with Unchecked_Union;

procedure Display_State_Value
  (V: Unchecked_State_Or_Integer);
end States;

with Ada.Text_IO; use Ada.Text_IO;

package body States is

  procedure Display_State_Value
    (V: Unchecked_State_Or_Integer)
  is
    begin
      Put_Line ("State: " & V.S'Image);
      Put_Line ("Value: " & V.I'Image);
      end Display_State_Value;
end States;

Because we now use the Unchecked_State_Or_Integer type for the input parameter of the Display_State_Value procedure, no exception is raised at runtime, as both components are now accessible. For example:

with States; use States;

  procedure Show_Unchecked_Union is
    V: State_Or_Integer (Use_Enum => True);
    begin
      V.S := On;
      Display_State_Value
        (Unchecked_State_Or_Integer (V));
      end Show_Unchecked_Union;

Note that, in the call to the Display_State_Value procedure, we first need to convert the V argument from the State_Or_Integer to the Unchecked_State_Or_Integer type.

Also, we can assign to any of the components of a record that has the Unchecked_Union aspect. In our example, we can now assign to both the S and the I components of the V
record:

Listing 182: show_unchecked_union.adb

```ada
with States; use States;

procedure Show_Unchecked_Union is
  V : Unchecked_State_Or_Integer
    (Use_Enum => True);
begin
  V := (Use_Enum => True, S => On);
  Display_State_Value (V);
  V := (Use_Enum => False, I => 4);
  Display_State_Value (V);
end Show_Unchecked_Union;
```

Code block metadata

MD5: bb472e91c5e7b7e63d6246dbcf5226a0

Runtime output

State: ON
Value: 2
State: WAITING
Value: 4

In the example above, we're use an aggregate in the assignments to V. By doing so, we avoid that Use_Enum is set to the wrong component. For example:

Listing 183: show_unchecked_union.adb

```ada
with States; use States;

procedure Show_Unchecked_Union is
  V : Unchecked_State_Or_Integer
    (Use_Enum => True);
begin
  V.S := On;
  Display_State_Value (V);
  V.I := 4;
  -- Error: cannot directly assign to V.I,
  -- as Use_Enum is set to True.
  Display_State_Value (V);
end Show_Unchecked_Union;
```

Code block metadata

MD5: 74ac11a3effd0fd3959ffface295a86da

Build output

show_unchecked_union.adb:10:05: warning: component not present in subtype of
"Unchecked_State_Or_Integer" defined at line 4 [enabled by default]
show_unchecked_union.adb:10:05: warning: Constraint_Error will be raised at run_time [enabled by default]

25.2. Types and Representation 389
Learning Ada

Runtime output

State: ON
Value: 2
raised CONSTRAINT_ERROR : show_unchecked_union.adb:10 discriminant check failed

Here, even though the record has the Unchecked_Union attribute, we cannot directly assign to the I component because Use_Enum is set to True, so only the S is accessible. We can, however, read its value, as we do in the Display_State_Value procedure.

Be aware that, due to the fact the union is not checked, we might write invalid data to the record. In the example below, we initialize the I component with 3, which is a valid integer value, but results in an invalid value for the S component, as the value 3 cannot be mapped to the representation of the State type.

Listing 184: show_unchecked_union.adb

```
with States; use States;

procedure Show_Unchecked_Union is
   V : Unchecked_State_Or_Integer
      (Use_Enum => True);
begin
   V := (Use_Enum => False, I => 3);
   Display_State_Value (V);
end Show_Unchecked_Union;
```

Code block metadata

Unchecked_State_Or_Integer
MD5: f63e64df137cfc3c29e41f784306f0e4

Runtime output

raised CONSTRAINT_ERROR : states.adb:9 invalid data

To mitigate this problem, we could use the Valid attribute — discussed in the previous section — for the S component before trying to use its value in the implementation of the Display_State_Value procedure:

Listing 185: states.adb

```
with Ada.Text_IO; use Ada.Text_IO;

package body States is

      procedure Display_State_Value
         (V : Unchecked_State_Or_Integer)
      is
    begin
      if V.S'Valid then
         Put_Line ("State: " & V.S'Image);
      else
         Put_Line ("State: <invalid>");
      end if;
      Put_Line ("Value: " & V.I'Image);
    end Display_State_Value;
```

Chapter 25. Data types
Listing 186: show_unchecked_union.adb

```ada
with States; use States;

procedure Show_Unchecked_Union is
  V : Unchecked_State_Or_Integer
    (Use_Enum => True);
begin
  V := (Use_Enum => False, I => 3);
  Display_State_Value (V);
end Show_Unchecked_Union;
```

However, in general, you should avoid using the Unchecked_Union aspect due to the potential issues you might introduce into your application. In the majority of the cases, you don't need it at all — except for special cases such as when interfacing with C code that makes use of union types or solving very specific problems when doing low-level programming.

In the Ada Reference Manual
- B.3.3 Unchecked Union Types

25.2.7 Shared variable control

Ada has built-in support for handling both volatile and atomic data. Let's start by discussing volatile objects.

In the Ada Reference Manual
- C.6 Shared Variable Control

Volatile

A volatile object can be described as an object in memory whose value may change between two consecutive memory accesses of a process A — even if process A itself hasn't changed the value. This situation may arise when an object in memory is being shared by multiple threads. For example, a thread B may modify the value of that object between two read accesses of a thread A. Another typical example is the one of memory-mapped I/O, where the hardware might be constantly changing the value of an object in memory.

Because the value of a volatile object may be constantly changing, a compiler cannot generate code to store the value of that object in a register and then use the value from the register in subsequent operations. Storing into a register is avoided because, if the value is stored there, it would be outdated if another process had changed the volatile object in the meantime. Instead, the compiler generates code in such a way that the process must read the value of the volatile object from memory for each access.

Let's look at a simple example:

---

49 [https://en.wikipedia.org/wiki/Memory-mapped_I/O](https://en.wikipedia.org/wiki/Memory-mapped_I/O)
Listing 187: show_volatile_object.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Volatile_Object is
  Val : Long_Float with Volatile;
begin
  Val := 0.0;
  for I in 0 .. 999 loop
    Val := Val + 2.0 * Long_Float (I);
  end loop;
  Put_Line ("Val: " & Long_Float'Image (Val));
end Show_Volatile_Object;
```

**Runtime output**

Val: 9.99000000000000E+05

In this example, Val has the Volatile aspect, which makes the object volatile. We can also use the Volatile aspect in type declarations. For example:

Listing 188: shared_var_types.ads

```ada
package Shared_Var_Types is
  type Volatile_Long_Float is new Long_Float with Volatile;
end Shared_Var_Types;
```

Listing 189: show_volatile_type.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Shared_Var_Types; use Shared_Var_Types;

procedure Show_Volatile_Type is
  Val : Volatile_Long_Float;
begin
  Val := 0.0;
  for I in 0 .. 999 loop
    Val := Val + 2.0 * Volatile_Long_Float (I);
  end loop;
  Put_Line ("Val: " & Volatile_Long_Float'Image (Val));
end Show_Volatile_Type;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Volatile_Object_Ada
MD5: aa1e276e64e69013bfc3e3ef39f3dd47

**Runtime output**

Runtime output
Here, we're declaring a new type `Volatile_Long_Float` in the `Shared_Var_Types` package. This type is based on the `Long_Float` type and uses the Volatile aspect. Any object of this type is automatically volatile.

In addition to that, we can declare components of an array to be volatile. In this case, we can use the Volatile_Components aspect in the array declaration. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Volatile_Array_Components is
  Arr : array (1 .. 2) of Long_Float
    with Volatile_Components;
begin
  Arr := (others => 0.0);
  for I in 0 .. 999 loop
    Arr (1) := Arr (1) + 2.0 * Long_Float (I);
    Arr (2) := Arr (2) + 10.0 * Long_Float (I);
  end loop;
  Put_Line ("Arr (1): " & Float'Image (Arr (1)));
  Put_Line ("Arr (2): " & Float'Image (Arr (2)));
end Show_Volatile_Array_Components;
```

Note that it's possible to use the Volatile aspect for the array declaration as well:

```ada
package Shared_Var_Types is
private
  Arr : array (1 .. 2) of Long_Float
    with Volatile;
end Shared_Var_Types;
```

Note that, if the Volatile aspect is specified for an object, then the Volatile_Components aspect is also specified automatically — if it makes sense in the context, of course. In the
example above, even though Volatile_Components isn't specified in the declaration of the Arr array, it's automatically set as well.

**Independent**

When you write code to access a single object in memory, you might actually be accessing multiple objects at once. For example, when you declare types that make use of representation clauses — as we've seen in previous sections —, you might be accessing multiple objects that are grouped together in a single storage unit. For example, if you have components A and B stored in the same storage unit, you cannot update A without actually writing (the same value) to B. Those objects aren't independently addressable because, in order to access one of them, we have to actually address multiple objects at once.

When an object is independently addressable, we call it an independent object. In this case, we make sure that, when accessing that object, we won't be simultaneously accessing another object. As a consequence, this feature limits the way objects can be represented in memory, as we'll see next.

To indicate that an object is independent, we use the Independent aspect:

```ada
package Shared_Var_Types is
  I : Integer with Independent;
end Shared_Var_Types;
```

Similarly, we can use this aspect when declaring types:

```ada
package Shared_Var_Types is
  type Independent_Boolean is new Boolean with Independent;
  type Flags is record
    F1 : Independent_Boolean;
    F2 : Independent_Boolean;
  end record;
end Shared_Var_Types;
```

In this example, we're declaring the Independent_Boolean type and using it in the declaration of the Flag record type. Let's now derive the Flags type and use a representation clause for the derived type:
As you can see when trying to compile this example, the representation clause that we used for Rep_Flags isn't following these limitations:

1. The size of each independent component must be a multiple of a storage unit.
2. The start position of each independent component must be a multiple of a storage unit.

For example, for architectures that have a storage unit of one byte — such as standard desktop computers —, this means that the size and the position of independent components must be a multiple of a byte. Let's correct the issues in the code above by:

- setting the size of each independent component to correspond to Storage_Unit — using a range between 0 and Storage_Unit - 1 —, and
- setting the start position to zero.

This is the corrected version:

```
Listing 195: shared_var_types-representation.ads

with System;

package Shared_Var_Types.Representation is

  type Rep_Flags is new Flags;

  for Rep_Flags use record
    F1 at 0 range 0 .. System.Storage_Unit - 1;
    F2 at 1 range 0 .. System.Storage_Unit - 1;
  end record;

end Shared_Var_Types.Representation;
```
end record;
end Shared_Var_Types.Representation;

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Independent_Type
MD5: ed57e57cd7466980909a4f7ce40a29dfc

Note that the representation that we're now using for Rep_FLAGS is most likely the representation that the compiler would have chosen for this data type. We could, however, have added an empty storage unit between F1 and F2 — by simply writing F2 at 2 ...:

Listing 196: shared_var_types-representation.ads

```ada
with System;
package Shared_Var_Types.Representation is
  type Rep_FLAGS is new Flags;
  for Rep_FLAGS use record
    F1 at 0 range 0 .. System.Storage_Unit - 1;
    F2 at 2 range 0 .. System.Storage_Unit - 1;
  end record;
end Shared_Var_Types.Representation;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Independent_Type
MD5: 71fedf8aac7c19bca1ba3b487efa9b17

As long as we follow the rules for independent objects, we're still allowed to use representation clauses that don't correspond to the one that the compiler might select.

For arrays, we can use the Independent_Components aspect:

Listing 197: shared_var_types.ads

```ada
package Shared_Var_Types is
  Flags : array (1 .. 8) of Boolean
  with Independent_Components;
end Shared_Var_Types;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Independent_Components
MD5: b331d0a13adf45624b664839fe4ba42c

We've just seen in a previous example that some representation clauses might not work with objects and types that have the Independent aspect. The same restrictions apply when we use the Independent_Components aspect. For example, this aspect prevents that array components are packed when the Pack aspect is used. Let's discuss the following erroneous code example:
Learning Ada

Listing 198: shared_var_types.ads

```ada
package Shared_Var_Types is

  type Flags is
    array (Positive range <>) of Boolean
    with Independent_Components, Pack;

  F : Flags (1 .. 8) with Size => 8;

end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Packed_Independent_Components
MD5: dbaff4f2559ef8a449dad251f42cddc0

Build output

shared_var_types.ads:5:37: warning: cannot pack independent components (RM 13.2(7))
shared_var_types.ads:7:36: error: size for "F" too small, minimum allowed is 64

gprbuild: *** compilation phase failed

As expected, this code doesn't compile. Here, we can have either independent components, or packed components. We cannot have both at the same time because packed components aren't independently addressable. The compiler warns us that the Pack aspect won't have any effect on independent components. When we use the Size aspect in the declaration of F, we confirm this limitation. If we remove the Size aspect, however, the code is compiled successfully because the compiler ignores the Pack aspect and allocates a larger size for F:

Listing 199: shared_var_types.ads

```ada
package Shared_Var_Types is

  type Flags is
    array (Positive range <>) of Boolean
    with Independent_Components;

end Shared_Var_Types;
```

Listing 200: show_flags_size.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System;
with Shared_Var_Types; use Shared_Var_Types;

procedure Show_Flags_Size is
  F : Flags (1 .. 8);

begin
  Put_Line ("Flags'Size: "
             & F'Size'Image & " bits");
  Put_Line ("Flags (1)'Size: "
             & F (1)'Size'Image & " bits");
  Put_Line ("# storage units: "
             & Integer'Image
             (F'Size / System.Storage_Unit));
end Show_Flags_Size;
```

25.2. Types and Representation 397
Learning Ada

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Packed_Independent_Components
MD5: b96f921b08b1d8207749517f833fc121

Build output

show_flags_size.adb:7:04: warning: variable "F" is read but never assigned [-gnatw]
shared_var_types.ads:5:37: warning: cannot pack independent components (RM 13.2(7))

Runtime output

| Flags'Size:    | 64 bits  |
| Flags (1)'Size: | 8 bits  |
| # storage units: | 8        |

As you can see in the output of the application, even though we specify the Pack aspect for the Flags type, the compiler allocates eight storage units, one per each component of the F array.

Atomic

An atomic object is an object that only accepts atomic reads and updates. The Ada standard specifies that "for an atomic object (including an atomic component), all reads and updates of the object as a whole are indivisible." In this case, the compiler must generate Assembly code in such a way that reads and updates of an atomic object must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes.

In other contexts

Generally, we can say that operations are said to be atomic when they can be completed without interruptions. This is an important requirement when we're performing operations on objects in memory that are shared between multiple processes.

This definition of atomicity above is used, for example, when implementing databases. However, for this section, we're using the term "atomic" differently. Here, it really means that reads and updates must be performed with a single Assembly instruction.

For example, if we have a 32-bit object composed of four 8-bit bytes, the compiler cannot generate code to read or update the object using four 8-bit store / load instructions, or even two 16-bit store / load instructions. In this case, in order to maintain atomicity, the compiler must generate code using one 32-bit store / load instruction.

Because of this strict definition, we might have objects for which the Atomic aspect cannot be specified. Lots of machines support integer types that are larger than the native word-sized integer. For example, a 16-bit machine probably supports both 16-bit and 32-bit integers, but only 16-bit integer objects can be marked as atomic — or, more generally, only objects that fit into at most 16 bits.

Atomicity may be important, for example, when dealing with shared hardware registers. In fact, for certain architectures, the hardware may require that memory-mapped registers are handled atomically. In Ada, we can use the Atomic aspect to indicate that an object is atomic. This is how we can use the aspect to declare a shared hardware register:
with System;

package Shared_Var_Types is

private
    R : Integer
        with Atomic,
        Address =>
            System'To_Address (16#FFFF00A0#);

end Shared_Var_Types;

Code block metadata

Project: Courses.Advanced_Ada.Data Types.Type Representation.Shared Variable_.
Control.Atomic Object
MD5: 5c2d8e0a9615084c2a15f896c61adaa6

Note that the Address aspect allows for assigning a variable to a specific location in the
memory. In this example, we're using this aspect to specify the address of the memory-
mapped register.

Later on, we talk again about the Address aspect (page 403) and the GNAT-specific System'To_Address attribute (page 404).

In addition to atomic objects, we can declare atomic types — similar to what we've seen before for volatile objects. For example:

with System;

package Shared_Var_Types is

type Atomic_Integer is new Integer
    with Atomic;

private
    R : Atomic_Integer
        with Address =>
            System'To_Address (16#FFFF00A0#);

end Shared_Var_Types;

Code block metadata

Project: Courses.Advanced_Ada.Data Types.Type Representation.Shared Variable_..
Control.Atomic Types
MD5: 009632ba0155d70def8281ba590f3d12

In this example, we're declaring the Atomic_Integer type, which is an atomic type. Objects
of this type — such as R in this example — are automatically atomic.

We can also declare atomic array components:

package Shared_Var_Types is

private

(continues on next page)
Learning Ada

(continued from previous page)

```
Arr : array (1 .. 2) of Integer
   with Atomic_Components;
end Shared_Var_Types;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Atomic_Array_Components
MD5: 7501bdf618621a822d451da8d731ef75

This example shows the declaration of the Arr array, which has atomic components — the atomicity of its components is indicated by the Atomic_Components aspect.

Note that if an object is atomic, it is also volatile and independent. In other words, these type declarations are equivalent:

```
Listing 204: shared_var_types.ads

package Shared_Var_Types is

   type Atomic_Integer_1 is new Integer
      with Atomic;

   type Atomic_Integer_2 is new Integer
      with Atomic,
      Volatile,
      Independent;

end Shared_Var_Types;
```

**Code block metadata**

MD5: 3034c7a07698491f961d9b4fb74f03d8

A similar rule applies to components of an array. When we use the Atomic_Components, the following aspects are implied: Volatile, Volatile_Components and Independent_Components. For example, these array declarations are equivalent:

```
```
package Shared_Var_Types is
  Arr_1 : array (1 .. 2) of Integer
       with Atomic_Components;
  Arr_2 : array (1 .. 2) of Integer
       with Atomic_Components,
       Volatile,
       Volatile_Components,
       Independent_Components;
end Shared_Var_Types;

25.2.8 Addresses

In other languages, such as C, the concept of pointers and addresses plays a prominent role. (In fact, in C, many optimizations rely on the usage of pointer arithmetic.) The concept of addresses does exist in Ada, but it's mainly reserved for very specific applications, mostly related to low-level programming. In general, other approaches — such as using access types — are more than sufficient. (We discuss access types (page 735) in another chapter. Also, later on in that chapter, we discuss the relation between access types and addresses (page 849).) In this section, we discuss some details about using addresses in Ada.

We make use of the Address type, which is defined in the System package, to handle addresses. In contrast to other programming languages (such as C or C++), an address in Ada isn't an integer value: its definition depends on the compiler implementation, and it's actually driven directly by the hardware. For now, let's consider it to usually be a private type — this can be seen as an attempt to achieve application code portability, given the variations in hardware that result in different definitions of what an address actually is.

The Address type has support for address comparison (page 405) and address arithmetic (page 407) (also known as pointer arithmetic in C). We discuss these topics later in this section. First, let's talk about the Address attribute and the Address aspect.

In the Ada Reference Manual

- 13.7 The Package System\(^\text{50}\)

**Address attribute**

The Address attribute allows us to get the address of an object. For example:

Listing 206: use_address.adb

```ada
with System; use System;
procedure Use_Address is
  I : aliased Integer := 5;
  A : Address;
begin
  A := I'Address;
end Use_Address;
```


25.2. Types and Representation
Learning Ada

Here, we're assigning the address of the I object to the A address.

In the GNAT toolchain

GNAT offers a very useful extension to the System package to retrieve a string for an address: System.Address_Image. This is the function profile:

```ada
function System.Address_Image
  (A : System.Address) return String;
```

We can use this function to display the address in an user message, for example:

```
with Ada.Text_IO; use Ada.Text_IO;
with System.Address_Image;

procedure Show_Address_Attribute is
  I : aliased Integer := 5;
begin
  Put_Line ("Address : ");
  & System.Address_Image (I'Address));
end Show_Address_Attribute;
```

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes
- 13.7 The Package System

---

**Address aspect**

Usually, we let the compiler select the address of an object in memory, or let it use a register to store that object. However, we can specify the address of an object with the `Address` aspect. In this case, the compiler won't select an address automatically, but use the address that we're specifying. For example:

```
with System; use System;
with System.Address_Image;
with Ada.Text_I0; use Ada.Text_I0;

procedure Show_Address is
  I_Main  : aliased Integer;
  I_Mapped : Integer
    with Address => I_Main'Address;
begin
  Put_Line ("I_Main'Address : 
            & System.Address_Image
            (I_Main'Address));
  Put_Line ("I_Mapped'Address : 
            & System.Address_Image
            (I_Mapped'Address));
end Show_Address;
```

This approach allows us to create an overlay. For example:

```
with Ada.Text_I0; use Ada.Text_I0;

procedure Simple_Overlay is
  type State is (Off, State_1, State_2)
    with Size => Integer'Size;
  for State use (Off       => 0,
                 State_1    => 32,
                 State_2    => 64);

  S : State;
  I : Integer
    with Address => S'Address, Import, Volatile;
begin
  S := State_2;
  Put_Line ("I = " & Integer'Image (I));
end Simple_Overlay;
```
Learning Ada

MD5: a65057882518824d3ea173d193a7ae67

Runtime output

I = 64

Here, I is an overlay of S, as it uses S'Address. With this approach, we can either use the enumeration directly (by using the S object of State type) or its integer representation (by using the I variable).

In the GNAT toolchain

We could call the GNAT-specific System'\texttt{To\_Address} attribute when using the Address aspect, as we did while talking about the Atomic (page 398) aspect:

Listing 210: shared_var_types.ads

```ada
with System;

package Shared_Var_Types is
private
    R : Integer
    with Atomic,
        Address =>
            System'\texttt{To\_Address}(16\texttt{#FFFF00A0#});
end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Show_Access_Address
MD5: 5c2d8e0a9615084c2a15f896c61adaa6

In this case, R will refer to the address in memory that we're specifying (16\texttt{#FFFF00A0#} in this case).

As explained in the GNAT Reference Manual\(^{53}\), the System'\texttt{To\_Address} attribute denotes a function identical to To\_Address (from the System.Storage_Elements package) except that it is a static attribute. (We talk about the \texttt{To\_Address} function (page 406) function later on.)

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes\(^{54}\)
- 13.7 The Package System\(^{55}\)
- 13.7.1 The Package System.Storage_Elements\(^{56}\)

\(^{53}\) https://gcc.gnu.org/onlinedocs/gnat_rm/Attribute-To_005f005fAddress.html


\(^{56}\) http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html
Address comparison

We can compare addresses using the common comparison operators. For example:

Listing 211: show_address.adb

```ada
with System; use System;
with System.Address_Image;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Address is
  I, J : Integer;
begin
  Put_Line ("I'Address : " & System.Address_Image (I'Address));
  Put_Line ("J'Address : " & System.Address_Image (J'Address));
  if I'Address = J'Address then
    Put_Line ("I'Address = J'Address");
  elsif I'Address < J'Address then
    Put_Line ("I'Address < J'Address");
  else
    Put_Line ("I'Address > J'Address");
  end if;
end Show_Address;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Address_Aspect
MD5: 24ddb7d05159f26ef3b2ff6bcc2691e8

Runtime output

I'Address : 00007FFD81B2F2FC
J'Address : 00007FFD81B2F2F8
I'Address > J'Address

In this example, we compare the address of the I object with the address of the J object using the =, < and > operators.

In the Ada Reference Manual

- 13.7 The Package System\(^{57}\)

Address to integer conversion

The System.Storage_Elements package offers an integer representation of an address via the Integer_Address type, which is an integer type unrelated to common integer types such as Integer and Long_Integer. (The actual definition of Integer_Address is compiler-dependent, and it can be a signed or modular integer subtype.)

We can convert between the Address and Integer_Address types by using the To_Address and To_Integer functions. Let's see an example:

```
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;
with System.Address_Image; with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Address is
  I : Integer; A1, A2 : Address; IA : Integer_Address;
begin
  A1 := I'Address;
  IA := To_Integer (A1);
  A2 := To_Address (IA);
  Put_Line ("IA : " & Integer_Address'Image (IA));
  Put_Line ("A2 : " & System.Address_Image (A2));
end Show_Address;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_Arith_Ada
MD5: 69e053886fb8e8571d6c94247dc9f30f

Runtime output

A1 : 00007FFFB2EEB1AC
IA : 140736195375532
A2 : 00007FFFB2EEB1AC

Here, we retrieve the address of the I object and store it in the A1 address. Then, we convert A1 to an integer address by calling To_Integer (and store it in IA). Finally, we convert this integer address back to an actual address by calling To_Address.

In the Ada Reference Manual

- 13.7.1 The Package System.Storage_Elements

---

58 http://www.adalib.org/standards/22rm/html/RM-13-7-1.html
Address arithmetic

Although Ada supports address arithmetic, which we discuss in this section, it should be reserved for very specific applications such as low-level programming. However, even in situations that require close access to the underlying hardware, using address arithmetic might not be the approach you should consider — make sure to evaluate other options first!

Ada supports address arithmetic via the System.Storage_Elements package, which includes operators such as + and - for addresses. Let's see a code example where we iterate over an array by incrementing an address that points to each component in memory:

Listing 213: show_address.adb

```ada
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;
with System.Address_Image;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Address is
  Arr : array (1 .. 10) of Integer;
  A  : Address := Arr'Address;
  -- Initializing address object with address of the first component of Arr.
  -- We could write this as well:
  -- ___ := Arr (1)'Address
  begin
    for I in Arr'Range loop
    declare
      Curr : Integer
      with Address => A;
      begin
        Curr := I;
        Put_Line ("Curr'Address : ",
                   & System.Address_Image (Curr'Address));
    end;

    -- Address arithmetic
    -- A := A + Storage_Offset (Integer'Size) / Storage_Unit;
    -- Moving to next component
    end loop;
    for I in Arr'Range loop
      Put_Line ("Arr (",
                 & Integer'Image (I) & "," & Integer'Image (Arr (I)));
    end loop;
  end Show_Address;
```

Code block metadata
In this example, we initialize the address \( A \) by retrieving the address of the first component of the array \( \text{Arr} \). (Note that we could have written \( \text{Arr}(1)\)'Address instead of \( \text{Arr}'\text{Address} \). In any case, the language guarantees that \( \text{Arr}'\text{Address} \) gives us the address of the first component, i.e. \( \text{Arr}'\text{Address} = \text{Arr}(1)'\text{Address} \).)

Then, in the loop, we declare an overlay \( \text{Curr} \) using the current value of the \( A \) address. We can then operate on this overlay — here, we assign \( I \) to \( \text{Curr} \). Finally, in the loop, we increment address \( A \) and make it point to the next component in the \( \text{Arr} \) array — to do so, we calculate the size of an \textbf{Integer} component in storage units. (For details on storage units, see the section on storage size attribute (page 353).)

### In other languages

The code example above corresponds (more or less) to the following C code:

Listing 214: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int i;
    int arr[10];
    int *a = arr;
    /* int *a = &arr[0]; */
    for (i = 0; i < 10; i++) {
        *a++ = i;
        printf("curr address: %p\n", a);
    }
    for (i = 0; i < 10; i++)
}
```

(continues on next page)
While pointer arithmetic is very common in C, using address arithmetic in Ada is far from common, and it should be only used when it's really necessary to do so.

---

In the Ada Reference Manual

- **13.3 Operational and Representation Attributes**[^59]
- **13.7.1 The Package System.Storage.Elements**[^60]

[^60]: http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html

25.2. Types and Representation 409
25.2.9 Discarding names

As we know, we can use the Image attribute of a type to get a string associated with this type. This is useful for example when we want to display a user message for an enumeration type:

Listing 215: show Enumeration_image.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure ShowEnumeration Image is

type Months is
  (January, February, March, April,
   May, June, July, August, September,
   October, November, December);

M : constant Months := January;
begin
  Put_Line ("Month: 
    & Months' Image (M));
end ShowEnumeration Image;
```

Runtime output

Month: JANUARY

This is similar to having this code:

Listing 216: show Enumeration_image.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure ShowEnumeration Image is

type Months is
  (January, February, March, April,
   May, June, July, August, September,
   October, November, December);

M : constant Months := January;

function Months Image (M : Months)
begin
  case M is
    when January => return "JANUARY";
    when February => return "FEBRUARY";
    when March => return "MARCH";
    when April => return "APRIL";
    when May => return "MAY";
    when June => return "JUNE";
    when July => return "JULY";
    when August => return "AUGUST";
    when September => return "SEPTEMBER";
    when October => return "OCTOBER";
  end case;
end Months Image;
```
when November => return "NOVEMBER";
when December => return "DECEMBER";
end case;
end Months_Image;

begin
  Put_Line ("Month: 
    & Months_Image (M));
end ShowEnumeration_Image;

Here, the Months_Image function associates a string with each month of the Months enumeration. As expected, the compiler needs to store the strings used in the Months_Image function when compiling this code. Similarly, the compiler needs to store strings for the Months enumeration for the Image attribute.

Sometimes, we don't need to call the Image attribute for a type. In this case, we could save some storage by eliminating the strings associated with the type. Here, we can use the Discard_Names aspect to request the compiler to reduce — as much as possible — the amount of storage used for storing names for this type. Let's see an example:

Listing 217: show_discard_names.adb

```
procedure Show_Discard_Names is
  pragma Warnings (Off, "is not referenced");

  type Months is
    (January, February, March, April,
     May, June, July, August, September,
     October, November, December)
  with Discard_Names;

  M : constant Months := January;
begin
  null;
end Show_Discard_Names;
```

In this example, the compiler attempts to not store strings associated with the Months type during compilation.

Note that the Discard_Names aspect is available for enumerations, exceptions, and tagged types.

In the GNAT toolchain

If we add this statement to the Show_Discard_Names procedure above:
we see that the application displays "0" instead of "JANUARY". This is because GNAT doesn't store the strings associated with the Months type when we use the Discard_Names aspect for the Months type. (Therefore, the Months' Image attribute doesn't have that information.) Instead, the compiler uses the integer value of the enumeration, so that Months' Image returns the corresponding string for this integer value.

In the Ada Reference Manual

• Aspect Discard_Names

25.3 Records

25.3.1 Default Initialization

As mentioned in the Introduction to Ada (page 65) course, record components can have default initial values. Also, we've seen that other kinds of types can have default values (page 334).

In the Ada Reference Manual, we refer to these default initial values as "default expressions of record components." The term default expression indicates that we can use any kind of expression for the default initialization of record components — which includes subprogram calls for example:

Listing 218: show_default_initialization.ads

```ada
package Show_Default_Initialization is

   function Init return Integer is
      (42);

   type Rec is record
      A : Integer := Init;
   end record;

end Show_Default_Initialization;
```

In this example, the A component is initialized by default by a call to the Init procedure.

In the Ada Reference Manual

• 3.8 Record Types

---


Dependencies

Default expressions cannot depend on other components. For example, if we have two components A and B, we cannot initialize B based on the value that A has:

Listing 219: show_default_initialization_dependency.ads

```ada
package Show_Default_Initialization_Dependency is

  function Init return Integer is (42);

  type Rec is record
    A : Integer := Init;
    B : Integer := Rec.A; -- Illegal!
  end record;

end Show_Default_Initialization_Dependency;
```

Build output

```
show_default_initialization_dependency.ads:8:25: error: component "Rec.A" cannot
    be used before end of record declaration
```

In this example, we cannot initialize the B component based on the value of the A component. (In fact, the syntax Rec.A as a way to refer to the A component is only allowed in predicates, not in the record component declaration.)

Initialization Order

The default initialization of record components is performed in arbitrary order. In fact, the order is decided by the compiler, so we don’t have control over it.

Let’s see an example:

Listing 220: simple_recs.ads

```ada
package Simple_Recs is

  function Init ($ : String; I : Integer) return Integer;

  type Rec is record
    A : Integer := Init (“A”, 1);
    B : Integer := Init (“B”, 2);
  end record;

end Simple_Recs;
```
Listing 221: simple_recs.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Simple_Recs is

   function Init (S : String;  
                   I : Integer) 
       return Integer is 
    begin 
       Put_Line (S & ": " & I'Image); 
       return I; 
    end Init;

end Simple_Recs;
```

Listing 222: show_initialization_order.adb

```ada
with Simple_Recs; use Simple_Recs;

procedure Show_Initialization_Order is 
   R : Rec;
   begin 
   null; 
   end Show_Initialization_Order;
```

**Code block metadata**

- Initialization_Order
- MD5: e3ab92ea9b2a99815cea8c2ea11cbbfb

**Runtime output**

A: 1
B: 2

When running this code example, you might see this:

A: 1
B: 2

However, the compiler is allowed to rearrange the operations, so this output is possible as well:

B: 2
A: 1

Therefore, we must write the default expression of each individual record components in such a way that the resulting initialization value is always correct, independently of the order that those expressions are evaluated.
Evaluation

According to the Annotated Ada Reference Manual, the "default expression of a record component is only evaluated upon the creation of a default-initialized object of the record type." This means that the default expression is by itself not evaluated when we declare the record type, but when we create an object of this type. It follows from this rule that the default is only evaluated when necessary, i.e., when an explicit initial value is not specified in the object declaration.

Let's see an example:

Listing 223: show_initialization_order.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Simple_Recs; use Simple_Recs;

procedure Show_Initialization_Order is
begin
  Put_Line ("Some processing first...");
  Put_Line ("Now, let's declare an object 
  & "of the record type Rec...");

  declare
    R : Rec;
  begin
    Put_Line ("An object of Rec type has 
    & "just been created.");
  end;
end Show_Initialization_Order;
```

Here, we only see the information displayed by the Init function — which is called to initialize the A and B components of the R record — during the object creation. In other words, the default expressions Init ("A", 1) and Init ("B", 2) are not evaluated when we declare the R type, but when we create an object of this type.

In the Ada Reference Manual

- 3.8 Record Types63

---

Defaults and object declaration

Note: This subsection was originally written by Robert A. Duff and published as Gem #12: Limited Types in Ada 2005\textsuperscript{64}.

Consider the following type declaration:

\begin{verbatim}
package Type_Defaults is
  type Color(Enum) is (Red, Blue, Green);
  type T is private;
  private
    type T is
      record
        Color : Color(Enum) := Red;
        Is_Gnarly : Boolean := False;
        Count : Natural;
      end record;
    procedure Do_Something;
end Type_Defaults;
\end{verbatim}

If we want to say, "make Count equal 100, but initialize Color and Is_Gnarly to their defaults", we can do this:

\begin{verbatim}
package body Type_Defaults is
  constant T := (Color => <>, Is_Gnarly => <>, Count => 100);
  procedure Do_Something is null;
end Type_Defaults;
\end{verbatim}

Historically
Prior to Ada 2005, the following style was common:

\textsuperscript{64} https://www.adacore.com/gems/ada-gem-12
package body Type_Defaults is

Object_100 : constant T :=
  (Color => Red,
   Is_Gnarly => False,
   Count    => 100);

procedure Do_Something is null;
end Type_Defaults;

Code block metadata
MD5: c1ddfae75d7f0c691356027903a6d144

Here, we only wanted Object_100 to be a default-initialized T, with Count equal to 100. It's a little bit annoying that we had to write the default values Red and False twice. What if we change our mind about Red, and forget to change it in all the relevant places? Since Ada 2005, the <> notation comes to the rescue, as we've just seen.

On the other hand, if we want to say, "make Count equal 100, but initialize all other components, including the ones we might add next week, to their defaults", we can do this:

package body Type_Defaults is

Object_100 : constant T := (Count => 100,
                          others => <>);

procedure Do_Something is null;
end Type_Defaults;

Code block metadata
MD5: 93f5d71ae80ff0ebad54f2569539f536

Note that if we add a component Glorp : Integer; to type T, then the others case leaves Glorp undefined just as this code would do:

package body Type_Defaults is

procedure Do_Something is
  Object_100 : T;
begin
  Object_100.Count := 100;
end Do_Something;
end Type_Defaults;

Code block metadata

25.3. Records
Therefore, you should be careful and think twice before using others.

**Advanced Usages**

In addition to expressions such as subprogram calls, we can use per-object expressions (page 432) for the default value of a record component. (We discuss this topic later on in more details.)

For example:

```
package Rec_Per_Object_Expressions is

  type T (D : Positive) is private;

private

  type T (D : Positive) is record
    V : Natural := D - 1;
    -- ^^^^^
    -- Per-object expression
  end record;

end Rec_Per_Object_Expressions;
```

In this example, component V is initialized by default with the per-object expression D - 1, where D refers to the discriminant D.

**25.3.2 Mutually dependent types**

In this section, we discuss how to use incomplete types (page 305) to declare mutually dependent types. Let's start with this example:

```
package Mutually_Dependent is

  type T1 is record
    B : T2;
  end record;

  type T2 is record
    A : T1;
  end record;

end Mutually_Dependent;
```
When you try to compile this example, you get a compilation error. The first problem with this code is that, in the declaration of the T1 record, the compiler doesn't know anything about T2. We could solve this by declaring an incomplete type (type T2;) before the declaration of T1. This, however, doesn't solve all the problems in the code: the compiler still doesn't know the size of T2, so we cannot create a component of this type. We could, instead, declare an access type and use it here. By doing this, even though the compiler doesn't know the size of T2, it knows the size of an access type designating T2, so the record component can be of such an access type.

To summarize, in order to solve the compilation error above, we need to:
- use at least one incomplete type;
- declare at least one component as an access to an object.

For example, we could declare an incomplete type T2 and then declare the component B of the T1 record as an access to T2. This is the corrected version:

```
package Mutually_Dependent is

  type T2;
  type T2_Access is access T2;

  type T1 is record
    B : T2_Access;
  end record;

  type T2 is record
    A : T1;
  end record;

end Mutually_Dependent;
```

We could strive for consistency and declare two incomplete types and two accesses, but this isn't strictly necessary in this case. Here's the adapted code:

```
package Mutually_Dependent is

  type T1;
  type T1_Access is access T1;

  type T2;
  type T2_Access is access T2;

end Mutually_Dependent;
```
Learning Ada

(continued from previous page)

```ada
type T1 is record
  B : T2_Access;
end record;

type T2 is record
  A : T1_Access;
end record;
end Mutually_Dependent;
```

Later on, we'll see that these code examples can be written using anonymous access types (page 877).

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations

### 25.3.3 Null records

A null record is a record that doesn't have any components. Consequently, it cannot store any information. When declaring a null record, we simply write `null` instead of declaring actual components, as we usually do for records. For example:

```ada
package Null_Recs is
  type Null_Record is record
    null;
  end record;
end Null_Recs;
```

Note that the syntax can be simplified to `is null record`, which is much more common than the previous form:

```ada
package Null_Recs is
  type Null_Record is null record;
end Null_Recs;
```

---

65 http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html
Although a null record doesn’t have components, we can still specify subprograms for it. For example, we could specify an addition operation for it:

Listing 235: null_recs.ads

```ada
package Null_Recs is

  type Null_Record is null record;

  function "+" (A, B : Null_Record) return Null_Record;

end Null_Recs;
```

Listing 236: null_recs.adb

```ada
package body Null_Recs is

  function "+" (A, B : Null_Record) return Null_Record
  is
    pragma Unreferenced (A, B);
  begin
    return (null record);
  end "+";

end Null_Recs;
```

Listing 237: show_null_rec.adb

```ada
with Null_Recs; use Null_Recs;

procedure Show_Null_Rec is
  A, B : Null_Record;
begin
  B := A + A;
  A := A + B;
end Show_Null_Rec;
```

In the Ada Reference Manual

- 4.3.1 Record Aggregates

---

[^66]: [http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html](http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html)
Simple Prototyping

A null record doesn't provide much functionality on its own, as we're not storing any information in it. However, it's far from being useless. For example, we can make use of null records to design an API, which we can then use in an application without having to implement the actual functionality of the API. This allows us to design a prototype without having to think about all the implementation details of the API in the first stage.

Consider this example:

Listing 238: devices.ads

```ada
package Devices is

  type Device is private;

  function Create (Active : Boolean) return Device;

  procedure Reset (D : out Device) is null;

  procedure Process (D : in out Device) is null;

  procedure Activate (D : in out Device) is null;

  procedure Deactivate (D : in out Device) is null;

private

  type Device is null record;

  function Create (Active : Boolean) return Device is
    (null record);

end Devices;
```

Listing 239: show_device.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Devices; use Devices;

procedure Show_Device is
  A : Device;
begin
  Put_Line ("Creating device...");
  A := Create (Active => True);

  Put_Line ("Processing on device...");
  Process (A);

  Put_Line ("Deactivating device...");
  Deactivate (A);

  Put_Line ("Activating device...");
  Activate (A);
```

(continues on next page)
Put_Line ("Resetting device...");
Reset (A);
end Show_Device;

**Code block metadata**

MD5: 7d2f4e20ac33607f7081381b307a564a

**Runtime output**

Creating device...
Processing on device...
Deactivating device...
Activating device...
Resetting device...

In the Devices package, we're declaring the Device type and its primitive subprograms: Create, Reset, Process, Activate and Deactivate. This is the API that we use in our prototype. Note that, although the Device type is declared as a private type, it's still defined as a null record in the full view.

In this example, the Create function, implemented as an expression function in the private part, simply returns a null record. As expected, this null record returned by Create matches the definition of the Device type.

All procedures associated with the Device type are implemented as null procedures, which means they don't actually have an implementation nor have any effect. We'll discuss this topic later on in the course (page 653).

In the Show_Device procedure — which is an application that implements our prototype —, we declare an object of Device type and call all subprograms associated with that type.

**Extending the prototype**

Because we're either using expression functions or null procedures in the specification of the Devices package, we don't have a package body for it (as there's nothing to be implemented). We could, however, move those user messages from the Show_Devices procedure to a dummy implementation of the Devices package. This is the adapted code:

```ada
package Devices is

    type Device is null record;

    function Create (Active : Boolean)
        return Device;

    procedure Reset (D : out Device);
    procedure Process (D : in out Device);
    procedure Activate (D : in out Device);
    procedure Deactivate (D : in out Device);

end Devices;
```

25.3. Records 423
with Ada.Text_IO; use Ada.Text_IO;

package body Devices is

  function Create (Active : Boolean) return Device is
    pragma Unreferenced (Active);
    begin
      Put_Line ("Creating device...");
      return (null record);
    end Create;

  procedure Reset (D : out Device) is
    pragma Unreferenced (D);
    begin
      Put_Line ("Processing on device...");
      end Reset;

  procedure Process (D : in out Device) is
    pragma Unreferenced (D);
    begin
      Put_Line ("Deactivating device...");
      end Process;

  procedure Activate (D : in out Device) is
    pragma Unreferenced (D);
    begin
      Put_Line ("Activating device...");
      end Activate;

  procedure Deactivate (D : in out Device) is
    pragma Unreferenced (D);
    begin
      Put_Line ("Resetting device...");
      end Deactivate;

end Devices;

with Devices; use Devices;

procedure Show_Device is
  A : Device;
  begin
    A := Create (Active => True);
    Process (A);
    Deactivate (A);
    Activate (A);
    Reset (A);
  end Show_Device;
As we changed the specification of the Devices package to not use null procedures, we now need a corresponding package body for it. In this package body, we implement the operations on the Device type, which actually just display a user message indicating which operation is being called.

Let's focus on this updated version of the Show_Device procedure. Now that we've removed all those calls to Put_Line from this procedure and just have the calls to operations associated with the Device type, it becomes more apparent that, even though Device is just a null record, we can design an application with a sequence of various commands operating on it. Also, when we just read the source-code of the Show_Device procedure, there's no clear indication that the Device type doesn't actually hold any information.

More complex applications

As we've just seen, we can use null records like any other type and create complex prototypes with them. We could, for instance, design an application that makes use of many null records, or even have types that depend on or derive from null records. Let's see a simple example:

```
package Many_Devices is

  type Device is null record;
  type Device_Config is null record;

  function Create (Config : Device_Config) return Device is
    (null record);
  type Derived_Device is new Device;

  procedure Process (D : Derived_Device) is null;

end Many_Devices;
```

```
with Many_Devices; use Many_Devices;

procedure Show_Derived_Device is
  A : Device;
  B : Derived_Device;
  C : Device_Config;

begin
  A := Create (Config => C);
  B := Create (Config => C);
```

(continues on next page)
In this example, the Create function has a null record parameter (of Device_Config type) and returns a null record (of Device type). Also, we derive the Derived_Device type from the Device type. Consequently, Derived_Device is also a null record (since it's derived from a null record). In the Show_Derived_Device procedure, we declare objects of those types (A, B and C) and call primitive subprograms to operate on them.

This example shows that, even though the types we've declared are just null records, they can still be used to represent dependencies in our application.

Implementing the API

Let's focus again on the previous example. After we have an initial prototype, we can start implementing some of the functionality needed for the Device type. For example, we can store information about the current activation state in the record:

```ada
package Devices is
  type Device is private;
  function Create (Active : Boolean)
    return Device;
  procedure Reset (D : out Device);
  procedure Process (D : in out Device);
  procedure Activate (D : in out Device);
  procedure Deactivate (D : in out Device);
private
  type Device is record
    Active : Boolean;
  end record;
end Devices;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Devices is
  function Create (Active : Boolean)
    return Device
  is
    pragma Unreferenced (Active);
```

(continues on next page)
begin
  Put_Line ("Creating device...");
  return (Active => Active);
end Create;

procedure Reset (D : out Device)
is
  pragma Unreferenced (D);
begin
  Put_Line ("Processing on device...");
end Reset;

procedure Process (D : in out Device)
is
  pragma Unreferenced (D);
begin
  Put_Line ("Deactivating device...");
end Process;

procedure Activate (D : in out Device)
is
begin
  Put_Line ("Activating device...");
  D.Active := True;
end Activate;

procedure Deactivate (D : in out Device)
is
begin
  Put_Line ("Resetting device...");
  D.Active := False;
end Deactivate;

end Devices;

Listing 247: show_device.adb

with Ada.Text_IO; use Ada.Text_IO;
with Devices; use Devices;

procedure Show_Device is
  A : Device;
begin
  A := Create (Active => True);
  Process (A);
  Deactivate (A);
  Activate (A);
  Reset (A);
end Show_Device;

Code block metadata
MD5: 348ce0c110b47a6b6fd1c9fe73ef0558

Build output
devices.adb:11:25: warning: pragma Unreferenced given for "Active" [enabled by _default]

Runtime output

25.3. Records
Learning Ada

Creating device...
Deactivating device...
Resetting device...
Activating device...
Processing on device...

Now, the Device record contains an Active component, which is used in the updated versions of Create, Activate and Deactivate.

Note that we haven't done any change to the implementation of the Show_Device procedure: It's still the same application as before. As we've been hinting in the beginning, using null records makes it easy for us to first create a prototype — as we did in the Show_Device procedure — and postpone the API implementation to a later phase of the project.

Tagged null records

A null record may be tagged, as we can see in this example:

Listing 248: null_recs.ads

```ada
package Null_Recs is
  type Tagged_Null_Record is
tagged null record;
  type Abstract_Tagged_Null_Record is
abstract tagged null record;
end Null_Recs;
```

As we see in this example, a type can be tagged, or even abstract tagged. We discuss abstract types later on in the course.

As expected, in addition to deriving from tagged types, we can also extend them. For example:

Listing 249: devices.ads

```ada
package Devices is
  type Device is private;
  function Create (Active : Boolean) return Device;
  type Derived_Device is private;
  private
    type Device is tagged null record;
    function Create (Active : Boolean) return Device is (null record);
end Devices;
```

(continues on next page)
type Derived_Device is new Device with record
Active : Boolean;
end record;

function Create (Active : Boolean)
return Derived_Device is
(Active => Active);
end Devices;

In this example, we derive Derived_Device from the Device type and extend it with the
Active component. (Because we have a type extension, we also need to override the
Create function.)

Since we're now introducing elements from object-oriented programming, we could con-
sider using interfaces instead of null records. We'll discuss this topic later on in the course.

25.3.4 Per-Object Expressions

In record type declarations, we might want to define a component that makes use of a name
that refers to a discriminant of the record type, or to the record type itself. An expression
where we use such a name is called a per-object expression.

The term "per-object" comes from the fact that, in the component definition, we're referring
to a piece of information that will be known just when creating an object of that type. For
example, if the per-object expression refers to a discriminant of a type T, the actual value of
that discriminant will only be specified when we declare an object of type T. Therefore, the
component definition is specific for that individual object — but not necessarily for other
objects of the same type, as we might use different values for the discriminant.

The constraint that contains a per-object expression is called a per-object constraint. The
actual constraint of that component isn't completely known when we declare the record
type, but only later on when an object of that type is created. (Note that the syntax of a
constraint includes the parentheses or the keyword range.)

In addition to referring to discriminants, per-object expressions can also refer to the record
type itself, as we'll see later.

Let's start with a simple record declaration:

```ada
package Rec_Per_Object_Expressions is

    type Stack (S : Positive) is private;

private

    type Integer_Array is
        array (Positive range <>) of Integer;

    type Stack (S : Positive) is record
        Arr : Integer_Array (1 .. S);
        -- ^^^^^^^
        --

end Rec_Per_Object_Expressions;
```

(continues on next page)
In this example, we see the Stack record type with a discriminant S. In the declaration of the Arr component of that type, S is a per-object expression, as it refers to the S discriminant. Also, (1 .. S) is a per-object constraint.

Let’s look at another example using anonymous access types (page 853):

Listing 251: rec_per_object_expressions.ads

```ada
package Rec_Per_Object_Expressions is

   type T is private;
   type T_Processor (Selected_T : access T) is private;

private

   type T is null record;
   type T_Container (Selected_T : access T) is null record;
   type T_Processor (Selected_T : access T) is
      record
         E : T_Container (Selected_T);
         --
         -- Selected_T
         -- ^^^^^^^^^^^^  -- Per-object expression
         --
         -- (Selected_T)
         -- ^^^^^^^^^^^^  -- Per-object constraint
      end record;
end Rec_Per_Object_Expressions;
```

Code block metadata

MD5: e4012454ea886fd429d82159b8d344b7

MD5: 8b404688be1e1037773c28a6977785836
Let's focus on the T_Processor type from this example. The Selected_T discriminant is being used in the definition of the E component. The per-object constraint is (Selected_T).

Finally, per-object expressions can also refer to the record type we're declaring. For example:

Listing 252: rec_per_object_expressions.ads

```ada
package Rec_Per_Object_Expressions is
  type T is limited private;

private
  type T_Processor (Selected_T : access T) is null record;
  type T is limited record
    E : T_Processor (T'Access);
    -- T'Access
    -- ^^^^^^^^  Per-object expression
    -- (T'Access)
    -- ^^^^^^^^^^ Per-object constraint
  end record;

end Rec_Per_Object_Expressions;
```

In this example, when we write T'Access within the declaration of the T record type, the actual value for the Access attribute will be known when an object of T type is created. In that sense, T'Access is a per-object expression — (T'Access) is the corresponding per-object constraint.

Note that T'Access is referring to the type within a type definition. This is generally treated as a reference to the object being created, the so-called current instance.

Relevant topics

- 3.8 Record Types

---

Default value

We can also use per-object expressions to calculate the default value of a record component:

Listing 253: rec_per_object_expressions.ads

```ada
package Rec_Per_Object_Expressions is

  type T (D : Positive) is private;

private

  type T (D : Positive) is record
    V : Natural := D - 1;  -- Per-object expression
    S : Natural := D'Size;  -- Per-object expression
  end record;

end Rec_Per_Object_Expressions;
```

Here, we calculate the default value of V using the per-object expression D - 1, and the
default of value of S using the per-object D'Size.

The default expression for a component of a discriminated record can be an arbitrary per-
object expression. (This contrasts with important restrictions (page 433) that exist for per-
object constraints, as we discuss later on.) Such expressions might include function calls
or uses of any defined operator. For this reason, the following code example is accepted by
the compiler:

Listing 254: rec_per_object_expressions.ads

```ada
package Rec_Per_Object_Expressions is

  type Stack (S : Positive) is private;

private

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Stack (S : Positive) is record
    Arr : Integer_Array (1 .. S);
    Top : Natural := 0;
    Overflow Warning : Positive := S * 9 / 10;  -- using computation for
    -- the default expression.
  end record
```

(continues on next page)
with
  Dynamic_Predicate =>
    Overflow_Warning in
    (S + 1) / 2 .. S - 1;
  --
  -- (S + 1) / 2
  -- ^^^^^^^^^
  -- Per-object expression
  -- using computation.
  --
  -- S - 1
  -- ^^^^^
  -- Per-object expression
  -- using computation.
end Rec_Per_Object_Expressions;

Code block metadata
MD5: 6783568fd3e76a85ca7c1cc65ba023c5

In this example, we can identify multiple per-object expressions that use a computation: $S \times \frac{9}{10}$, $(S + 1) / 2$, and $S - 1$.

Restrictions

There are some important restrictions on per-object constraints:

# Per-object range constraints such as $1 .. T'\text{Size}$ are not allowed.
• For example, the following code example doesn't compile:

Listing 255: rec_per_object_expressions.ads

```ada
package Rec_Per_Object_Expressions is

  type Bit_Field is
    array (Positive range <>) of Boolean
    with Pack;

  type T is record
    Arr : Bit_Field (1 .. T'\text{Size});
    -- ^^^^^^^
    -- ERROR: per-object range constraint
    -- using the Size attribute
    -- is illegal.
  end record;

end Rec_Per_Object_Expressions;
```

Code block metadata
MD5: c2ac9588c1d1adac8c584a0e36a8c542

Build output

25.3. Records 433
1. Within a per-object index constraint or discriminant constraint, each per-object expression must be the name of a discriminant directly, without any further computation.
   - Therefore, we're allowed to write \((1 .. S)\) — as we've seen in a previous example. However, writing \((1 .. S - 1)\) would be illegal.
   - For example, the following adaptation to the previous code example doesn't compile:

```ada
package Rec_Per_Object_Expressions is
  type Stack (S : Positive) is private;
private
  type Integer_Array is array (Natural range <>) of Integer;
  type Stack (S : Positive) is record
    Arr : Integer_Array (0 .. S - 1);
    -- ERROR: computation in per-object expression is illegal.
    Top : Integer := -1;
  end record;
end Rec_Per_Object_Expressions;
```

2. We can only use access attributes (\(T'\text{Access}\) and \(T'\text{Unchecked_Access}\)) in per-object constraints.
25.4 Aggregates

25.4.1 Container Aggregates

Note: This feature was introduced in Ada 2022.

A container aggregate is a list of elements — such as \([1, 2, 3]\) — that we use to initialize or assign to a container. For example:

Listing 257: show_container_aggregate.adb

```ada
pragma Ada_2022;
with Ada.Containers.Vectors;

procedure Show_Container_Aggregate is

package Float_Vec is new Ada.Containers.Vectors (Positive, Float);

V : constant Float_Vec.Vector := [1.0, 2.0, 3.0];
pragma Unreferenced (V);

begin
null;
end Show_Container_Aggregate;
```

In this example, \([1.0, 2.0, 3.0]\) is a container aggregate that we use to initialize a vector V.

We can specify container aggregates in three forms:

- as a null container aggregate, which indicates a container without any elements and is represented by the \([\ ]\) syntax;
- as a positional container aggregate, where the elements are simply listed in a sequence (such as \([1, 2]\));
- as a named container aggregate, where a key is indicated for each element of the list (such as \([1 => 10, 2 => 15]\)).

Let's look at a complete example:

Listing 258: show_container_aggregate.adb

```ada
pragma Ada_2022;
with Ada.Containers.Vectors;

procedure Show_Container_Aggregate is

package Float_Vec is new Ada.Containers.Vectors (Positive, Float);
```

(continues on next page)
9. -- Null container aggregate
11. 
12. -- Positional container aggregate
13. Pos_V : constant Float_Vec.Vector := [1.0, 2.0, 3.0];
14. 
15. -- Named container aggregate
16. Named_V : constant Float_Vec.Vector := [1 => 1.0, 2 => 2.0, 3 => 3.0];
17. 
18. pragma Unreferenced (Null_V, Pos_V, Named_V);
19. begin
20. null;
21. end Show_Container_Aggregate;

Code block metadata

MD5: 15ed6370377423044368a5d56402e940

In this example, we see the three forms of container aggregates. The difference between positional and named container aggregates is that:

- for positional container aggregates, the vector index is implied by its position;
- for named container aggregates, the index (or key) of each element is explicitly indicated.

Also, the named container aggregate in this example (Named_V) is using an index as the name (i.e. it’s an indexed aggregate). Another option is to use non-indexed aggregates, where we use actual keys — as we do in maps. For example:

Listing 259: show_named_container_aggregate.adb

pragma Ada_2022;

with Ada.Containers.Vectors;
with Ada.Containers.Indefinite_Hashed_Maps;
with Ada.Strings.Hash;

procedure Show_Named_Container_Aggregate is

package Float_Vec is new Ada.Containers.Vectors (Positive, Float);

package Float_Hashed_Maps is new Ada.Containers.Indefinite_Hashed_Maps
  (Key_Type => String, Element_Type => Float, Hash => Ada.Strings.Hash,
   Equivalent_Keys => "=");

-- Named container aggregate
-- using an index
In this example, Indexed_Named_V and Keyed_Named_V are both initialized with a named container aggregate. However:

- the container aggregate for Indexed_Named_V is an indexed aggregate, so we use an index for each element;

while

- the container aggregate for Keyed_Named_V has a key for each element.

Later on, we'll talk about the Aggregate aspect, which allows for defining custom container aggregates for any record type.

In the Ada Reference Manual

- 4.3.5 Container Aggregates

25.4.2 Record aggregates

We've already seen record aggregates in the Introduction to Ada (page 66) course, so this is just a brief overview on the topic.

As we already know, record aggregates can have positional and named component associations. For example, consider this package:

Listing 260: points.ads

```
package Points is
  type Point_3D is record
    X, Y, Z : Integer;
  end record;
```

68 http://www.ada-auth.org/standards/22rm/html/RM-4-3-5.html
procedure Display (P : Point_3D);
end Points;

Listing 261: points.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Points is
  procedure Display (P : Point_3D) is
  begin
    Put_Line ("(X => ", Integer'Image (P.X) & ", "");
    Put_Line (" Y => ", Integer'Image (P.Y) & ", "");
    Put_Line (" Z => ", Integer'Image (P.Z) & ")");
  end Display;
end Points;

Code block metadata
MD5: fd01961cf1da9b48d2a6150da30f7377

We can use positional or named record aggregates when assigning to an object P of Point_3D type:

Listing 262: show_record_aggregates.adb

with Points; use Points;

procedure Show_Record_Aggregates is
  P : Point_3D;
begin
  -- Positional component association
  P := (0, 1, 2);
  Display (P);
  -- Named component association
  P := (X => 3, Y => 4, Z => 5);
  Display (P);
end Show_Record_Aggregates;

Code block metadata
MD5: fc4cff950e31a633ab4e2ae3d21ddc7b
Runtime output

(X => 0,
Y => 1,
Z => 2)
(X => 3,
Y => 4,
Z => 5)

Also, we can have a mixture of both:

Listing 263: show_record_aggregates.adb

```ada
with Points; use Points;

procedure Show_Record_Aggregates is
  P : Point_3D;
begin
  -- Positional and named component associations
  P := (3, 4,
       Z => 5);
  Display (P);
end Show_Record_Aggregates;
```

Code block metadata

MD5: 493a2a87b4b28dfb0882ad73acf84710

Runtime output

(X => 3,
Y => 4,
Z => 5)

In this case, only the Z component has a named association, while the other components have a positional association.

Note that a positional association cannot follow a named association, so we cannot write P := (3, Y => 4, 5);, for example. Once we start using a named association for a component, we have to continue using it for the remaining components.

In addition, we can choose multiple components at once and assign the same value to them. For that, we use the | syntax:

Listing 264: show_record_aggregates.adb

```ada
with Points; use Points;

procedure Show_Record_Aggregates is
  P : Point_3D;
begin
  -- Multiple component selection
  P := (X | Y => 5,
       Z => 6);
  Display (P);
end Show_Record_Aggregates;
```

Code block metadata
Here, we assign 5 to both X and Y.

In the Ada Reference Manual

- 4.3.1 Record Aggregates

\[\text{<>}\]

We can use the \(<\text{>}\) syntax to tell the compiler to use the default value for specific components. However, if there's no default value for specific components, that component isn't initialized to a known value. For example:

Listing 265: show_record_aggregates.adb

```ada
with Points; use Points;

procedure Show_Record_Aggregates is
  P : Point_3D;
begin
  P := (0, 1, 2);
  Display (P);

  -- Specifying X component.
  P := (X => 42,
        Y => <>,
        Z => <>);
  Display (P);

  -- Specifying Y and Z components.
  P := (X => <>,
        Y => 10,
        Z => 20);
  Display (P);
end Show_Record_Aggregates;
```

69 http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html
Y => 1,
Z => 2)
(X => 42,
Y => 10,
Z => 20)

Here, as the components of Point_3D don't have a default value, those components that have <> are not initialized:

- when we write \( X \Rightarrow 42, \ Y \Rightarrow \text{<>}, \ Z \Rightarrow \text{<>} \), only \( X \) is initialized;
- when we write \( X \Rightarrow \text{<>}, \ Y \Rightarrow 10, \ Z \Rightarrow 20 \) instead, only \( X \) is uninitialized.

**For further reading...**

As we've just seen, all components that get a <> are uninitialized because the components of Point_3D don't have a default value. As no initialization is taking place for those components of the aggregate, the actual value that is assigned to the record is undefined. In other words, the resulting behavior might depend on the compiler's implementation.

When using GNAT, writing \( X \Rightarrow 42, \ Y \Rightarrow \text{<>}, \ Z \Rightarrow \text{<>} \) keeps the value of \( Y \) and \( Z \) intact, while \( X \Rightarrow \text{<>}, \ Y \Rightarrow 10, \ Z \Rightarrow 20 \) keeps the value of \( X \) intact.

If the components of Point_3D had default values, those would have been used. For example, we may change the type declaration of Point_3D and use default values for each component:

```ada
package Points is
  type Point_3D is record
    X : Integer := 10;
    Y : Integer := 20;
    Z : Integer := 30;
  end record;
  procedure Display (P : Point_3D);
end Points;
```

Then, writing <> makes use of those default values we've just specified:

```ada
with Points; use Points;

procedure Show Record Aggregates is
  P : Point_3D := (0, 0, 0);
begin
  -- Using default value for
  -- all components
  P := (X => <>,
       Y => <>,
```
Learning Ada

(continued from previous page)

10    Z => <>);
11    Display (P);
12   end Show_Record_Aggregates;

Code block metadata
MD5: e64c6fe4e4b3dbaa084d9b97b4fb971f

Runtime output
(X => 10,
Y => 20,
Z => 30)

Now, as expected, the default values of each component (10, 20 and 30) are used when we write `<>

Similarly, we can specify a default value for the type of each component. For example, let's declare a Point_Value type with a default value — using the Default_Value aspect — and use it in the Point_3D record type:

Listing 268: points.ads

package Points is

   type Point_Value is new Float
      with Default_Value => 99.9;

   type Point_3D is record
      X : Point_Value;
      Y : Point_Value;
      Z : Point_Value;
   end record;

   procedure Display (P : Point_3D);

end Points;

Listing 269: points.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Points is

   procedure Display (P : Point_3D) is
      begin
         Put_Line ("(X => 
               & Point_Value'Image (P.X)
               & ",");
         Put_Line (" Y => 
               & Point_Value'Image (P.Y)
               & ",");
         Put_Line (" Z => 
               & Point_Value'Image (P.Z)
               & ")");
      end Display;

end Points;

Code block metadata
Then, writing `<>` makes use of the default value of the Point_Value type:

```
with Points; use Points;

procedure Show_Record_Aggregates is
  P : Point_3D := (0.0, 0.0, 0.0);
begin
  -- Using default value of Point_Value
  -- for all components
  P := (X => <>,
        Y => <>,
        Z => <>);
  Display (P);
end Show_Record_Aggregates;
```

Runtime output

```
(X => 9.99000E+01,
 Y => 9.99000E+01,
 Z => 9.99000E+01)
```

In this case, the default value of the Point_Value type (99.9) is used for all components when we write `<>`.

### others

Also, we can use the `others` selector to assign a value to all components that aren't explicitly mentioned in the aggregate. For example:

```
with Points; use Points;

procedure Show_Record_Aggregates is
  P : Point_3D;
begin
  -- Specifying X component;
  -- using 42 for all
  -- other components.
  P := (X => 42,
        others => 100);
  Display (P);

  -- Specifying all components
  P := (others => 256);
  Display (P);
end Show_Record_Aggregates;
```

### Code block metadata

25.4. Aggregates
When we write \( P := (X \Rightarrow 42, \text{others} \Rightarrow 100) \), we're assigning 42 to \( X \) and 100 to all other components (\( Y \) and \( Z \) in this case). Also, when we write \( P := (\text{others} \Rightarrow 256) \), all components have the same value (256).

Note that writing a specific value in \text{others} \— such as \( (\text{others} \Rightarrow 256) \) 
— only works when all components have the same type. In this example, all components of \text{Point_3D} have the same type: \text{Integer}. If we had components with different types in the components selected by \text{others}, say \text{Integer} and \text{Float}, then \( (\text{others} \Rightarrow 256) \) would trigger a compilation error. For example, consider this package:

```ada
package Custom_Records is
  type Integer_Float is record
    A, B : Integer := 0;
    Y, Z : Float := 0.0;
  end record;
end Custom_Records;
```

If we had written an aggregate such as \((\text{others} \Rightarrow 256)\) for an object of type \text{Integer_Float}, the value (256) would be OK for components \( A \) and \( B \), but not for components \( Y \) and \( Z \):

```ada
with Custom_Records; use Custom_Records;
procedure Show_Record_Aggregates_Others is
  Dummy : Integer_Float;
begin
  -- ERROR: components selected by -- others must be of same -- type.
  Dummy := (others => 256);
end Show_Record_Aggregates_Others;
```
Build output

show_record_aggregates_others.adb:9:14: error: components in "others" choice must have same type
show_record_aggregates_others.adb:9:24: error: expected type "Standard.Float"
show_record_aggregates_others.adb:9:24: error: found type universal integer
gprbuild: *** compilation phase failed

We can fix this compilation error by making sure that others only refers to components of the same type:

Listing 274: show_record_aggregates_others.adb

```ada
code
with Custom_Records; use Custom_Records;
procedure Show_Record_Aggregates_Others is
  Dummy : Integer_Float;
begin
  -- OK: components selected by others have Integer type.
  Dummy := (Y | Z => 256.0, others => 256);
end Show_Record_Aggregates_Others;
```

Code block metadata

MD5: d01977a49e08d2c6cb6b7788581ed56f

In any case, writing (others => <>) is always accepted by the compiler because it simply selects the default value of each component, so the type of those values is unambiguous:

Listing 275: show_record_aggregates_others.adb

```ada
code
with Custom_Records; use Custom_Records;
procedure Show_Record_Aggregates_Others is
  Dummy : Integer_Float;
begin
  Dummy := (others => <>);
end Show_Record_Aggregates_Others;
```

Code block metadata

MD5: db9b72ff933436e76305887276eeafdd

This code compiles because <> uses the appropriate default value of each component.
Learning Ada

Record discriminants

When a record type has discriminants, they must appear as components of an aggregate of that type. For example, consider this package:

Listing 276: points.ads

```ada
package Points is

  type Point_Dimension is (Dim_1, Dim_2, Dim_3);

  type Point (D : Point_Dimension) is record
    case D is
    when Dim_1 =>
      X1 : Integer;
    when Dim_2 =>
      X2, Y2 : Integer;
    when Dim_3 =>
      X3, Y3, Z3 : Integer;
    end case;
  end record;

  procedure Display (P : Point);
end Points;
```

Listing 277: points.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Points is

  procedure Display (P : Point) is
  begin
    Put_Line (Point_Dimension'Image (P.D));
    case P.D is
    when Dim_1 =>
      Put_Line (" (X => ", Integer'Image (P.X1), ")");
    when Dim_2 =>
      Put_Line (" (X => ", Integer'Image (P.X2), ", Y => ", Integer'Image (P.Y2), ")");
    when Dim_3 =>
    end case;
  end Display;
end Points;
```
To write aggregates of the Point type, we have to specify the D discriminant as a component of the aggregate. The discriminant must be included in the aggregate — and must be static — because the compiler must be able to examine the aggregate to determine if it is both complete and consistent. All components must be accounted for one way or another, as usual — but, in addition, references to those components whose existence depends on the discriminant's values must be consistent with the actual discriminant value used in the aggregate. For example, for type Point, an aggregate can only reference the X3, Y3, and Z3 components when Dim_3 is specified for the discriminant D; otherwise, those three components don’t exist in that aggregate. Also, the discriminant D must be the first one if we use positional component association. For example:

```ada
with Points; use Points;

procedure Show_Rec_Aggregate_Discriminant is
  -- Positional component association
  P1 : constant Point := (Dim_1, 0);
  -- Named component association
  P2 : constant Point := (D => Dim_2,
                          X2 => 3,
                          Y2 => 4);
  -- Positional / named component association
  P3 : constant Point := (Dim_3,
                          X3 => 3,
                          Y3 => 4,
                          Z3 => 5);
begin
  Display (P1);
  Display (P2);
  Display (P3);
end Show_Rec_Aggregate_Discriminant;
```

As we see in this example, we can use any component association in the aggregate, as long as we make sure that the discriminants of the type appear as components — and are the first components in the case of positional component association.
25.4.3 Full coverage rules for Aggregates

Note: This section was originally written by Robert A. Duff and published as Gem #1: Limited Types in Ada 2005.

One interesting feature of Ada are the full coverage rules for aggregates. For example, suppose we have a record type:

Listing 279: persons.ads

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

package Persons is
  type Years is new Natural;
  type Person is record
    Name : Unbounded_String;
    Age : Years;
  end record;
end Persons;
```

We can create an object of the type using an aggregate:

Listing 280: show_aggregate_init.adb

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

with Persons; use Persons;

procedure Show_Aggregate_Init is
  X : constant Person :=
    (Name =>
      To_Unbounded_String ("John Doe"),
    Age => 25);
begin
null;
end Show_Aggregate_Init;
```

The full coverage rules say that every component of Person must be accounted for in the aggregate. If we later modify type Person by adding a component:

70 https://www.adacore.com/gems/gem-1
Learning Ada

Listing 281: persons.ads

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

package Persons is
  type Years is new Natural;

  type Person is record
    Name : Unbounded_String;
    Age  : Natural;
    Shoe_Size : Positive;
  end record;

end Persons;
```

and we forget to modify X accordingly, the compiler will remind us. Case statements also
have full coverage rules, which serve a similar purpose.

Of course, we can defeat the full coverage rules by using `others` (usually for array ag-
gregates (page 450) and case statements, but occasionally useful for record aggregates (page 437)):

Listing 282: show_aggregate_init_others.adb

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

with Persons; use Persons;

procedure Show_Aggregate_Init_Others is
  X : constant Person :=
    (Name =>
      To_Unbounded_String ("John Doe"),
    others => 25);
begin
  null;
end Show_Aggregate_Init_Others;
```

According to the Ada RM, `others` here means precisely the same thing as `Age` | `Shoe_Size`. But that's wrong: what `others` really means is "all the other components, including the ones we might add next week or next year". That means you shouldn't use `others` unless you're pretty sure it should apply to all the cases that haven't been invented yet.

Later on, we'll discuss full coverage rules for limited types (page 951).
25.4.4 Array aggregates

We've already discussed array aggregates in the *Introduction to Ada* (page 71) course. Therefore, this section just presents some details about this topic.

**In the Ada Reference Manual**
- 4.3.3 Array Aggregates

**Positional and named array aggregates**

*Note:* The array aggregate syntax using brackets (e.g.: [1, 2, 3]), which we mention in this section, was introduced in Ada 2022.

Similar to *record aggregates* (page 437), array aggregates can be positional or named. Consider this package:

``` ada
package Points is
  type Point_3D is array (1 .. 3) of Integer;
  procedure Display (P : Point_3D);
end Points;
```

``` ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
package body Points is
  procedure Display (P : Point_3D) is
    begin
      Put_Line ("(X => ", Integer'Image (P (1)) & ",")
      Put_Line (" Y => ", Integer'Image (P (2)) & ",")
      Put_Line (" Z => ", Integer'Image (P (3)) & ")
    end Display;
end Points;
```

**Code block metadata**

MD5: 7ed70d1c9685bc36900e1713619f3321

71 http://www.ada-auth.org/standards/22rm/html/RM-4-3-3.html
We can write positional or named aggregates when assigning to an object \( P \) of Point_3D type:

```
pragma Ada_2022;
with Points; use Points;
procedure Show_Array_Aggregates is
  P : Point_3D;
begin
  -- Positional component association
  P := [0, 1, 2];
  Display (P);

  -- Named component association
  P := [1 => 3, 2 => 4, 3 => 5];
  Display (P);
end Show_Array_Aggregates;
```

In this example, we assign a positional array aggregate ([0, 1, 2]) to \( P \). Then, we assign a named array aggregate ([1 => 3, 2 => 4, 3 => 5]) to \( P \). In this case, the names are the indices of the components we're assigning to.

We can also assign array aggregates to slices:

```
pragma Ada_2022;
with Points; use Points;
procedure Show_Array_Aggregates is
  P : Point_3D := [others => 0];
begin
  -- Positional component association
  P (2 .. 3) := [1, 2];
  Display (P);

  -- Named component association
  P (2 .. 3) := [1 => 3, 2 => 4];
end Show_Array_Aggregates;
```
Historically

In the first versions of Ada, we could only write array aggregates using parentheses.

Listing 287: show_array_aggregates.adb

```ada
pragma Ada_2012;
with Points; use Points;

procedure Show_Array_Aggregates is
  P : Point_3D;
begin
  -- Positional component association
  P := (0, 1, 2);
  Display (P);

  -- Named component association
  P := (1 => 3,
       2 => 4,
       3 => 5);
  Display (P);
end Show_Array_Aggregates;
```

Runtime output

(X => 0,
 Y => 1,
 Z => 2)

Note that, when using a named array aggregate, the index (name) that we use in the aggregate doesn't have to match the slice. In this example, we're assigning the component from index 1 of the aggregate to the component of index 2 of the array P (and so on).
Learning Ada

(continued from previous page)

(X => 3, Y => 4, Z => 5)

This syntax is considered obsolescent since Ada 2022: brackets ([1, 2, 3]) should be used instead.

Null array aggregate

Note: This feature was introduced in Ada 2022.

We can also write null array aggregates: []. As the name implies, this kind of array aggregate doesn't have any components.

Consider this package:

Listing 288: integer_arrays.ads

```ada
package Integer_Arrays is

  type Integer_Array is
    array (Positive range <>) of Integer;

  procedure Display (A : Integer_Array);

end Integer_Arrays;
```

Listing 289: integer_arrays.adb

```ada
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;

package body Integer_Arrays is

  procedure Display (A : Integer_Array) is
    begin
      Put_Line ("Length = "
        & A'Length'Image);

      Put_Line ("[");
      for I in A'Range loop
        Put (" 
          & I'Image
          & " => "
          & A (I)'Image);
        if I /= A'Last then
          Put_Line (","');
        else
          New_Line;
        end if;
      end loop;
      Put_Line ("]");
    end Display;

end Integer_Arrays;
```

25.4. Aggregates 453
We can initialize an object \( N \) of \( \text{Integer_Array} \) type with a null array aggregate:

```ada
pragma Ada_2022;
with Integer_Arrays; use Integer_Arrays;
procedure Show_Array_Aggregates is
  \( N \) : constant Integer_Array := []; begin
  Display (N);
end Show_Array_Aggregates;
```

### Runtime output

Length = 0

In this example, when we call the `Display` procedure, we confirm that \( N \) doesn't have any components.

\(|, <=, \text{others}\

We've seen the following syntactic elements when we were discussing *record aggregates* (page 437): |, <= and `others`. We can apply them to array aggregates as well:

```ada
pragma Ada_2022;
with Points; use Points;
procedure Show_Array_Aggregates is
  \( P \) : Point_3D; begin
  -- All components have a value of zero.
  \( P \) := [\( \text{others} \) => 0];
  Display (\( P \));
  -- Both first and second components have
  -- a value of three.
  \( P \) := [1 | 2 => 3,
              3 => 4];
  Display (\( P \));
```

(continues on next page)
The default value is used for the first component, and all other components have a value of five.

\[
P := \begin{cases} 1 & \Rightarrow \langle \rangle, \\
others & \Rightarrow 5 \end{cases}.
\]

Display (P);

end Show_Array_Aggregates;

--- The default value is used for the first component, and all other components have a value of five.

P := \begin{cases} 1 & \Rightarrow \langle \rangle, \\
others & \Rightarrow 5 \end{cases};

Display (P);

end Show_Array_Aggregates;

**Code block metadata**


MD5: 053d4f162cc676b61d8e8a720321d40f

**Runtime output**

(X => 0, Y => 0, Z => 0)

(X => 3, Y => 3, Z => 4)

(X => 1659340856, Y => 5, Z => 5)

In this example, we use the |, <> and others elements in a very similar way as we did with record aggregates. (See the comments in the code example for more details.)

Note that, as for record aggregates, the <> makes use of the default value (if it is available). We discuss this topic in more details later on (page 464).

..

We can also use the range syntax (..) with array aggregates:

**Listing 292: show_array_aggregates.adb**

```ada
pragma Ada_2022;
with Points; use Points;

procedure Show_Array_Aggregates is
  P : Point_3D;
begin
  -- All components have a value of zero.
  P := [1..3 => 0];

  Display (P);

  -- Both first and second components have a value of three.
  P := [1..2 => 3,
        3 => 4];

  Display (P);

  -- The default value is used for the first component, and all other components
```

(continues on next page)
This example is a variation of the previous one. However, in this case, we're using ranges instead of the | and others syntax.

### Missing components

All aggregate components must have an associated value. If we don't specify a value for a certain component, an exception is raised:

```ada
pragma Ada_2022;
with Points; use Points;
procedure Show_Array_Aggregates is
  P : Point_3D;
begin
  P := [1 => 4];
  -- ERROR: value of components at indices
  -- 2 and 3 are missing
  Display (P);
end Show_Array_Aggregates;
```

This example is a variation of the previous one. However, in this case, we're using ranges instead of the | and others syntax.
Runtime output

raised CONSTRAINT_ERROR : show_array_aggregates.adb:8 range check failed

We can use others to specify a value to all components that haven’t been explicitly mentioned in the aggregate:

Listing 294: show_array_aggregates.adb

```ada
pragma Ada_2022;
with Points; use Points;
procedure Show_Array_Aggregates is
  P : Point_3D;
begin
  P := [1 => 4, others => 0];
  -- OK: unspecified components have a
  -- value of zero
  Display (P);
end Show_Array_Aggregates;
```

Code block metadata

MD5: 63b60de44e7c08eeae19a6a9117818f5

Runtime output

(X => 4,
 Y => 0,
 Z => 0)

However, others can only be used when the range is known — compilation fails otherwise:

Listing 295: show_array_aggregates.adb

```ada
pragma Ada_2022;
with Integer_Arrays; use Integer_Arrays;
procedure Show_Array_Aggregates is
  N1 : Integer_Array := [others => 0];
  -- ERROR: range is unknown
  Display (N1);
  N2 : Integer_Array (1 .. 3) := [others => 0];
  -- OK: range is known
  Display (N2);
begin
end Show_Array_Aggregates;
```

Code block metadata
Iterated component association

Note: This feature was introduced in Ada 2022.

We can use an iterated component association to specify an aggregate. This is the general syntax:

```ada
-- All components have a value of zero
P := [for I in 1 .. 3 => 0];
```

Let's see a complete example:

```ada
pragma Ada_2022;
with Points; use Points;
procedure Show_Array_Aggregates is
  P : Point_3D;
begin
  -- All components have a value of zero
  P := [for I in 1 .. 3 => 0];
  Display (P);

  -- Both first and second components have
  -- a value of three
  P := [for I in 1 .. 3 =>
        (if I = 1 or I = 2
         then 3
         else 4)];
  Display (P);

  -- The first component has a value of 99
  -- and all other components have a value
  -- that corresponds to its index
  P := [1 => 99,
        for I in 2 .. 3 => I];
  Display (P);
end Show_Array_Aggregates;
```
In this example, we use iterated component associations in different ways:

1. We write a simple iteration ([for I in 1 .. 3 => 0]).
2. We use a conditional expression in the iteration: [for I in 1 .. 3 => (if I = 1 or I = 2 then 3 else 4)].
3. We use a named association for the first element, and then iterated component association for the remaining components: [1 => 99, for I in 2 .. 3 => I].

So far, we’ve used a discrete choice list (in the for I in Range form) in the iterated component association. We could use an iterator (in the for E of form) instead. For example:

```
pragma Ada_2022;
with Points; use Points;
procedure Show_Array_Aggregates is
  P : Point_3D := [for I in Point_3D'Range => I];
begin
  -- Each component is doubled
  P := [for E of P => E * 2];
  Display (P);

  -- Each component is increased
  -- by one
  P := [for E of P => E + 1];
  Display (P);
end Show_Array_Aggregates;
```
Y => 5,
Z => 7)

In this example, we use iterators in different ways:

1. We write \([\text{for } E \text{ of } P \Rightarrow E \ast 2]\) to double the value of each component.
2. We write \([\text{for } E \text{ of } P \Rightarrow E + 1]\) to increase the value of each component by one.

Of course, we could write more complex operations on \(E\) in the iterators.

**Multidimensional array aggregates**

So far, we’ve discussed one-dimensional array aggregates. We can also use the same constructs when dealing with multidimensional arrays. Consider, for example, this package:

Listing 298: matrices.ads

```ada
package Matrices is
  type Matrix is array (Positive range <>,
                        Positive range <>)
                          of Integer;
  procedure Display (M : Matrix);
end Matrices;
```

Listing 299: matrices.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
package body Matrices is
  procedure Display (M : Matrix) is
    procedure Display_Row (M : Matrix; I : Integer) is
      begin
        Put_Line (" ("; for J in M’Range (2) loop
          Put (" ",
               & J’Image
               & " => "
               & M (I, J)’Image);
          if J /= M’Last (2) then
            Put_Line (","");
          else
            New_Line;
          end if;
        end loop;
        Put (" )");
      end Display_Row;
      begin
        Put_Line ("Length (1) = ",
                  & M’Length (1)’Image);
        Put_Line ("Length (2) = ",
                  & M’Length (2)’Image);
```

(continues on next page)
& M'Length (2)'Image);

Put_Line ("(");
for I in M'Range (1) loop
  Display_Row (M, I);
  if I /= M'Last (1) then
    Put_Line (",");
  else
    New_Line;
  end if;
end loop;
Put_Line (")");
end Display;
end Matrices;

---

**Code block metadata**

MD5: 748c7c695dfe43d7d4926edf5ddd3ae

We can assign multidimensional aggregates to a matrix M using positional or named component association:

Listing 300: show_array_aggregates.adb

```
pragma Ada_2022;
with Matrices; use Matrices;

procedure Show_Array_Aggregates is
  M : Matrix (1 .. 2, 1 .. 3);
begin
  -- Positional component association
  M := [[0, 1, 2],
        [3, 4, 5]];
  Display (M);

  -- Named component association
  M := [[1 => 3,
         2 => 4,
         3 => 5],
         [1 => 6,
          2 => 7,
          3 => 8]];
  Display (M);
end Show_Array_Aggregates;
```

---

**Code block metadata**

MD5: 78e1fad3b90d4f4d0f9d45f299e5ae10

**Runtime output**

---

25.4. Aggregates
The first aggregate we use in this example is \([\begin{bmatrix} 0 & 1 \end{bmatrix}, \begin{bmatrix} 3 & 4 & 5 \end{bmatrix}]\). Here, \([0, 1, 2]\) and \([3, 4, 5]\) are subaggregates of the multidimensional aggregate. Subaggregates don't have a type themselves, but are rather just considered part of a multidimensional aggregate (which, of course, has an array type). In this sense, a subaggregate such as \([0, 1, 2]\) is different from a one-dimensional aggregate (such as \([0, 1, 2]\)), even though they are written in the same way.

### Strings in subaggregates

In the case of matrices using characters, we can use strings in the corresponding array aggregates. Consider this package:

**Listing 301: string_lists.ads**

```ada
package String_Lists is
  type String_List is array (Positive range <>, Positive range <>) of Character;
  procedure Display (SL : String_List);
end String_Lists;
```

**Listing 302: string_lists.adb**

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)
package body String_Lists is

   procedure Display (SL : String_List) is

   procedure Display_Row (SL : String_List; I : Integer) is

   begin
     Put (" (";
     for J in SL'Range (2) loop
       Put (SL (I, J));
     end loop;
     Put (" ");
   end Display_Row;

   begin
     Put_Line ("Length (1) = 
               & SL'Length (1)'Image);
     Put_Line ("Length (2) = 
               & SL'Length (2)'Image);

     Put_Line ("(");
     for I in SL'Range (1) loop
       Display_Row (SL, I);
       if I /= SL'Last (1) then
         Put_Line (" ");
       else
         New_Line;
       end if;
     end loop;
   end Display;

end String_Lists;

25.4. Aggregates
In the first assignment to SL, we have the aggregate ['ABC', 'DEF'], which uses strings as subaggregates. (Of course, we can use a named aggregate and assign characters to the individual components.)

<> and default values

As we indicated earlier, the <> syntax sets a component to its default value — if such a default value is available. If a default value isn’t defined, however, the component will remain uninitialized, so that the behavior is undefined. Let's look at more complex example to illustrate this situation. Consider this package, for example:

Listing 304: points.ads

```ada
package Points is
  subtype Point_Value is Integer;
end Points;
```

(continues on next page)
type Point_3D is record
  X, Y, Z : Point_Value;
end record;

procedure Display (P : Point_3D);

type Point_3D_Array is
  array (Positive range <>) of Point_3D;

procedure Display (PA : Point_3D_Array);

end Points;

Listing 305: points.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Points is

procedure Display (P : Point_3D)
is
begin
  Put (" (X => ",
    & Point_Value'Image (P.X)
    & ",")
New_Line;
  Put (" Y => ",
    & Point_Value'Image (P.Y)
    & ",")
New_Line;
  Put (" Z => ",
    & Point_Value'Image (P.Z)
    & ")
end Display;

procedure Display (PA : Point_3D_Array)
is
begin
  Put_Line ("(");
  for I in PA'Range (1) loop
    Put_Line (", ",
      & Integer'Image (I)
      & ":=")
    Display (PA (I));
    if I /= PA'Last (1) then
      Put_Line (", ");
    else
      New_Line;
    end if;
  end loop;
  Put_Line (")
end Display;

end Points;

Then, let's use <> for the array components:

Code block metadata

MD5: ffaf3745621a30362c6aadaec2c3cefa
Learning Ada

Listing 306: show_record_aggregates.adb

```ada
pragma Ada_2022;

with Points; use Points;

procedure Show_Record_Aggregates is
  PA : Point_3D_Array (1 .. 2);
begin
  PA := [ (X => 3,
           Y => 4,
           Z => 5),
         (X => 6,
           Y => 7,
           Z => 8) ];
  Display (PA);

  -- Array components are
  -- uninitialized.
  PA := [ 1 => <>,
          2 => <> ];
  Display (PA);
end Show_Record_Aggregates;
```

Code block metadata


MD5: 1dee9505222fe9837cd5aa3bf119ee3a

Runtime output

```
( 1 =>
   (X => 3,
    Y => 4,
    Z => 5),
  2 =>
   (X => 6,
    Y => 7,
    Z => 8))
```

Because the record components (of the Point_3D type) don't have default values, they remain uninitialized when we write `[1 => <>, 2 => <>]. (In fact, you may see garbage in the values displayed by the Display procedure.)

When a default value is specified, it is used whenever <> is specified. For example, we could use a type that has the Default_Value aspect in its specification:
Learning Ada

Listing 307: integer_arrays.ads

package Integer_Arrays is

  type Value is new Integer
    with Default_Value => 99;

  type Integer_Array is
    array (Positive range <>) of Value;

  procedure Display (A : Integer_Array);

end Integer_Arrays;

Listing 308: show_array_aggregates.adb

pragma Ada_2022;

with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Aggregates is
  N : Integer_Array (1 .. 4);
  begin
    N := [for I in N'Range => Value (I)];
    Display (N);
    N := [others => <>];
    Display (N);
  end Show_Array_Aggregates;

Code block metadata

MD5: 17641d696172b052925d5549f53b9712

Runtime output

Length =  4
(
  1 => 1,
  2 => 2,
  3 => 3,
  4 => 4
)
Length =  4
(
  1 => 99,
  2 => 99,
  3 => 99,
  4 => 99
)

When writing an aggregate for the Point_3D type, any component that has <> gets the default value of the Point type (99):

For further reading...

Similarly, we could specify the Default_Component_Value aspect (which we discussed earlier on (page 335)) in the declaration of the array type:

25.4. Aggregates
package Integer_Arrays is

  type Value is new Integer;

  type Integer_Array is
    array (Positive range <>) of Value
    with Default_Component_Value => 9999;

  procedure Display (A : Integer_Array);
end Integer_Arrays;

pragma Ada_2022;

with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Aggregates is
  N : Integer_Array (1 .. 4);
begin
  N := [for I in N'Range => Value (I)];
  Display (N);
  N := [others => <>];
  Display (N);
end Show_Array_Aggregates;

Code block metadata

MD5: c6b38711937a1a7bb92d6b4c207404e

Runtime output

Length = 4
( 1 => 1, 2 => 2, 3 => 3, 4 => 4 )
Length = 4
( 1 => 9999, 2 => 9999, 3 => 9999, 4 => 9999 )

In this case, when writing <> for a component, the value specified in the Default_Component_Value aspect is used.

Finally, we might want to use both Default_Value (which we discussed previously (page 334)) and Default_Component_Value aspects at the same time. In this case, the value specified in the Default_Component_Value aspect has higher priority:
package Integer_Arrays is

   type Value is new Integer
      with Default_Value => 99;

   type Integer_Array is
      array (Positive range <>) of Value
      with Default_Component_Value => 9999;

   procedure Display (A : Integer_Array);

end Integer_Arrays;

pragma Ada_2022;

with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Aggregates is
   N : Integer_Array (1 .. 4);
begin
   N := [for I in N’Range => Value (I)];
   Display (N);
   N := [others => <>];
   Display (N);
end Show_Array_Aggregates;

Code block metadata

MD5: c5b6d45576d59e2d3ba1634953c58b02

Runtime output

Length = 4
( 1 => 1,
 2 => 2,
 3 => 3,
 4 => 4
)
Length = 4
( 1 => 9999,
 2 => 9999,
 3 => 9999,
 4 => 9999
)

Here, 9999 is used when we specify <> for a component.
25.4.5 Extension Aggregates

Extension aggregates provide a convenient way to express an aggregate for a type that extends — adds components to — some existing type (the "ancestor"). Although mainly a matter of convenience, an extension aggregate is essential when we want to express an aggregate for an extension of a private ancestor type, that is, when we don't have compile-time visibility to the ancestor type's components.

Assignments to objects of derived types

Before we discuss extension aggregates in more detail, though, let's start with a simple use-case. Let's say we have:

- an object A of tagged type T1, and
- an object B of tagged type T2, which extends T1.

We can initialize object B by:

- copying the T1 specific information from A to B, and
- initializing the T2 specific components of B.

We can translate the description above to the following code:

```ada
A : T1;
B : T2;
begin
  T1 (B) := A;
  B.Extended_Component_1 := Some_Value;
-- [...]
```

Here, we use T1 (B) to select the ancestor view of object B, and we copy all the information from A to this part of B. Then, we initialize the remaining components of B. We'll elaborate on this kind of assignments later on.

Example: Points

To present a more concrete example, let's start with a package that defines one, two and three-dimensional point types:

Listing 313: points.ads

```ada
package Points is
  type Point_1D is tagged record
    X : Float;
  end record;

  procedure Display (P : Point_1D);

  type Point_2D is new Point_1D with record
    Y : Float;
  end record;

  type Point_3D is new Point_1D with record
    Z : Float;
  end record;

end Points;
```

http://www.ada-auth.org/standards/2Zrm/html/RM-4-3-2.html
Y : Float;
end record;

procedure Display (P : Point_2D);

type Point_3D is new Point_2D with record
Z : Float;
end record;

procedure Display (P : Point_3D);
end Points;

Listing 314: points.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Points is

    procedure Display (P : Point_1D) is
    begin
        Put_Line ("(X => " & P.X'Image & ")");
    end Display;

    procedure Display (P : Point_2D) is
    begin
        Put_Line ("(X => " & P.X'Image & ", Y => " & P.Y'Image & ")");
    end Display;

    procedure Display (P : Point_3D) is
    begin
        Put_Line ("(X => " & P.X'Image & ", Y => " & P.Y'Image & ", Z => " & P.Z'Image & ")");
    end Display;

end Points;

Code block metadata
MD5: 0acc05ae2310ab4ba038dfdb6bae0495

Let's now focus on the Show_Points procedure below, where we initialize a two-dimensional point using a one-dimensional point.

Listing 315: show_points.adb

with Points; use Points;

procedure Show_Points is
    P_1D : Point_1D;
    P_2D : Point_2D;
begin
    P_1D := (X => 0.5);
    Display (P_1D);
    Point_1D (P_2D) := P_1D;
    -- Equivalent to: "P_2D.X := P_1D.X;"

(continues on next page)
In this example, we're initializing \(P_{2D}\) using the information stored in \(P_{1D}\). By writing \(\text{Point}_1 \text{D}(P_{2D})\) on the left side of the assignment, we specify that we want to limit our focus on the \(\text{Point}_1 \text{D}\) view of the \(P_{2D}\) object. Then, we assign \(P_{1D}\) to the \(\text{Point}_1 \text{D}\) view of the \(P_{2D}\) object. This assignment initializes the X component of the \(P_{2D}\) object. The \(\text{Point}_2 \text{D}\) specific components are not changed by this assignment. (In other words, this is equivalent to just writing \(P_{2D}.X := P_{1D}.X\), as the \(\text{Point}_1 \text{D}\) type only has the X component.) Finally, in the next line, we initialize the Y component with 0.7.

**Using extension aggregates**

Note that, in the assignment to \(P_{1D}\), we use a record aggregate. Extension aggregates are similar to record aggregates, but they include the \textbf{with} keyword — for example: \((\text{Obj1 with Y} \Rightarrow 0.5)\). This allows us to assign to an object with information from another object \text{Obj1} of a parent type and, in the same expression, set the value of the Y component of the type extension.

Let's rewrite the previous Show_Points procedure using extension aggregates:

```
with Points; use Points;

procedure Show_Points is
  P_1D : Point_1D;
  P_2D : Point_2D;
begin
  P_1D := (X => 0.5);
  Display (P_1D);

  P_2D := (P_1D with Y => 0.7);
  Display (P_2D);
end Show_Points;
```

**Runtime output**

\((X => 5.00000E-01)\)
\((X => 5.00000E-01, Y => 7.00000E-01)\)
When we write \( P_{2D} := (P_{1D} \text{ with } Y \Rightarrow 0.7) \), we're initializing \( P_{2D} \) using:

- the information from the \( P_{1D} \) object — of \( \text{Point}_1D \) type, which is an ancestor of the \( \text{Point}_2D \) type —, and
- the information from the record component association list for the remaining components of the \( \text{Point}_2D \) type. (In this case, the only remaining component of the \( \text{Point}_2D \) type is \( Y \).)

We could also specify the type of the extension aggregate. For example, in the previous assignment to \( P_{2D} \), we could write \( \text{Point}_2D'(\ldots) \) to indicate that we expect the \( \text{Point}_2D \) type for the extension aggregate.

```
-- Explicitly state that the type of the
-- extension aggregate is Point_2D:
P_{2D} := \text{Point}_2D'(P_{1D} \text{ with } Y \Rightarrow 0.7);
```

Also, we don't have to use named association in extension aggregates. We could just use positional association instead. Therefore, we could simplify the assignment to \( P_{2D} \) in the previous example by just writing:

```
P_{2D} := (P_{1D} \text{ with } 0.7);
```

### More extension aggregates

We can use extension aggregates for descendants of the \( \text{Point}_2D \) type as well. For example, let's extend our previous code example by declaring an object of \( \text{Point}_3D \) type (called \( P_{3D} \)) and use extension aggregates in assignments to this object:

```
procedure Show_Points is
  P_{1D} : \text{Point}_1D;
P_{2D} : \text{Point}_2D;
P_{3D} : \text{Point}_3D;
begin
  P_{1D} := (X \Rightarrow 0.5);
  Display (P_{1D});
  P_{2D} := (P_{1D} \text{ with } Y \Rightarrow 0.7);
  Display (P_{2D});
  P_{3D} := (P_{2D} \text{ with } Z \Rightarrow 0.3);
  Display (P_{3D});
  P_{3D} := (P_{1D} \text{ with } Y \mid Z \Rightarrow 0.1);
  Display (P_{3D});
end Show_Points;
```

**Code block metadata**

- **Project:** Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
- **MD5:** 2ecc6831557c43f697bffce8496962b53

25.4. Aggregates
In the first assignment to \texttt{P\_3D} in the example above, we're initializing this object with information from \texttt{P\_2D} and specifying the value of the \texttt{Z} component. Then, in the next assignment to the \texttt{P\_3D} object, we're using an aggregate with information from \texttt{P\_1} and specifying values for the \texttt{Y} and \texttt{Z} components. (Just as a reminder, we can write \texttt{Y | Z => 0.1} to assign 0.1 to both \texttt{Y} and \texttt{Z} components.)

\textbf{with others}

Other versions of extension aggregates are possible as well. For example, we can combine keywords and write \texttt{with others} to focus on all remaining components of an extension aggregate.

\begin{Verbatim}
with Points; use Points;

procedure Show_Points is
  P\_1D : Point\_1D;
  P\_2D : Point\_2D;
  P\_3D : Point\_3D;
begin
  P\_1D := (X => 0.5);
  P\_2D := (P\_1D with Y => 0.7);
  -- Initialize P\_3D with P\_1D and set other components to 0.6.
  --
  P\_3D := (P\_1D with others => 0.6);
  Display (P\_3D);
  -- Initialize P\_3D with P\_2D, and other components with their default value.
  --
  P\_3D := (P\_2D with others => <>);
  Display (P\_3D);
end Show_Points;
\end{Verbatim}

\textbf{Code block metadata}

MD5: 0594586fc59ead106258cef8682927e9

\begin{Verbatim}
(X => 5.00000E-01, Y => 6.00000E-01, Z => 6.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01, Z => 5.93170E-39)
\end{Verbatim}

In this example, the first assignment to \texttt{P\_3D} has an aggregate with information from \texttt{P\_1D}, while the remaining components — in this case, \texttt{Y} and \texttt{Z} — are just set to 0.6.

Continuing with this example, in the next assignment to \texttt{P\_3D}, we're using information from \texttt{P\_2} in the extension aggregate. This covers the \texttt{Point\_2D} part of the \texttt{P\_3D} object — components \texttt{X} and \texttt{Y}, to be more specific. The \texttt{Point\_3D} specific components of \texttt{P\_3D} — component
Z in this case — receive their corresponding default value. In this specific case, however, we haven't specified a default value for component Z in the declaration of the Point_3D type, so we cannot rely on any specific value being assigned to that component when using others => <>.

**with null record**

We can also use extension aggregates with null records. Let's focus on the P_3D_Ext object of Point_3D_Ext type. This object is declared in the Show_Points procedure of the next code example.

```
package Points.Extensions is
  type Point_3D_Ext is new Point_3D with null record;
end Points.Extensions;
```

```
with Points; use Points;
with Points.Extensions; use Points.Extensions;

procedure Show_Points is
  P_3D : Point_3D;
  P_3D_Ext : Point_3D_Ext;
begin
  P_3D := (X => 0.0, Y => 0.5, Z => 0.4);
  P_3D_Ext := (P_3D with null record);
  Display (P_3D_Ext);
end Show_Points;
```

The P_3D_Ext object is of Point_3D_Ext type, which is declared in the Points.Extensions package and derived from the Point_3D type. Note that we're not extending Point_3D_Ext with new components, but using a null record instead in the declaration. Therefore, as the Point_3D_Ext type doesn't own any new components, we just write (P_3D with null record) to initialize the P_3D_Ext object.
Extension aggregates and descendent types

In the examples above, we've been initializing objects of descendent types by using objects of ascending types in extension aggregates. We could, however, do the opposite and initialize objects of ascending types using objects of descendent type in extension aggregates. Consider this code example:

```
Listing 321: show_points.adb

with Points; use Points;

procedure Show_Points is
  P_2D : Point_2D;
  P_3D : Point_3D;
begin
  P_3D := (X => 0.5, Y => 0.7, Z => 0.3);
  Display (P_3D);
  P_2D := (Point_1D (P_3D) with Y => 0.3);
  Display (P_2D);
end Show_Points;
```

Here, we're using `Point_1D` (P_3D) to select the `Point_1D` view of an object of `Point_3D` type. At this point, we have specified the `Point_1D` part of the aggregate, so we still have to specify the remaining components of the `Point_2D` type — the Y component, to be more specific. When we do that, we get the appropriate aggregate for the `Point_2D` type. In summary, by carefully selecting the appropriate view, we're able to initialize an object of ascending type (`Point_2D`), which contains less components, using an object of a descendent type (`Point_3D`), which contains more components.

25.4.6 Delta Aggregates

Note: This feature was introduced in Ada 2022.

Previously, we've discussed extension aggregates (page 472), which are used to assign an object `Obj_From` of a tagged type to an object `Obj_To` of a descendent type.

We may want also to assign an object `Obj_From` of to an object `Obj_To` of the same type, but change some of the components in this assignment. To do this, we use delta aggregates.
Delta Aggregates for Tagged Records

Let's reuse the Points package from a previous example:

Listing 322: points.ads

```ada
package Points is

  type Point_1D is tagged record
      X : Float;
  end record;

  type Point_2D is new Point_1D with record
      Y : Float;
  end record;

  type Point_3D is new Point_2D with record
      Z : Float;
  end record;

  procedure Display (P : Point_3D);

end Points;
```

Listing 323: points.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Points is

  procedure Display (P : Point_3D) is
  begin
    Put_Line ("(X => " & P.X'Image & ", Y => " & P.Y'Image & ", Z => " & P.Z'Image & ")");
  end Display;

end Points;
```

Listing 324: show_points.adb

```ada
pragma Ada_2022;

with Points; use Points;

procedure Show_Points is
  P1, P2, P3 : Point_3D;
begin
  P1 := (X => 0.5, Y => 0.7, Z => 0.3);
  Display (P1);

  P2 := (P1 with delta X => 1.0);
  Display (P2);

  P3 := (P1 with delta X => 0.2, Y => 0.3);
  Display (P3);
end Show_Points;
```

25.4. Aggregates
Here, we assign \( P_1 \) to \( P_2 \), but change the \( X \) component. Also, we assign \( P_1 \) to \( P_3 \), but change the \( X \) and \( Y \) components.

We can use class-wide types with delta aggregates. Consider this example:

```ada
pragma Ada_2022;
with Points; use Points;

procedure Show_Points is
  P_3D : Point_3D;
  function Reset (P_2D : Point_2D'Class) return Point_2D'Class is
    ((P_2D with delta X | Y => 0.0));
  begin
    P_3D := [X => 0.1, Y => 0.2, Z => 0.3];
    Display (P_3D);

    P_3D := Point_3D (Reset (P_3D));
    Display (P_3D);
  end Show_Points;
```

In this example, the Reset function returns an object of `Point_2D'Class` where all components of `Point_2D'Class` type are zero. We call the Reset function for the `P_3D` object of `Point_3D` type, so that only the \( Z \) component remains untouched.

Note that we use the syntax \( X \mid Y \) in the body of the Reset function and assign the same value to both components.

For further reading...

We could have implemented Reset as a procedure — in this case, without using delta aggregates:
with Points; use Points;

procedure Show_Points is
  P_3D : Point_3D;

  procedure Reset
    (P_2D : in out Point_2D'Class) is
  begin
    Point_2D (P_2D) := (others => 0.0);
  end Reset;

begin
  P_3D := (X => 0.1, Y => 0.2, Z => 0.3);
  Display (P_3D);
  Reset (P_3D);
  Display (P_3D);
end Show_Points;

Delta Aggregates for Non-Tagged Records

The examples above use tagged types. We can also use delta aggregates with non-tagged types. Let's rewrite the Points package and convert Point_3D to a non-tagged record type.

package Points is
  type Point_3D is record
    X : Float;
    Y : Float;
    Z : Float;
  end record;

  procedure Display (P : Point_3D);
end Points;

with Ada.Text_IO; use Ada.Text_IO;

package body Points is
  procedure Display (P : Point_3D) is
  begin
    Put_Line ("(X => " & P.X'Image & ", Y => " & P.Y'Image & ", Z => " & P.Z'Image & ")");
  end Display;
end Points;

25.4. Aggregates
Listing 329: show_points.adb

```ada
pragma Ada_2022;
with Points; use Points;

procedure Show_Points is
  P1, P2, P3 : Point_3D;
begind
  P1 := (X => 0.5, Y => 0.7, Z => 0.3);
  Display (P1);

  P2 := (P1 with delta X => 1.0);
  Display (P2);

  P3 := (P1 with delta X => 0.2, Y => 0.3);
  Display (P3);
end Show_Points;
```

In this example, Point_3D is a non-tagged type. Note that we haven't changed anything in the Show_Points procedure: it still works as it did with tagged types.

**Delta Aggregates for Arrays**

We can use delta aggregates for arrays. Let's change the declaration of Point_3D and use an array to represent a 3-dimensional point:

```
package Points is
  type Float_Array is array (Positive range <>) of Float;
  type Point_3D is new Float_Array (1..3);
  procedure Display (P : Point_3D);
end Points;
```

Run-time output

(X => 5.00000E-01, Y => 7.00000E-01, Z => 3.00000E-01)
(X => 1.00000E+00, Y => 7.00000E-01, Z => 3.00000E-01)
(X => 2.00000E-01, Y => 3.00000E-01, Z => 3.00000E-01)

(continues on next page)
begin
  Put ("(");
  for I in P'Range loop
    Put (I'Image & " => " & P (I)'Image);
  end loop;
  Put_Line (")");
end Display;
end Points;

Listing 332: show_points.adb

pragma Ada_2022;
with Points; use Points;

procedure Show_Points is
  P1, P2, P3 : Point_3D;
begin
  P1 := [0.5, 0.7, 0.3];
  Display (P1);

  P2 := [P1 with delta 1 => 1.0];
  Display (P2);

  P3 := [P1 with delta 1 => 0.2, 2 => 0.3];
  -- Alternatively:
  -- P3 := [P1 with delta 1 .. 2 => 0.2, 0.3];
  Display (P3);
end Show_Points;

The implementation of Show_Points in this example is very similar to the version where use a record type. In this case, we:

- assign P1 to P2, but change the first component, and
- we assign P1 to P3, but change the first and second components.
Learning Ada

Using slices

In the assignment to P3, we can either specify each component of the delta individually or use a slice: both forms are equivalent. Also, we can use slices to assign the same number to multiple components:

Listing 333: show_points.adb

```ada
pragma Ada_2022;
with Points; use Points;
procedure Show_Points is
  P1, P3 : Point_3D;
begin
  P1 := [0.5, 0.7, 0.3];
  Display (P1);
  P3 := [P1 with delta
         P3'First + 1 .. P3'Last => 0.0];
  Display (P3);
end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_Aggregates_Array
MD5: 6d1db1634c42a885f7bfce7f7eccc59

Runtime output

```
( 1 => 5.00000E-01 2 => 7.00000E-01 3 => 3.00000E-01)
( 1 => 5.00000E-01 2 => 0.00000E+00 3 => 0.00000E+00)
```

In this example, we're assigning P1 to P3, but resetting all components of the array starting by the second one.

Multiple components

We can also assign multiple components or slices:

Listing 334: float_arrays.ads

```ada
package Float_Arrays is
  type Float_Array is
    array (Positive range <>) of Float;
  procedure Display (P : Float_Array);
end Float_Arrays;
```

Listing 335: float_arrays.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Float_Arrays is
  procedure Display (P : Float_Array) is
```

(continues on next page)
begin

Put ("(");
for I in P'Range loop
Put (I'Image
& " => "
& P (I)'Image);
end loop;
Put_Line (")");
end Display;
end Float_Arrays;

Listing 336: show_multiple_delta_slices.adb

pragma Ada_2022;
with Float_Arrays; use Float_Arrays;
procedure Show_Multiple_Delta_Slices is
  P1, P2 : Float_Array (1 .. 5);
begin
  P1 := [1.0, 2.0, 3.0, 4.0, 5.0];
  Display (P1);
  P2 := [P1 with delta
        P2'First + 1 .. P2'Last - 2 => 0.0,
        P2'Last - 1 .. P2'Last => 0.2];
  Display (P2);
end Show_Multiple_Delta_Slices;

In this example, we have two arrays P1 and P2 of Float_Array type. We assign P1 to P2, but change:
- the second to the last-but-two components to 0.0, and
- the last-but-one and last components to 0.2.

In the Ada Reference Manual
- Delta Aggregates

http://www.ada-auth.org/standards/22rm/html/RM-4-3-4.html

25.4. Aggregates
25.5 Arrays

25.5.1 Unconstrained Arrays

In the *Introduction to Ada course* (page 78), we've seen that we can declare array types whose bounds are not fixed: in that case, the bounds are provided when creating objects of those types. For example:

```
package MeasurementDefs is
  type Measurements is array (Positive range <>) of Float;
  -- ^ Bounds are of type Positive, but not known at this point.
end MeasurementDefs;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with MeasurementDefs; use MeasurementDefs;
procedure ShowMeasurements is
  M : Measurements (1 .. 10);
  -- ^ Providing bounds here!
begin
  Put_Line ("First index: " & M'First'Image);
  Put_Line ("Last index: " & M'Last'Image);
end ShowMeasurements;
```

In this example, the Measurements array type from the MeasurementDefs package is unconstrained. In the ShowMeasurements procedure, we declare a constrained object (M) of this type.

The *Introduction to Ada course* (page 79) also highlights the fact that the bounds are fixed once an object is declared:

Although different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime. This allows Ada to implement unconstrained arrays efficiently; instances can be stored on the stack and do not require heap allocation as in languages like Java.
In the Show_Measurements procedure above, once we declare M, its bounds are fixed for
the whole lifetime of M. We cannot add another component to this array. In other words, M
will have 10 components for its whole lifetime.

In the Ada Reference Manual

- 3.6 Array Types

Unconstrained Arrays vs. Vectors

If you need, however, the flexibility of increasing the length of an array, you could use
vectors instead. This is how we could rewrite the previous example using vectors:

Listing 339: measurement_defs.ads

```ada
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;
package Measurement_Defs is
  package Vectors is new Ada.Containers.Vectors
    (Index_Type => Positive,
     Element_Type => Float);
  subtype Measurements is Vectors.Vector;
end Measurement_Defs;
```

Listing 340: show_measurements.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Measurement_Defs; use Measurement_Defs;

procedure Show_Measurements is
  use Measurement_Defs.Vectors;
  M : Measurements := To_Vector (10);
  -- ^ Creating 10-element
  -- vector.
begin
  Put_Line ("First index: " & M.First_Index'Image);
  Put_Line ("Last index: " & M.Last_Index'Image);
  Put_Line ("Adding element...");
  M.Append (1.0);
  Put_Line ("First index: " & M.First_Index'Image);
  Put_Line ("Last index: " & M.Last_Index'Image);
end Show_Measurements;
```

Code block metadata

In the declaration of M in this example, we’re creating a 10-element vector by calling To_Vector and specifying the element count. Later on, with the call to Append, we’re increasing the length of the M to 11 elements.

As you might expect, the flexibility of vectors comes with a price: every time we add an element that doesn't fit in the current capacity of the vector, the container has to reallocate memory in the background due to that new element. Therefore, arrays are more efficient, as the memory allocation only happens once for each object.

In the Ada Reference Manual

- 3.6 Array Types\(^7^5\)
- A.18.2 The Generic Package Containers.Vectors\(^7^6\)

### 25.5.2 Multidimensional Arrays

So far, we've discussed unidimensional arrays, since they are very common in Ada. However, Ada also supports multidimensional arrays using the same facilities as for unidimensional arrays. For example, we can use the First, Last, **Range** and Length attributes for each dimension of a multidimensional array. This section presents more details on this topic.

To create a multidimensional array, we simply separate the ranges of each dimension with a comma. The following example presents the one-dimensional array A1, the two-dimensional array A2 and the three-dimensional array A3:

```
package Multidimensional_Arrays_Decl is
  A1 : array (1 .. 10) of Float;
  A2 : array (1 .. 5, 1 .. 10) of Float;
      ^ first dimension
      ^ second dimension
  A3 : array (1 .. 2, 1 .. 5, 1 .. 10) of Float;
      ^ first dimension
      ^ second dimension
      ^ third dimension
end Multidimensional_Arrays_Decl;
```

---


The two-dimensional array \( A_2 \) has 5 components in the first dimension and 10 components in the second dimension. The three-dimensional array \( A_3 \) has 2 components in the first dimension, 5 components in the second dimension, and 10 components in the third dimension. Note that the ranges we've selected for \( A_1, A_2 \) and \( A_3 \) are completely arbitrary. You may select ranges for each dimension that are the most appropriate in the context of your application. Also, the number of dimensions is not limited to three, so you could declare higher-dimensional arrays if needed.

We can use the Length attribute to retrieve the length of each dimension. We use an integer value in parentheses to specify which dimension we're referring to. For example, if we write \( A'\text{Length} (2) \), we're referring to the length of the second dimension of a multidimensional array \( A \). Note that \( A'\text{Length} \) is equivalent to \( A'\text{Length} (1) \). The same equivalence applies to other array-related attributes such as First, Last and Range.

Let's use the Length attribute for the arrays we declared in the Multidimensional_Arrays_Decl package:

```
with Ada.Text_IO; use Ada.Text_IO;
with Multidimensional_Arrays_Decl; use Multidimensional_Arrays_Decl;

procedure Show_Multidimensional_Arrays is
begin
  Put_Line ("A1'Length: ", A1'Length'Image);
  Put_Line ("A1'Length (1): ", A1'Length (1)'Image);
  Put_Line ("A2'Length (1): ", A2'Length (1)'Image);
  Put_Line ("A2'Length (2): ", A2'Length (2)'Image);
  Put_Line ("A3'Length (1): ", A3'Length (1)'Image);
  Put_Line ("A3'Length (2): ", A3'Length (2)'Image);
  Put_Line ("A3'Length (3): ", A3'Length (3)'Image);
end Show_Multidimensional_Arrays;
```

Code block metadata

Multidimensional_Arrays
MD5: 70b9b8df7e46302b92613fa484ef71ca

Runtime output

A1'Length: 10
A1'Length (1): 10
A2'Length (1): 5
A2'Length (2): 10
A3'Length (1): 2
A3'Length (2): 5
A3'Length (3): 10
As this simple example shows, we can easily retrieve the length of each dimension. Also, as we've just mentioned, \( A_1' \text{Length} \) is equal to \( A_1' \text{Length} \) (1).

Let's consider an application where we make hourly measurements for the first 12 hours of the day, on each day of the week. We can create a two-dimensional array type called Measurements to store this data. Also, we can have three procedures for this array:

- Show_Indices, which presents the indices (days and hours) of the two-dimensional array;
- Show_Values, which presents the values stored in the array; and
- Reset, which resets each value of the array.

This is the complete code for this application:

```
package Measurement_Defs is
  type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  type Hours is range 0 .. 11;
  subtype Measurement is Float;
  type Measurements is array (Days, Hours) of Measurement;
  procedure Show_Indices (M : Measurements);
  procedure Show_Values (M : Measurements);
  procedure Reset (M : out Measurements);
end Measurement_Defs;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Measurement_Defs is
  procedure Show_Indices (M : Measurements) is
  begin
    Put_Line ("---- Indices ----");
    for D in M'Range (1) loop
      Put (D'Image & "");
      for H in M'First (2) .. M'Last (2) - 1 loop
        Put (H'Image & "");
      end loop;
      Put (M'Last (2)'Image);
    end loop;
    end Show_Indices;
  procedure Show_Values (M : Measurements) is
    package H_IO is
      new Ada.Text_IO.Integer_IO (Hours);
    package M_IO is
      new Ada.Text_IO.Float_IO (Measurement);
    end M_IO;
    package M_IO is
      new Ada.Text_IO.Float_IO (Measurement);
    end M_IO;
    ... (continues on next page)
```

(continues on next page)
new Ada.Text_IO.Float_IO (Measurement);

procedure Set_IO_Defaults is
begin
  H_IO.Default_Width := 5;
  M_IO.Default_Fore := 1;
  M_IO.Default_Aft := 2;
  M_IO.Default_Exp := 0;
end Set_IO_Defaults;

begin
  Set_IO_Defaults;

  Put_Line ("---- Values ----");
  Put (" ");
  for H in M'Range (2) loop
    H_IO.Put (H);
  end loop;
  New_Line;
  for D in M'Range (1) loop
    Put (D'Image & " ");
    for H in M'Range (2) loop
      M_IO.Put (M (D, H));
      Put (" ");
    end loop;
    New_Line;
  end loop;
  end Show_Values;
procedure Reset (M : out Measurements) is
begin
  M := (others => (others => 0.0));
end Reset;
end Measurement_Defs;

Listing 345: show_measurements.adb

with Measurement_Defs; use Measurement_Defs;

procedure Show_Measurements is
M : Measurements;
begin
  Reset (M);
  Show_Indices (M);
  Show_Values (M);
end Show_Measurements;

Code block metadata
  Multidimensional_Measurements
MD5: bcffa3913007bd9152149ad9616842b8

Runtime output
---- Indices ----
MON 0 1 2 3 4 5 6 7 8 9 10 11
TUE 0 1 2 3 4 5 6 7 8 9 10 11
We recommend that you spend some time analyzing this example. Also, we'd like to highlight the following aspects:

- We access a value from a multidimensional array by using commas to separate the index values within the parentheses. For example: \( M(D, H) \) allows us to access the value on day \( D \) and hour \( H \) from the multidimensional array \( M \).

- To loop over the multidimensional array \( M \), we write \( \text{for } D \text{ in } M'\text{Range (1) loop} \) and \( \text{for } H \text{ in } M'\text{Range (2) loop} \) for the first and second dimensions, respectively.

- To reset all values of the multidimensional array, we use an aggregate with this form: \( (\text{others} => (\text{others} => 0.0)) \).

In the Ada Reference Manual

- 3.6 Array Types\(^{77}\)

### Unconstrained Multidimensional Arrays

Previously, we've discussed unconstrained arrays for the unidimensional case. It's possible to declare unconstrained multidimensional arrays as well. For example:

```
package Multidimensional_Arrays_Decl is
  type F1 is array (Positive range <>) of Float;
  type F2 is array (Positive range <>,
                   Positive range <>) of Float;
  type F3 is array (Positive range <>,
                   Positive range <>,
                   Positive range <>) of Float;
end Multidimensional_Arrays_Decl;
```

Here, we're declaring the one-dimensional type \( F1 \), the two-dimensional type \( F2 \) and the three-dimensional type \( F3 \).

\(^{77}\) http://www.ada-auth.org/standards/22rm/html/RM-3-6.html
As is the case with unidimensional arrays, we must specify the bounds when declaring objects of unconstrained multidimensional array types:

Listing 347: show_multidimensional_arrays.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Multidimensional_Arrays_Decl; use Multidimensional_Arrays_Decl;

procedure Show_Multidimensional_Arrays is
  A1 : F1 (1 .. 2);
  A2 : F2 (1 .. 4, 10 .. 20);
  A3 : F3 (2 .. 3, 1 .. 5, 1 .. 2);
begin
  Put_Line ("A1'Length (1): " & A1'Length (1)'Image);
  Put_Line ("A2'Length (1): " & A2'Length (1)'Image);
  Put_Line ("A2'Length (2): " & A2'Length (2)'Image);
  Put_Line ("A3'Length (1): " & A3'Length (1)'Image);
  Put_Line ("A3'Length (2): " & A3'Length (2)'Image);
  Put_Line ("A3'Length (3): " & A3'Length (3)'Image);
end Show_Multidimensional_Arrays;
```

Unconstrained_Multidimensional_Arrays
MD5: 9fb007abbfe238345d80cb315bb834c9

Build output
show_multidimensional_arrays.adb:7:04: warning: variable "A1" is read but never assigned [-gnatwv]
show_multidimensional_arrays.adb:8:04: warning: variable "A2" is read but never assigned [-gnatwv]
show_multidimensional_arrays.adb:9:04: warning: variable "A3" is read but never assigned [-gnatwv]

Runtime output
A1'Length (1): 2
A2'Length (1): 4
A2'Length (2): 11
A3'Length (1): 2
A3'Length (2): 5
A3'Length (3): 2

25.5. Arrays 491
Arrays of arrays

It's important to distinguish between multidimensional arrays and arrays of arrays. Both are supported in Ada, but they're very distinct from each other. We can create an array by first specifying a one-dimensional array type $T_1$, and then specifying another one-dimensional array type $T_2$ where each component of $T_2$ is of $T_1$ type:

Listing 348: array_of_arrays_decl.ads

```ada
package Array_Of_Arrays_Decl is
  type $T_1$ is
    array (Positive range <>) of Float;
  type $T_2$ is
    array (Positive range <>) of $T_1$ (1 .. 10);
  -- ^^^^^^^
  -- bounds must be set!
end Array_Of_Arrays_Decl;
```

Note that, in the declaration of $T_2$, we must set the bounds for the $T_1$ type. This is a major difference to multidimensional arrays, which allow for unconstrained ranges in multiple dimensions.

We can rewrite the previous application for measurements using arrays of arrays. This is the adapted code:

Listing 349: measurement_defs.ads

```ada
package MeasurementDefs is
  type Days is
    (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  type Hours is range 0 .. 11;
  subtype Measurement is Float;
  type Hourly_Measurements is
    array (Hours) of Measurement;
  type Measurements is
    array (Days) of Hourly_Measurements;
  procedure Show_Indices (M: Measurements);
  procedure Show_Values (M: Measurements);
  procedure Reset (M: out Measurements);
end MeasurementDefs;
```

Listing 350: measurement_defs.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
(continues on next page)```
package body Measurement_Defs is

procedure Show_Indices (M : Measurements) is begin
  Put_Line ("---- Indices ----");
  for D in M'Range loop
    Put (D'Image & " ");
    for H in M (D)'First .. M (D)'Last - 1 loop
      Put (H'Image & " ");
    end loop;
    Put_Line (M (D)'Last'Image);
  end loop;
end Show_Indices;

procedure Show_Values (M : Measurements) is
  package H_IO is new Ada.Text_IO.Integer_IO (Hours);
  package M_IO is new Ada.Text_IO.Float_IO (Measurement);
  procedure Set_IO_Defaults is begin
    H_IO.Default_Width := 5;
    M_IO.Default_Fore := 1;
    M_IO.Default_Aft := 2;
    M_IO.Default_Exp := 0;
    Set_IO_Defaults;
  end Set_IO_Defaults;
  Put_Line ("---- Values ----");
  Put (" ");
  for H in M (M'First)'Range loop
    H_IO.Put (H);
  end loop;
  New_Line;
  for D in M'Range loop
    Put (D'Image & " ");
    for H in M (D)'Range loop
      M_IO.Put (M (D) (H));
      Put (" ");
    end loop;
    New_Line;
  end loop;
end Show_Values;

procedure Reset (M : out Measurements) is begin
  M := (others => (others => 0.0));
end Reset;
end Measurement_Defs;

25.5. Arrays 493
Listing 351: show_measurements.adb

```ada
with MeasurementDefs; use MeasurementDefs;

procedure ShowMeasurements is
  M : Measurements;
begin
  Reset(M);
  Show_Indices(M);
  Show_Values(M);
end ShowMeasurements;
```

Code block metadata

MD5: 5cb66bbb1890787b7c023406b2cafb4d

Runtime output

```
---- Indices ----
MON  0  1  2  3  4  5  6  7  8  9 10 11
TUE  0  1  2  3  4  5  6  7  8  9 10 11
WED  0  1  2  3  4  5  6  7  8  9 10 11
THU  0  1  2  3  4  5  6  7  8  9 10 11
FRI  0  1  2  3  4  5  6  7  8  9 10 11
SAT  0  1  2  3  4  5  6  7  8  9 10 11
SUN  0  1  2  3  4  5  6  7  8  9 10 11

---- Values ----
0    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
1    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
2    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
3    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
4    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
5    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
6    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
7    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
8    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
9    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
10   0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
11   0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
```

Again, we recommend that you spend some time analyzing this example and comparing it to the previous version that uses multidimensional arrays. Also, we’d like to highlight the following aspects:

- We access a value from an array of arrays by specifying the index of each array separately. For example: `M(D)(H)` allows us to access the value on day D and hour H from the array of arrays M.
- To loop over an array of arrays M, we write `for D in M'Range loop` for the first level of M and `for H in M(D)'Range loop` for the second level of M.
- Resetting all values of an array of arrays is very similar to how we do it for multidimensional arrays. In fact, we can still use an aggregate with this form: `(others => (others => 0.0))`. 
25.6 Strings

25.6.1 Wide and Wide-Wide Strings

We've seen many source-code examples so far that includes strings. In most of them, we were using the standard string type: `String`. This type is useful for the common use-case of displaying messages or dealing with information in plain English. Here, we define "plain English" as the use of the language that avoids French accents or German umlaut, for example, and doesn't make use of any characters in non-Latin alphabets.

There are two additional string types in Ada: `Wide_String`, and `Wide_Wide_String`. These types are particularly important when dealing with textual information in non-standard English, or in various other languages, non-Latin alphabets and special symbols.

These string types use different bit widths for their characters. This becomes more apparent when looking at the type definitions:

```ada
type String is
  array (Positive range <>) of Character;

type Wide_String is
  array (Positive range <>) of Wide_Character;

type Wide_Wide_String is
  array (Positive range <>) of Wide_Wide_Character;
```

The following table shows the typical bit-width of each character of the string types:

<table>
<thead>
<tr>
<th>Character Type</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>8 bits</td>
</tr>
<tr>
<td>Wide_Character</td>
<td>16 bits</td>
</tr>
<tr>
<td>Wide_Wide_Character</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

We can see that when running this example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Wide_Char_Types is begin
  Put_Line ("Character'Size: " & Integer'Image (Character'Size));
  Put_Line ("Wide_Character'Size: " & Integer'Image (Wide_Character'Size));
  Put_Line ("Wide_Wide_Character'Size: " & Integer'Image (Wide_Wide_Character'Size));
end Show_Wide_Char_Types;
```

Code block metadata

- Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide_Wide_Strings.Wide_Char_Types
- MD5: a0e9fb9e8d43e9fa707dc8c57f7562f8
Let's look at another example, this time using wide strings:

Listing 353: show_wide_string_types.adb

```ada
with Ada.Text_IO;
with Ada.Wide_Text_IO;
with Ada.Wide_Wide_Text_IO;

procedure Show_Wide_String_Types is
    package TI renames Ada.Text_IO;
    package WTI renames Ada.Wide_Text_IO;
    package WWTI renames Ada.Wide_Wide_Text_IO;

    S : constant String := "hello";
    WS : constant Wide_String := "hello";
    WWS : constant Wide_Wide_String := "hello";

begin
    TI.Put_Line ("String: " & S);
    TI.Put_Line ("Length: " & Integer'Image (S'Length));
    TI.Put_Line ("Size: " & Integer'Image (S'Size));
    TI.Put_Line ("Component_Size: " & Integer'Image (S'SComponent_Size));
    TI.Put_Line ("------------------------");

    WTI.Put_Line ("Wide string: " & WS);
    TI.Put_Line ("Length: " & Integer'Image (WS'Length));
    TI.Put_Line ("Size: " & Integer'Image (WS'Size));
    TI.Put_Line ("Component_Size: " & Integer'Image (WS'SComponent_Size));
    TI.Put_Line ("------------------------");

    WWTI.Put_Line ("Wide-wide string: " & WWS);
    TI.Put_Line ("Length: " & Integer'Image (WWS'Length));
    TI.Put_Line ("Size: " & Integer'Image (WWS'Size));
    TI.Put_Line ("Component_Size: " & Integer'Image (WWS'SComponent_Size));
    TI.Put_Line ("------------------------");

end Show_Wide_String_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Wide_ String_Types
MD5: 137816c6fd78add34287a72e45cf4fb7

Runtime output
Here, all strings (S, W5 and WWS) have the same length of 5 characters. However, the size of each character is different — thus, each string has a different overall size.

The recommendation is to use the **String** type when the textual information you're processing is in standard English. In case any kind of internationalization is needed, using **Wide_Wide_String** is probably the best choice, as it covers all possible use-cases.

**In the Ada Reference Manual**

- 3.6.3 String Types\(^{78}\)

**Text I/O**

Note that, in the previous example, we were using different versions of the Ada.Text_IO package depending on the string type we were using:

- Ada.Text_IO for objects of **String** type,
- Ada.Wide_Text_IO for objects of **Wide_String** type,
- Ada.Wide_Wide_Text_IO for objects of **Wide_Wide_String** type.

In that example, we were also using package renaming to differentiate among those packages.

Similarly, there are different versions of text I/O packages for individual types. For example, if we want to display the value of a **Long_Integer** variable based on the **Wide_Wide_String** type, we can select the Ada.Long_Integer_Wide_Wide_Text_IO package. In fact, the list of packages resulting from the combination of those types is quite long:

\(^{78}\) http://www.ada-auth.org/standards/22rm/html/RM-3-6-3.html
Learning Ada

<table>
<thead>
<tr>
<th>Scalar Type</th>
<th>Text I/O Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>• Ada.Integer_Text_IO&lt;br&gt;• Ada.Integer_Wide_Text_IO&lt;br&gt;• Ada.Integer_Wide_Wide_Text_IO</td>
</tr>
<tr>
<td>Long_Integer</td>
<td>• Ada.Long_Integer_Text_IO&lt;br&gt;• Ada.Long_Integer_Wide_Text_IO&lt;br&gt;• Ada.Long_Integer_Wide_Wide_Text_IO</td>
</tr>
<tr>
<td>Long_Long_Integer</td>
<td>• Ada.Long_Long_Integer_Text_IO&lt;br&gt;• Ada.Long_Long_Integer_Wide_Text_IO&lt;br&gt;• Ada.Long_Long_Integer_Wide_Wide_Text_IO</td>
</tr>
<tr>
<td>Float</td>
<td>• Ada.Float_Text_IO&lt;br&gt;• Ada.Float_Wide_Text_IO&lt;br&gt;• Ada.Float_Wide_Wide_Text_IO</td>
</tr>
<tr>
<td>Long_Float</td>
<td>• Ada.Long_Float_Text_IO&lt;br&gt;• Ada.Long_Float_Wide_Text_IO&lt;br&gt;• Ada.Long_Float_Wide_Wide_Text_IO</td>
</tr>
<tr>
<td>Long_Long_Float</td>
<td>• Ada.Long_Long_Float_Text_IO&lt;br&gt;• Ada.Long_Long_Float_Wide_Text_IO&lt;br&gt;• Ada.Long_Long_Float_Wide_Wide_Text_IO</td>
</tr>
</tbody>
</table>

Also, there are different versions of the generic packages Integer_IO and Float_IO:

<table>
<thead>
<tr>
<th>Scalar Type</th>
<th>Text I/O Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer types</td>
<td>• Ada.Text_IO.Integer_IO&lt;br&gt;• Ada.Wide_Text_IO.Integer_IO&lt;br&gt;• Ada.Wide_Wide_Text_IO.Integer_IO</td>
</tr>
<tr>
<td>Real types</td>
<td>• Ada.Text_IO.Float_IO&lt;br&gt;• Ada.Wide_Text_IO.Float_IO&lt;br&gt;• Ada.Wide_Wide_Text_IO.Float_IO</td>
</tr>
</tbody>
</table>

In the Ada Reference Manual

- A.10 Text Input-Output[^79]
- A.10.1 The Package Text_IO[^80]

Wide and Wide-Wide String Handling

As we’ve just seen, we have different versions of the Ada.Text_IO package. The same applies to string handling packages. As we’ve seen in the Introduction to Ada course (page 239), we can use the Ada.Strings.Fixed and Ada.Strings.Maps packages for string handling. For other formats, we have these packages:

- Ada.Strings.Wide_Fixed,
- Ada.Strings.Wide_Wide_Fixed,
- Ada.Strings.Wide_Maps,

Let’s look at this example (page 240) from the Introduction to Ada course, which we adapted for wide-wide strings:

Listing 354: show_find_words.adb

```ada
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Wide_Wide_Fixed; use Ada.Strings.Wide_Wide_Fixed;
with Ada.Strings.Wide_Wide_Maps; use Ada.Strings.Wide_Wide_Maps;
with Ada.Wide_Wide_Text_IO; use Ada.Wide_Wide_Text_IO;

procedure Show_Find_Words is
  S : constant Wide_Wide_String := "Hello" & 3 * " World";
  F : Positive;
  L : Natural;
  I : Natural := 1;
  Whitespace : constant Wide_Wide_Character_Set := To_Set (' ');
begin
  Put_Line ("String: " & S);
  Put_Line ("String length: 
    & Integer'Wide_Wide_Image (S'Length));

  while I in S'Range loop
    Find_Token
      (Source => S,
       Set => Whitespace,
       (continues on next page)
```
Learning Ada

(continued from previous page)

From => I,
Test => Outside,
First => F,
Last => L);

exit when L = 0;

Put_Line ("Found word instance at position ",
& F'Wide_Wide_Image
& ": ": S (F .. L) & ":");

I := L + 1;
end loop;
end Show_Find_Words;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Wide_Wide_
.String_Handling
MD5: 3b5a4d61e6dc5bd16e85f85580ad82ae

Runtime output

String: Hello World World World
String length: 23
Found word instance at position 1: 'Hello'
Found word instance at position 7: 'World'
Found word instance at position 13: 'World'
Found word instance at position 19: 'World'

In this example, we're using the Find_Token procedure to find the words from the phrase stored in the S constant. All the operations we're using here are similar to the ones for String type, but making use of the Wide_Wide_String type instead. (We talk about the Wide_Wide_Image attribute later on (page 514).)

In the Ada Reference Manual

- A.4.6 String-Handling Sets and Mappings
- A.4.7 Wide_String Handling
- A.4.8 Wide_Wide_String Handling

Bounded and Unbounded Wide and Wide-Wide Strings

We've seen in the Introduction to Ada course that other kinds of String types are available. For example, we can use bounded (page 244) and unbounded strings (page 246) — those correspond to the Bounded_String and Unbounded_String types.

Those kinds of string types are available for Wide_String, and Wide_Wide_String. The following table shows the available types and corresponding packages:

---

The same applies to text I/O for those strings. For the standard case, we have `Ada.Text_IO.Bounded_IO` for the `Bounded_String` type and `Ada.Text_IO.Unbounded_IO` for the `Unbounded_String` type.

For wider string types, we have:

<table>
<thead>
<tr>
<th>Type</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded_Wide_String</td>
<td>Ada.Strings.Wide_Wide_Bounded</td>
</tr>
<tr>
<td>Bounded_Wide_Wide_String</td>
<td>Ada.Strings.Wide_Wide_Unbounded</td>
</tr>
<tr>
<td>Unbounded_Wide_String</td>
<td>Ada.Strings.Wide_Wide_Unbounded</td>
</tr>
<tr>
<td>Unbounded_Wide_Wide_String</td>
<td>Ada.Strings.Wide_Wide_Unbounded</td>
</tr>
</tbody>
</table>

Let's look at a simple example:

Listing 355: `show_unbounded_wide_wide_string.adb`

```ada
with Ada.Strings.Wide_Wide_Unbounded;
use Ada.Strings.Wide_Wide_Unbounded;

with Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO;
use Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO;

procedure Show_Unbounded_Wide_Wide_String is
    S : Unbounded_Wide_Wide_String := To_Unbounded_Wide_Wide_String ("Hello");
begin
    S := S & Wide_Wide_String'(" hello");
    Put_Line ("Unbounded wide-wide string: " & S);
end Show_Unbounded_Wide_Wide_String;
```

In this example, we're declaring a variable `S` and initializing it with the word "Hello." Then, we're concatenating it with " hello" and displaying it. All the operations we're using here are similar to the ones for `Unbounded_String` type, but they've been adapted for the `Unbounded_Wide_Wide_String` type.

### In the Ada Reference Manual

- A.4.7 Wide_String Handling\(^{87}\)


25.6. Strings 501
Learning Ada

- A.4.8 Wide_Wide_String Handling
- A.11 Wide Text Input-Output and Wide Wide Text Input-Output

25.6.2 String Encoding

Unicode is one of the most widespread standards for encoding writing systems other than the Latin alphabet. It defines a format called Unicode Transformation Format (UTF) in various versions, which vary according to the underlying precision, support for backwards-compatibility and other requirements.

In the Ada Reference Manual
- A.4.11StringEncoding

UTF-8 encoding and decoding

A common UTF format is UTF-8, which encodes strings using up to four (8-bit) bytes and is backwards-compatible with the ASCII format. While encoding of ASCII characters requires only one byte, Chinese characters require three bytes, for example.

In Ada applications, UTF-8 strings are indicated by using the UTF_8_String from the Ada.Strings.UTF_Encoding package. In order to encode from and to UTF-8 strings, we can use the Encode and Decode functions. Those functions are specified in the child packages of the Ada.Strings.UTF_Encoding package. We select the appropriate child package depending on the string type we're using, as you can see in the following table:

<table>
<thead>
<tr>
<th>Child Package of Ada.Strings.UTF_Encoding</th>
<th>Convert from / to</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Strings</td>
<td>String type</td>
</tr>
<tr>
<td>.Wide_Words</td>
<td>Wide_String type</td>
</tr>
<tr>
<td>.Wide_Wide_Words</td>
<td>Wide_Wide_String type</td>
</tr>
</tbody>
</table>

Let's look at an example:

Listing 356: show_ww_utf_string.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.UTF_Encoding;
use Ada.Strings.UTF_Encoding;
with Ada.Strings.UTF_Encoding.Wide_Wide_Words;
use Ada.Strings.UTF_Encoding.Wide_Wide_Words;
with Ada.Strings.Wide_Wide_Unbounded;
use Ada.Strings.Wide_Wide_Unbounded;
procedure Show_WW_UTF_String is
  function To_UWWS
```

(continues on next page)
Learning Ada

(Source : Wide_Wide_String)
return Unbounded_Wide_Wide_String
renames To_Unbounded_Wide_Wide_String;

function To_WWS
(Source : Unbounded WWide_String)
return Wide_Wide_String
renames To_Wide_Wide_String;

Hello World Arabic : constant
UTF_8_String := "أَلْهَمَ حَبَايَنَّا";
WWS Hello World Arabic : constant
Wide_Wide_String :=
   Decode (Hello_World_Arabic);

begin
UWWS := "Hello World: 
h& To_UWWS (WWS_Hello_World_Arabic);

Show_WW_String : declare
WWS : constant Wide_Wide_String :=
   To_WWS (UWWS);
begin
   Put_Line ("Wide_Wide_String Length: 
& WWS'Length'Image);
   Put_Line ("Wide_Wide_String Size: 
& WWS'Size'Image);
end Show_WW_String;

Put_Line ("-----------------------------");
Put_Line ("Converting Wide_Wide_String to UTF-8...");

Show_UTF_8_String : declare
S_UTF_8 : constant UTF_8_String :=
   Encode (To_WWS (UWWS));
begin
   Put_Line ("UTF-8 String: 
& S_UTF_8);
   Put_Line ("UTF-8 String Length: 
& S_UTF_8'Length'Image);
   Put_Line ("UTF-8 String Size: 
& S_UTF_8'Size'Image);
end Show_UTF_8_String;

end Show_WW_UTF_String;

Code block metadata
MD5: cecfb420bb80f42e7a65b793abcbef5

Runtime output
Wide_Wide_String Length: 26
Wide_Wide_String Size: 832
---------------------------------------
Converting Wide_Wide_String to UTF-8...
UTF-8 String: Hello World: أَلْهَمَ حَبَايَنَّا
(continues on next page)
In this application, we start by storing a string in Arabic in the Hello_World_Arabic constant. We then use the Decode function to convert that string from UTF_8_String type to Wide_Wide_String type — we store it in the WWS_Hello_World_Arabic constant.

We use a variable of type Unbounded_Wide_Wide_String (UWWS) to manipulate strings: we append the string in Arabic to the "Hello World: " string and store it in UWWS.

In the Show_WW_String block, we convert the string — stored in UWWS — from the Unbounded Wide_Wide_String type to the Wide_Wide_String type and display the length and size of the string. We do something similar in the Show_UTF_8_String block, but there, we convert to the UTF_8_Type type.

Also, in the Show_UTF_8_String block, we use the Encode function to convert that string from Wide_Wide_String type to then UTF_8_String type — we store it in the S_UTF_8 constant.

**UTF-8 size and length**

As you can see when running the last code example from the previous subsection, we have different sizes and lengths depending on the string type:

<table>
<thead>
<tr>
<th>String type</th>
<th>Size</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wide_Wide_String</td>
<td>832</td>
<td>26</td>
</tr>
<tr>
<td>UTF_8_String</td>
<td>296</td>
<td>37</td>
</tr>
</tbody>
</table>

The size needed for storing the string when using the Wide_Wide_String type is bigger than the one when using the UTF_8_String type. This is expected, as the Wide_Wide_String uses 32-bit characters, while the UTF_8_String type uses 8-bit codes to store the string in a more efficient way (memory-wise).

The length of the string using the Wide_Wide_String type is equivalent to the number of symbols we have in the original string: 26 characters / symbols. When using UTF-8, however, we may need more 8-bit codes to represent one symbol from the original string, so we may end up with a length value that is bigger than the actual number of symbols from the original string — as it is the case in this source-code example.

This difference in sizes might not always be the case. In fact, the sizes match when encoding a symbol in UTF-8 that requires four 8-bit codes. For example:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.UTF_Encoding;
use Ada.Strings.UTF_Encoding;
with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
use Ada.Strings.UTF_Encoding.Wide_Wide_Strings;

procedure Show_UTF_8 is
  Symbol_UTF_8 : constant UTF_8_String := "x";
  Symbol_WWS  : constant Wide_Wide_String :=
    Decode (Symbol_UTF_8);
```

(continues on next page)
begin
Put_Line ("Wide_Wide_String Length: 
& Symbol_WWS'Length'Image);
Put_Line ("Wide_Wide_String Size: 
& Symbol_WWS'Size'Image);
Put_Line ("UTF-8 String Length: 
& Symbol_UTF_8'Length'Image);
Put_Line ("UTF-8 String Size: 
& Symbol_UTF_8'Size'Image);
New_Line;
Put_Line ("UTF-8 String: 
& Symbol_UTF_8);
end Show_UTF_8;

In this case, both strings — using the Wide_Wide_String type or the UTF_8_String type — have the same size: 32 bits. (Here, we're using the \( \text{x} \) symbol from the Mathematical Alphanumeric Symbols block\(^{92}\), not the standard "x" from the Basic Latin block\(^{93}\).)

**UTF-8 encoding in source-code files**

In the past, it was common to use different character sets in text files when writing in different (human) languages. By default, Ada source-code files are expected to use the Latin-1 coding, which is a 8-bit character set.

Nowadays, however, using UTF-8 coding for text files — including source-code files — is very common. If your Ada code only uses standard ASCII characters, but you're saving it in a UTF-8 coded file, there's no need to worry about character sets, as UTF-8 is backwards compatible with ASCII.

However, you might want to use Unicode symbols in your Ada source code to declare constants — as we did in the previous sections — and store the source code in a UTF-8 coded file. In this case, you need be careful about how this file is parsed by the compiler.

Let's look at this source-code example:

**Listing 358: show_utf_8_strings.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.UTF_Encoding;
use Ada.Strings.UTF_Encoding;
```

---

\(^{92}\) [https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols](https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols)

Here, we're using Unicode symbols to initialize the Symbols_UTF_8 constant of UTF_8_String type.

Now, let's assume this source-code example is stored in a UTF-8 coded file. Because the "♥♫" string makes use of non-ASCII Unicode symbols, representing this string in UTF-8 format will require more than 2 bytes. In fact, each one of those Unicode symbols requires 2 bytes to be encoded in UTF-8. (Keep in mind that Unicode symbols may require between 1 to 4 bytes\(^\text{94}\) to be encoded in UTF-8 format.) Also, in this case, the UTF-8 encoding process is using two additional bytes. Therefore, the total length of the string is six, which matches what we see when running the Show_UTF_8_Strings procedure. In other words, the length of the Symbols_UTF_8 string doesn't refer to those two characters ("♥♫") that we were using in the constant declaration, but the length of the encoded bytes in its UTF-8 representation.

The UTF-8 format is very useful for storing and transmitting texts. However, if we want to process Unicode symbols, it's probably better to use string types with 32-bit characters — such as Wide_Wide_String. For example, let's say we want to use the "♥♫" string again to initialize a constant of Wide_Wide_String type:

\(^{94}\) https://en.wikipedia.org/wiki/UTF-8
In this case, as mentioned above, if we store this source code in a text file using UTF-8 format, we need to ensure that the UTF-8 coded symbols are correctly interpreted by the compiler when it parses the text file. Otherwise, we might get unexpected behavior. (Interpreting the characters in UTF-8 format as Latin-1 format is certainly an example of what we want to avoid here.)

In the GNAT toolchain

You can use UTF-8 coding in your source-code file and initialize strings of 32-bit characters. However, as we just mentioned, you need to make sure that the UTF-8 coded symbols are correctly interpreted by the compiler when dealing with types such as Wide_Wide_String. For this case, GNAT offers the -gnatW8 switch. Let’s run the previous example using this switch:

Listing 360: show_wws_strings.adb

```ada
with Ada.Text_IO;
with Ada.Wide_Wide_Text_IO;

procedure Show_WWS_Strings is
  package TIO renames Ada.Text_IO;
  package WWTIO renames Ada.Wide_Wide_Text_IO;
  Symbols_WWS : constant Wide_Wide_String := "♥♫";

begin
  WWTIO.Put_Line ("Wide_Wide_String: ",
                   Symbols_WWS);
  TIO.Put_Line ("Length: ",
                 Symbols_WWS'Length'Image);
end Show_WWS_Strings;
```

Runtime output

Wide_Wide_String: ♥♫
Length: 2

In this case, as mentioned above, if we store this source code in a text file using UTF-8 format, we need to ensure that the UTF-8 coded symbols are correctly interpreted by the compiler when it parses the text file. Otherwise, we might get unexpected behavior. (Interpreting the characters in UTF-8 format as Latin-1 format is certainly an example of what we want to avoid here.)
Because the **Wide_Wide_String** type has 32-bit characters, we expect the length of the string to match the number of symbols that we’re using. Indeed, when running the Show_WWS_Strings procedure, we see that the Symbols_WWS string has a length of two characters, which matches the number of characters of the "♥♫" string.

When we use the -gnatW8 switch, GNAT converts the UTF-8-coded string ("♥♫") to UTF-32 format, so we get two 32-bit characters. It then uses the UTF-32-coded string to initialize the Symbols_WWS string.

If we don’t use the -gnatW8 switch, however, we get wrong results. Let’s look at the same example again without the switch:

```
with Ada.Text_IO;
with Ada.Wide_Wide_Text_IO;

procedure Show_WWS_Strings is
  package TIO renames Ada.Text_IO;
  package WWTIO renames Ada.Wide_Wide_Text_IO;

  Symbols_WWS : constant Wide_Wide_String := "♥♫";

begin
  WWTIO.Put_Line ("Wide_Wide_String: " & Symbols_WWS);
  TIO.Put_Line ("Length: " & Symbols_WWS'Length'Image);
end Show_WWS_Strings;
```

Now, the "♥♫" string is being interpreted as a string of six 8-bit characters. (In other words, the UTF-8-coded string isn't converted to the UTF-32 format.) Each of those 8-bit characters is then stored in a 32-bit character of the Wide_Wide_String type. This explains why the Show_WWS_Strings procedure reports a length of 6 components for the Symbols_WWS string.
Portability of UTF-8 in source-code files

In a previous code example, we were assuming that the format that we use for the source-code file is UTF-8. This allows us to simply use Unicode symbols directly in strings:

```
Symbol_UTF_8 : constant UTF_8_String := "★";
```

This approach, however, might not be portable. For example, if the compiler uses a different string encoding for source-code files, it might interpret that Unicode character as something else — or just throw a compilation error.

If you're afraid that format mismatches might happen in your compilation environment, you may want to write strings in your code in a completely portable fashion, which consists in entering the exact sequence of codes in bytes — using the `Character'Val` function — for the symbols you want to use.

We can reuse parts of the previous example and replace the UTF-8 character with the corresponding UTF-8 code:

Listing 362: show_utf_8.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.UTF_Encoding; use Ada.Strings.UTF_Encoding;
procedure Show_UTF_8 is
  Symbol_UTF_8 : constant
    UTF_8_String :=
      Character'Val (16#e2#) & Character'Val (16#98#) & Character'Val (16#85#);
begin
  Put_Line ("UTF-8 String: "
     & Symbol_UTF_8);
end Show_UTF_8;
```

Code block metadata

MD5: 8ff02bc1793c0c5ac1ff24f62941af73

Runtime output

UTF-8 String: ★

Here, we use a sequence of three calls to the `Character'Val` function for the UTF-8 code that corresponds to the "★" symbol.
UTF-16 encoding and decoding

So far, we’ve discussed the UTF-8 encoding scheme. However, other encoding schemes exist and are supported as well. In fact, the Ada.Strings.UTF_Encoding package defines three encoding schemes:

```
type Encoding_Scheme is (UTF_8,
                         UTF_16BE,
                         UTF_16LE);
```

For example, instead of using UTF-8 encoding, we can use UTF-16 encoding — either in the big-endian or in the little-endian version. To convert between UTF-8 and UTF-16 encoding schemes, we can make use of the conversion functions from the Ada.Strings.UTF_Encoding.Conversions package.

To declare a UTF-16 encoded string, we can use one of the following data types:

- the 8-bit-character based UTF_String type, or
- the 16-bit-character based UTF_16_Wide_String type.

When using the 8-bit version, though, we have to specify the input and output schemes when converting between UTF-8 and UTF-16 encoding schemes.

Let’s see a code example that makes use of both UTF_String and UTF_16_Wide_String types:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.UTF_Encoding;
use Ada.Strings.UTF_Encoding;
with Ada.Strings.UTF_Encoding.Conversions;
use Ada.Strings.UTF_Encoding.Conversions;
procedure Show_UTF16_Types is
  Symbols_UTF_8  : constant UTF_8_String := "♥♫";
  Symbols_UTF_16 : constant UTF_16_Wide_String :=
    Convert (Symbols_UTF_8);
    -- ^ Calling Convert for UTF_8_String
    -- to UTF_16_Wide_String conversion.
  Symbols_UTF_16BE : constant
    UTF_String :=
    Convert (Item => Symbols_UTF_8,
              Input_Scheme => UTF_8,
              Output_Scheme => UTF_16BE);
    -- ^ Calling Convert for UTF_8_String
    -- to UTF_String conversion in UTF-16BE encoding.
    begin
    Put_Line ("UTF_8_String: "
              & Symbols_UTF_8);
    Put_Line ("UTF_16_Wide_String: "
              & Convert (Symbols_UTF_16));
    -- ^ Calling Convert for
    -- the UTF_16_Wide_String to
```
-- UTF_8_String conversion.

Put_Line
("UTF_String / UTF_16BE: "
& Convert
(Item => Symbols_UTF_16BE,
Input_Scheme => UTF_16BE,
Output_Scheme => UTF_8));
end Show_UTF16_Types;

In this example, we're declaring a UTF-8 encoded string and storing it in the Symbols_UTF_8 constant. Then, we're calling the Convert functions to convert between UTF-8 and UTF-16 encoding schemes. We're using two versions of this function:

- the Convert function that returns an object of UTF_16_Wide_String type for an input of UTF_8_String type, and
- the Convert function that returns an object of UTF_String type for an input of UTF_8_String type.

In this case, we need to specify the input and output schemes (see Input_Scheme and Output_Scheme parameters in the code example).

Previously, we’ve seen that the Ada.Strings.UTF_Encoding.Wide_Wide_Strings package offers functions to convert between UTF-8 and the Wide_Wide_String type. The same kind of conversion functions exist for UTF-16 strings as well. Let’s look at this code example:

Listing 364: show_ww_utf16_string.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.UTF_Encoding;
use Ada.Strings.UTF_Encoding;
with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
use Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
with Ada.Strings.UTF_Encoding.Conversions;
use Ada.Strings.UTF_Encoding.Conversions;

procedure Show_WW_UTF16_String is

Symbols_UTF_16 : constant
UTF_16_Wide_String :=
Wide_Character'Val (16#2665#) &
Wide_Character'Val (16#266B#);
-- ^ Calling Wide_Character’Val
-- to specify the UTF-16 BE code
-- for "♥" and "♫".

Symbols_WWS : constant
Wide_Wide_String :=
Decode (Symbols_UTF_16);
-- ^ Calling Decode for UTF_16 Wide String
to Wide_Wide_String conversion.
begin
  Put_Line ("UTF_16_Wide_String: "
    & Convert (Symbols_UTF_16));
  -- ^ Calling Convert for the
  -- UTF_16_Wide_String to
  -- UTF_8_String conversion.

  Put_Line ("Wide_Wide_String: "
    & Encode (Symbols_WWS));
  -- ^ Calling Encode for the
  -- Wide_Wide_String to
  -- UTF_8_String conversion.
end Show_WW_UTF16_String;

25.6.3 Image attribute

Overview

In the Introduction to Ada (page 13) course, we’ve seen that the Image attribute returns a string that contains a textual representation of an object. For example, we write Integer'Image (V) to get a string for the integer variable V:

Listing 365: show_simple_image.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Image is
  V : Integer;
begin
  V := 10;
  Put_Line ("V: " & Integer'Image (V));
end Show_Simple_Image;
```

Runtime output

V: 10
Naturally, we can use the Image attribute with other scalar types. For example:

Listing 366: show_simple_image.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Image is
   type Status is (Unknown, Off, On);
   V : Float;
   S : Status;
begin
   V := 10.0;
   S := Unknown;
   Put_Line ("V: " & Float'Image (V));
   Put_Line ("S: " & Status'Image (S));
end Show_Simple_Image;
```

In this example, we retrieve a string representing the floating-point variable V. Also, we use Status’ `Image` (V) to retrieve a string representing the textual version of the Status.

In the Ada Reference Manual

- Image Attributes

---

Type 'Image and Obj' Image

We can also apply the Image attribute to an object directly:

Listing 367: show_simple_image.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Image is
   V : Integer;
begin
   V := 10;
   Put_Line ("V: " & V'Image);
   -- Equivalent to:
   -- Put_Line ("V: " & Integer'Image (V));
end Show_Simple_Image;
```

---

85 http://www.ada-auth.org/standards/22rm/html/RM-4-10.html
In this example, the `Integer'Image (V)` and `V'Image` forms are equivalent.

### Wider versions of Image

Although we've been talking only about the Image attribute, it's important to mention that each of the wider versions of the string types also has a corresponding Image attribute. In fact, this is the attribute for each string type:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type of Returned String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>String</td>
</tr>
<tr>
<td>Wide_Image</td>
<td>Wide_String</td>
</tr>
<tr>
<td>Wide_Wide_Image</td>
<td>Wide_Wide_String</td>
</tr>
</tbody>
</table>

Let's see a simple example:

```
with Ada.Wide_Wide_Text_IO;
use Ada.Wide_Wide_Text_IO;

procedure Show_Wide_Wide_Image is
  F : Float;
begin
  F := 100.0;
  Put_Line ("F = " & F'Wide_Wide_Image);
end Show_Wide_Wide_Image;
```

In this example, we use the `Wide_Wide_Image` attribute to retrieve a string of `Wide_Wide_String` type for the floating-point variable `F`.

### Image attribute for non-scalar types

**Note:** This feature was introduced in Ada 2022.

In the previous code examples, we were using the Image attribute with scalar types, but it isn't restricted to those types. In fact, we can also use this attribute when dealing with non-scalar types. For example:
package Simple_Records is
  type Rec is limited private;
  type Rec_Access is access Rec;
  function Init return Rec;
  type Null_Rec is null record;
private
  type Rec is limited record
    F : Float;
    I : Integer;
  end record;
  function Init return Rec is
    ((F => 10.0, I => 4));
end Simple_Records;

pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Deallocation;
with Simple_Records; use Simple_Records;
procedure Show_Non_Scalar_Image is
  procedure Free is
    new Ada.Unchecked_Deallocation
      (Object => Rec, Name => Rec_Access);
    R_A : Rec_Access :=
      new Rec'(Init);
    N_R : Null_Rec :=
      (null record);
begin
  R_A := new Rec'(Init);
  N_R := (null record);
  Put_Line ("R_A: " & R_A'Image);
  Put_Line ("R_A.all: " & R_A.all'Image);
  Put_Line ("N_R: " & N_R'Image);
  Free (R_A);
  Put_Line ("R_A: " & R_A'Image);
end Show_Non_Scalar_Image;

Code block metadata
MD5: d7d15e96a03c882995262a5cfca5e771
Learning Ada

Runtime output

<table>
<thead>
<tr>
<th>R_A:</th>
<th>(access 10fb2c0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_A.all:</td>
<td></td>
</tr>
<tr>
<td>(F =&gt; 1.00000E+01, I =&gt; 4)</td>
<td></td>
</tr>
<tr>
<td>N_R:</td>
<td>(NULL RECORD)</td>
</tr>
<tr>
<td>R_A:</td>
<td>null</td>
</tr>
</tbody>
</table>

In the Show_Non_Scalar_Image procedure from this example, we display the access value of R_A and the contents of the dereferenced access object (R_A.all). Also, we see the indication that N_R is a null record and R_A is null after the call to Free.

Historically

Since Ada 2022, the Image attribute is available for all types. Prior to this version of the language, it was only available for scalar types. (For other kind of types, programmers had to use the Image attribute for each component of a record, for example.)

In fact, prior to Ada 2022, the Image attribute was described in the 3.5 Scalar Types96 section of the Ada Reference Manual, as it was only applied to those types. Now, it is part of the new Image Attributes97 section.

Let's see another example, this time with arrays:

Listing 371: show_array_image.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Array_Image is
  type Float_Array is
    array (Positive range <>) of Float;
  FA_3C : Float_Array (1 .. 3);
  FA_Null : Float_Array (1 .. 0);
begin
  FA_3C := [1.0, 3.0, 2.0];
  FA_Null := [];
  Put_Line ("FA_3C: " & FA_3C'Image);
  Put_Line ("FA_Null: " & FA_Null'Image);
end Show_Array_Image;
```

Code block metadata

MD5: 2d3fcd5e57451f08185618d357b705f

Runtime output

FA_3C: [1.00000E+00, 3.00000E+00, 2.00000E+00]
FA_Null: []

96 http://www.ada-auth.org/standards/22rm/html/RM-3-5.html
97 http://www.ada-auth.org/standards/22rm/html/RM-4-10.html
In this example, we display the values of the three components of the FA_3C array. Also, we display the null array FA_Null.

**Image attribute for tagged types**

In addition to untagged types, we can also use the `Image` attribute with tagged types. For example:

```ada
package Simple_Records is
  type Rec is tagged limited private;
  function Init return Rec;
  type Rec_Child is new Rec with private;
  overriding function Init return Rec_Child;
private
  type Status is (Unknown, Off, On);
  type Rec is tagged limited record
    F : Float;
    I : Integer;
  end record;
  function Init return Rec is
    ((F => 10.0, I => 4));
  type Rec_Child is new Rec with record
    Z : Status;
  end record;
  function Init return Rec_Child is
    (Rec'(Init) with Z => Off);
end Simple_Records;
```

```ada
pragma Ada_2022;
with Ada.Text_IO;  use Ada.Text_IO;
with Simple_Records; use Simple_Records;
procedure Show_Tagged_Image is
  R : constant Rec := Init;
  R_Class : constant Rec'Class := Rec'(Init);
  R_C : constant Rec_Child := Init;
begin
  Put_Line ("R: " & R'Image);
  Put_Line ("R_Class: " & R_Class'Image);
  Put_Line ("R_C: " & R_C'Image);
end Show_Tagged_Image;
```
In the Show_Tagged_Image procedure from this example, we display the contents of the R object of Rec type and the R_Class object of Rec 'Class type. Also, we display the contents of the R_C object of the Rec_Child type, which is derived from the Rec type.

Image attribute for task and protected types

We can also apply the Image attribute to protected objects and tasks:

Listing 374: simple_tasking.ads

```ada
package Simple_Tasking is
  protected type Protected_Float (I : Integer) is
    private
    V : Float := Float (I);
  end Protected_Float;

  protected type Protected_Null is
    private
  end Protected_Null;

  task type T is
    entry Start;
  end T;
end Simple_Tasking;
```

Listing 375: simple_tasking.adb

```ada
package body Simple_Tasking is
  protected body Protected_Float is
    end Protected_Float;
  end Protected_Float;

  protected body Protected_Null is
    end Protected_Null;
  end Protected_Null;

  task body T is
    begin
      accept Start;
    end T;
end Simple_Tasking;
```
Listing 376: show_protected_task_image.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Simple_Tasking; use Simple_Tasking;

procedure Show_Protected_Task_Image is
   PF : Protected_Float (0);
   PN : Protected_Null;
   T1 : T;

begin
   Put_Line ("PF: " & PF'Image);
   Put_Line ("PN: " & PN'Image);
   Put_Line ("T1: " & T1'Image);
   T1.Start;
end Show_Protected_Task_Image;
```

**Code block metadata**

MD5: 9d8c667015878eb14e5b3950a70b86b1

**Runtime output**

PF: (protected object)
PN: (protected object)
T1: (task t1_0000000000CCE090)

In this example, we display information about the protected object PF, the componentless protected object PN and the task T1.

### 25.6.4 Put_Image aspect

**Note:** This feature was introduced in Ada 2022.

**Overview**

In the previous section, we discussed many details about the Image attribute. In the code examples from that section, we've seen the default behavior of this attribute: the string returned by the calls to Image was always in the format defined by the Ada standard.

In some situations, however, we might want to customize the string that is returned by the Image attribute of a type T. Ada allows us to do that via the Put_Image aspect. This is what we have to do:

1. Specify the Put_Image aspect for the type T and indicate a procedure with a specific parameter profile — let's say, for example, a procedure named P.
2. Implement the procedure P and write the information we want to use into a buffer (by calling the routines defined for Root_Buffer_Type, such as the Put procedure).

We can see these steps performed in the code example below:
In the `Show_Put_Image` package, we use the `Put_Image` aspect in the declaration of the `T` type. There, we indicate that the `Image` attribute shall use the `Put_Image_T` procedure instead of the default version.

In the body of the `Put_Image_T` procedure, we implement our custom version of the `Image` attribute. We do that by calling the `Put` procedure with the information we want to provide in the `Image` attribute. Here, we access a buffer of `Root_Buffer_Type` type, which is defined in the `Ada.Strings.Text_Buffers` package. (We discuss more about this package later on (page 527).)

In the Ada Reference Manual

- Image Attributes\(^{98}\)

\(^{98}\) http://www.ada-auth.org/standards/22rm/html/RM-4-10.html
Complete Example of Put_Image

Let's see a complete example in which we use the Put_Image aspect and write useful information to the buffer:

Listing 379: custom_numerics.ads

```ada
pragma Ada_2022;
with Ada.Strings.Text_Buffers;

package Custom_Numerics is
  type Float_Integer is record
    F : Float := 0.0;
    I : Integer := 0;
  end record
  with Dynamic_Predicate =>
    Integer (Float_Integer.F) = Float_Integer.I, Put_Image => Put_Float_Integer;
  -- ^ Custom version of Put_Image

use Ada.Strings.Text_Buffers;

procedure Put_Float_Integer
  (Buffer : in out Root_Buffer_Type'Class;
   Arg : Float_Integer);
end Custom_Numerics;
```

Listing 380: custom_numerics.adb

```ada
package body Custom_Numerics is

procedure Put_Float_Integer
  (Buffer : in out Root_Buffer_Type'Class;
   Arg : Float_Integer) is
begin
  -- Call Wide_Wide_Put with customized
  -- information
  Buffer.Wide_Wide_Put
    ("(F : " & Arg.F'Wide_Wide_Image & ",
     & I : " & Arg.I'Wide_Wide_Image & ")");
end Custom_Numerics;
```

Listing 381: show_put_image.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Numerics; use Custom_Numerics;

procedure Show_Put_Image is
  V : Float_Integer;
begin
  V := (F => 100.2,
       I => 100);
```

(continues on next page)
Learning Ada

12 Put_Line ("V = " 
13 & V'Image); 
14 end Show_Put_Image;

Code block metadata

MD5: 18d31150d7a9ff9af0359495543c011f

Runtime output

V = (F : 1.00200E+02, I : 100)

In the Custom_Numerics package of this example, we specify the Put_Image aspect and indicate the Put_Float_Integer procedure. In that procedure, we display the information of components F and I. Then, in the Show_Put_Image procedure, we use the Image attribute for the V variable and see the information in the exact format we specified. (If you like to see the default version of the Put_Image instead, you may comment out the Put_Image aspect part in the declaration of Float_Integer.)

Relation to the Image attribute

Note that we cannot override the Image attribute directly — there's no Image aspect that we could specify. However, as we've just seen, we can do this indirectly by using our own version of the Put_Image procedure for a type T.

The Image attribute of a type T makes use of the procedure indicated in the Put_Image aspect. Let's say we have the following declaration:

```ada
type T is null record
   with Put_Image => Put_Image_T;
```

When we then use the T'Image attribute in our code, the custom Put_Image_T procedure is automatically called. This is a simplified example of how the Image function is implemented:

```ada
function Image (V : T) return String is
   Buffer : Custom_Buffer; -- ^ of Root_Buffer_Type'Class
begin
   -- Calling Put_Image procedure
   -- for type T
   Put_Image_T (Buffer, V);
   -- Retrieving the text from the
   -- buffer as a string
   return Buffer.Get;
end Image;
```

In other words, the Image attribute basically:

- calls the Put_Image procedure specified in the Put_Image aspect of type T's declaration and passes a buffer;

and

- retrieves the contents of the buffer as a string and returns it.
If the Put_Image aspect of type T isn't specified, the default version is used. (We've seen the default version of various types in the previous section (page 512) about the Image attribute.)

**Put_Image and derived types**

Types that were derived from untagged types (or null extensions) make use of the Put_Image procedure that was specified for their parent type — either a custom procedure indicated in the Put_Image aspect or the default one. Naturally, if a derived type has the Put_Image aspect, the procedure indicated in the aspect is used instead. For example:

```ada
pragma Ada_2022;
with Ada.Strings.Text_Buffers;
package Unagged_Put_Image is
    use Ada.Strings.Text_Buffers;

type T is null record
    with Put_Image => Put_Image_T;

procedure Put_Image_T
    (Buffer : in out Root_Buffer_TYPE'Class;
    Arg    :    T);

type T_Derived_1 is new T;

type T_Derived_2 is new T
    with Put_Image => Put_Image_T_Derived_2;

procedure Put_Image_T_Derived_2
    (Buffer : in out Root_Buffer_TYPE'Class;
    Arg    :    T_Derived_2);
end Unagged_Put_Image;
```

Listing 382: untagged_put_image.ads

```ada
package body Unagged_Put_Image is
    procedure Put_Image_T
        (Buffer : in out Root_Buffer_TYPE'Class;
        Arg    :    T) is
    pragma Unreferenced (Arg);
    begin
        Buffer.Wide_Wide_put ("Put_Image_T");
    end Put_Image_T;

    procedure Put_Image_T_Derived_2
        (Buffer : in out Root_Buffer_TYPE'Class;
        Arg    :    T_Derived_2) is
    pragma Unreferenced (Arg);
    begin
        Buffer.Wide_Wide_put ("Put_Image_T_Derived_2");
    end Put_Image_T_Derived_2;
end Unagged_Put_Image;
```

Listing 383: untagged_put_image.adb
Listing 384: show_untagged_put_image.adb

```
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;
with Untagged_Put_Image; use Untagged_Put_Image;

procedure Show_Untagged_Put_Image is
  Obj_T : T;
  Obj_T_Derived_1 : T_Derived_1;
  Obj_T_Derived_2 : T_Derived_2;
begin
  Put_Line ("T'Image : 
    & Obj_T'Image);
  Put_Line ("T_Derived_1'Image : 
    & Obj_T_Derived_1'Image);
  Put_Line ("T_Derived_2'Image : 
    & Obj_T_Derived_2'Image);
end Show_Untagged_Put_Image;
```

In this example, we declare the type T and its derived types T_Derived_1 and T_Derived_2. When running this code, we see that:

- T_Derived_1 makes use of the Put_Image_T procedure from its parent.
  - Note that, if we remove the Put_Image aspect from the declaration of T, the default version of the Put_Image procedure is used for both T and T_Derived_1 types.
- T_Derived_2 makes use of the Put_Image_T_Derived_2 procedure, which was indicated in the Put_Image aspect of that type, instead of its parent's procedure.

### Put_Image and tagged types

Types that are derived from a tagged type may also inherit the Put_Image aspect. However, there are a couple of small differences in comparison to untagged types, as we can see in the following example:

Listing 385: tagged_put_image.ads

```
pragma Ada_2022;

with Ada.Strings.Text_Buffers;

package Tagged_Put_Image is
  use Ada.Strings.Text_Buffers;
```
type T is tagged record
  I : Integer := 0;
end record
  with Put_Image => Put_Image_T;

procedure Put_Image_T
  (Buffer : in out Root_Buffer_Type'Class;
   Arg : T);

procedure Put_Image_T_Child_3
  (Buffer : in out Root_Buffer_Type'Class;
   Arg : T_Child_3);

end Tagged_Put_Image;

package body Tagged_Put_Image is
  procedure Put_Image_T
    (Buffer : in out Root_Buffer_Type'Class;
     Arg : T) is
  begin
    pragma Unreferenced (Arg);
    Buffer.Wide_Wide_Put ("Put_Image_T");
  end Put_Image_T;

  procedure Put_Image_T_Child_3
    (Buffer : in out Root_Buffer_Type'Class;
     Arg : T_Child_3) is
  begin
    Buffer.Wide_Wide_Put ("Put_Image_T_Child_3");
  end Put_Image_T_Child_3;

end Tagged_Put_Image;

pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;
with Tagged_Put_Image; use Tagged_Put_Image;

procedure Show_Tagged_Put_Image is
  Obj_T : T;
  Obj_T_Child_1 : T_Child_1;
In this example, we declare the type \( T \) and its derived types \( T_{\text{Child}_1}, T_{\text{Child}_2} \) and \( T_{\text{Child}_3} \). When running this code, we see that:

- for both \( T_{\text{Child}_1} \) and \( T_{\text{Child}_2} \), the parent's \( \text{Put}_\text{Image} \) aspect (the \( \text{Put}_\text{Image}_T \) procedure) is called and its information is combined with the information from the type extension;
  - The information from the parent's \( \text{Put}_\text{Image}_T \) procedure is presented in an aggregate syntax — in this case, this results in \( (\text{Put}_\text{Image}_T \text{ with } I1 \Rightarrow 0) \).
  - For the \( T_{\text{Child}_1} \) type, the \( I1 \) component of the type extension is displayed by calling a default version of the \( \text{Put}_\text{Image} \) procedure for that component — \( (\text{Put}_\text{Image}_T \text{ with } I1 \Rightarrow 0) \) is displayed.
  - For the \( T_{\text{Child}_2} \) type, no additional information is displayed because this type has a null extension.

- for the \( T_{\text{Child}_3} \) type, the \( \text{Put}_\text{Image}_T_{\text{Child}_3} \) procedure, which was indicated in the \( \text{Put}_\text{Image} \) aspect of the type, is used.

Finally, class-wide types (such as \( T\text{'Class} \)) include additional information. Here, the tag of the specific derived type is displayed first — in this case, the tag of the \( T_{\text{Child}_1} \) type — and then the actual information for the derived type is displayed.
25.6.5 Universal text buffer

In the previous section (page 519), we've seen that the first parameter of the procedure indicated in the Put_Image aspect has the Root_Buffer_Type'Class type, which is defined in the Ada.Strings.Text_Buffers package. In this section, we talk more about this type and additional procedures associated with this type.

Note: This feature was introduced in Ada 2022.

Overview

We use the Root_Buffer_Type'Class type to implement a universal text buffer that is used to store and retrieve information about data types. Because this text buffer isn't associated with specific data types, it is universal — in the sense that we can really use it for any data type, regardless of the characteristics of this type.

In theory, we could use Ada's universal text buffer to implement applications that actually process text in some form — for example, when implementing a text editor. However, in general, Ada programmers are only expected to make use of the Root_Buffer_Type'Class type when implementing a procedure for the Put_Image aspect. For this reason, we won't discuss any kind of type derivation — or any other kind of usages of this type — in this section. Instead, we'll just focus on additional subprograms from the Ada.Strings.Text_Buffers package.

In the Ada Reference Manual

- Universal Text Buffers

Additional procedures

In the previous section, we used the Put procedure — and the related Wide_Put and Wide_Wide_Put procedures — from the Ada.Strings.Text_Buffers package. In addition to these procedures, the package also includes:

- the New_Line procedure, which writes a new line marker to the text buffer;
- the Increase_Indent procedure, which increases the indentation in the text buffer; and
- the Decrease_Indent procedure, which decreases the indentation in the text buffer.

The Ada.Strings.Text_Buffers package also includes the Current_Indent function, which retrieves the current indentation counter.

Let's revisit an example from the previous section and use the procedures mentioned above:

Listing 388: custom_numerics.ads

```ada
pragma Ada_2022;
with Ada.Strings.Text_Buffers;
package Custom_Numerics is
  type Float_Integer is record
  end record;
end Custom_Numerics;
```

25.6. Strings
Listing 389: custom_numerics.adb

```ada
package body Custom_Numerics is

procedure Put_Float_Integer
(Buffer : in out Root_Buffer_Type'Class;
   Arg  :   Float_Integer)
is
begin
  Buffer.Wide_Wide_Put ("(");
  Buffer.New_Line;
  Buffer.Increase_Indent;
  Buffer.Wide_Wide_Put ("F : ";
                        & Arg.F'Wide_Wide_Image);
  Buffer.New_Line;
  Buffer.Wide_Wide_Put ("I :");
  Buffer.Decrease_Indent;
  Buffer.New_Line;
  Buffer.Wide_Wide_Put (")");
end Put_Float_Integer;

end Custom_Numerics;
```

Listing 390: show_put_image.adb

```ada
pragma Ada_2022;

with Ada.Text_IO;   use Ada.Text_IO;
with Custom_Numerics; use Custom_Numerics;

procedure Show_Put_Image is
  V : Float_Integer;
begin
  V := (F => 100.2,
        I => 100);
```

(continues on next page)
In the body of the `Put_Float_Integer` procedure, we're using the `New_Line`, `Increase_Indent` and `Decrease_Indent` procedures to improve the format of the string returned by the `Float_Integer'Image` attribute. Using these procedures, you can create any kind of output format for your custom type.

### 25.7 Numerics

#### 25.7.1 Modular Types

In the Introduction to Ada course, we've seen that Ada has two kinds of integer type: `signed` (page 47) and `modular` (page 50) types. For example:

```ada
package Num_Types is
  type Signed_Integer is range 1 .. 1_000_000;
  type Modular is mod 2**32;
end Num_Types;
```

In this section, we discuss two attributes of modular types: Modulus and `Mod`. We also discuss operations on modular types.

**In the Ada Reference Manual**

- [3.5.4 Integer Types](http://www.ada-auth.org/standards/22rm/html/RM-3-5-4.html)
Learning Ada

Modulus Attribute

The Modulus attribute returns the modulus of the modular type as a universal integer value. Let's get the modulus of the 32-bit Modular type that we've declared in the Num_Types package of the previous example:

Listing 392: show_modular.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Num_Types; use Num_Types;
procedure Show_Modular is
  Modulus_Value : constant := Modular'Modulus;
begin
  Put_Line (Modulus_Value'Image);
end Show_Modular;
```

Code block metadata

MD5: 336254ebc8c09ee9921633f6919994fe

Runtime output

4294967296

When we run this example, we get 4294967296, which is equal to \(2^{32}\).

Mod Attribute

Note: This section was originally written by Robert A. Duff and published as Gem #26: The Mod Attribute.¹⁰¹

Operations on signed integers can overflow: if the result is outside the base range, Constraint_Error will be raised. In our previous example, we declared the Signed_Integer type:

```ada
type Signed_Integer is range 1 .. 1_000_000;
```

The base range of Signed_Integer is the range of Signed_Integer'Base, which is chosen by the compiler, but is likely to be something like \(-2^{31} .. 2^{31} - 1\). (Note: we discussed the Base attribute in this section (page 283).)

Operations on modular integers use modular (wraparound) arithmetic. For example:

Listing 393: show_modular.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Num_Types; use Num_Types;
procedure Show_Modular is
  X : Modular;
begin
  X := 1;
  Put_Line (X'Image);
end Show_Modular;
```

¹⁰¹ https://www.adacore.com/gems/gem-26

Chapter 25. Data types
Negating X gives -1, which wraps around to $2^{32} - 1$, i.e. all-one-bits.

But what about a type conversion from signed to modular? Is that a signed operation (so it should overflow) or is it a modular operation (so it should wrap around)? The answer in Ada is the former — that is, if you try to convert, say, `Integer'(-1)` to `Modular`, you will get `Constraint_Error`:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Num_Types; use Num_Types;

procedure Show_Modular is
  I : Integer := -1;
  X : Modular := 1;
begin
  X := Modular (I); -- raises Constraint_Error
  Put_Line (X'Image);
end Show_Modular;
```

To solve this problem, we can use the `Mod` attribute:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Num_Types; use Num_Types;
```

(continues on next page)
procedure Show_Modular is
  I : constant Integer := -1;
  X : Modular := 1;
begin
  X := Modular'Mod (I);
  Put_Line (X'Image);
end Show_Modular;

The Mod attribute will correctly convert from any integer type to a given modular type, using wraparound semantics.

Historically
In older versions of Ada — such as Ada 95 —, the only way to do this conversion is to use Unchecked_Conversion, which is somewhat uncomfortable. Furthermore, if you're trying to convert to a generic formal modular type, how do you know what size of signed integer type to use? Note that Unchecked_Conversion might malfunction if the source and target types are of different sizes.

The Mod attribute was added to Ada 2005 to solve this problem. Also, we can now safely use this attribute in generics. For example:

Listing 396: mod_attribute.ads

generic
type Formal_Modular is mod <>;
package Mod_Attribute is
  function F return Formal_Modular;
end Mod_Attribute;

Listing 397: mod_attribute.adb

generic

type Formal_Modular is mod <>;
package Mod_Attribute is
  function F return Formal_Modular;
end Mod_Attribute;

In this example, F will return the all-ones bit pattern, for whatever modular type is passed to Formal_Modular.
Operations on modular types

Modular types are particularly useful for bit manipulation. For example, we can use the `and`, `or`, `xor` and `not` operators for modular types.

Also, we can perform bit-shifting by multiplying or dividing a modular object with a power of two. For example, if `M` is a variable of modular type, then `M := M * 2 ** 3` shifts the bits to the left by three bits. Likewise, `M := M / 2 ** 3` shifts the bits to the right. Note that the compiler selects the appropriate shifting operator when translating these operations to machine code — no actual multiplication or division will be performed.

Let’s see a simple implementation of the CRC-CCITT (0x1D0F) algorithm:

Listing 398: crc_defs.ads

```ada
package Crc_Defs is

  type Byte is mod 2 ** 8;
  type Crc is mod 2 ** 16;

  type Byte_Array is
    array (Positive range <>) of Byte;

  function Crc_CCITT (A : Byte_Array) return Crc;

  procedure Display (Crc_A : Crc);
  procedure Display (A : Byte_Array);

end Crc_Defs;
```

Listing 399: crc_defs.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Crc_Defs is

  package Byte_IO is new Modular_IO (Byte);
  package Crc_IO is new Modular_IO (Crc);

  function Crc_CCITT (A : Byte_Array) return Crc is
    X       : Byte;
    Crc_A   : Crc := 16#1d0f#;
  begin
    for I in A'Range loop
      X := Byte (Crc_A / 2 ** 8) xor A (I);
      X := X xor (X / 2 ** 4);
      declare
        Crc_X : constant Crc := Crc (X);
      begin
        Crc_A := Crc_A * 2 ** 8 xor Crc_X * 2 ** 12 xor Crc_X * 2 ** 5 xor Crc_X;
      end;
    end loop;
    return Crc_A;
  end Crc_CCITT;
```

(continues on next page)
procedure Display (Crc_A : Crc) is  
begin  
Crc_IO.Put (Crc_A);  
New_Line;  
end Display;  

procedure Display (A : Byte_Array) is  
begin  
for E of A loop  
Byte_IO.Put (E);  
Put (", ");  
end loop;  
New_Line;  
end Display;  

begin  
Byte_IO.Default_Width := 1;  
Byte_IO.Default_Base := 16;  
Crc_IO.Default_Width := 1;  
Crc_IO.Default_Base := 16;  
end Crc_Defs;  

with Ada.Text_IO; use Ada.Text_IO;  
with Crc_Defs; use Crc_Defs;  

procedure Show_Crc is  
AA : constant Byte_Array :=  
(16#0#, 16#20#, 16#30#);  
Crc_A : Crc;  
begin  
Crc_A := Crc_CCITT (AA);  
Put ("Input array: ");  
Display (AA);  
Put ("CRC-CCITT: ");  
Display (Crc_A);  
end Show_Crc;  

In this example, the core of the algorithm is implemented in the Crc_CCITT function. There, we use bit shifting — for instance, \( * 2 \) \( \times 8 \) and \( / 2 \) \( ^8 \), which shift left and right, respectively, by eight bits. We also use the xor operator.
25.7.2 Numeric Literals

Classification

We’ve already discussed basic characteristics of numeric literals in the Introduction to Ada course — although we haven’t used this terminology there. There are two kinds of numeric literals in Ada: integer literals and real literals. They are distinguished by the absence or presence of a radix point. For example:

Listing 401: real_integer_literals.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Real_Integer_Literals is
  Integer_Literal : constant := 365;
  Real_Literal    : constant := 365.2564;
begin
  Put_Line ("Integer Literal: ", Image (Integer_Literal));
  Put_Line ("Real Literal: ", Image (Real_Literal));
end Real_Integer_Literals;
```

Runtime output

Integer Literal: 365
Real Literal: 3.65256400000000000E+02

Another classification takes the use of a base indicator into account. (Remember that, when writing a literal such as 2#1011#, the base is the element before the first # sign.) So here we distinguish between decimal literals and based literals. For example:

Listing 402: decimal_based_literals.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Decimal_Based_Literals is
  package F_IO is new Ada.Text_IO.Float_IO (Float);

  -- DECIMAL LITERALS
  Dec_Integer  : constant := 365;
  Dec_Real     : constant := 365.2564;
  Dec_Real_Exp : constant := 0.365_256_4e3;

  -- BASED LITERALS
  Based_Integer : constant := 16#16D#;
```

(continues on next page)
Based literals use the base#number# format. Also, they aren't limited to simple integer literals such as 16#16D#. In fact, we can use a radix point or an exponent in based literals, as well as underscores. In addition, we can use any base from 2 up to 16. We discuss these aspects further in the next section.
Neat and Flexibility

**Note:** This section was originally written by Franco Gasperoni and published as Gem #7: The Beauty of Numeric Literals in Ada\(^{102}\).

Ada provides a simple and elegant way of expressing numeric literals. One of those simple, yet powerful aspects is the ability to use underscores to separate groups of digits. For example, \(3.14159_{26535}\_89793\_23846\_26433\_83279\_50288\_41971\_69399\_37510\) is more readable and less error prone to type than \(3.14159265358979323846264338327950288419716939937510\). Here’s the complete code:

```ada
with Ada.Text_IO;

procedure Ada_Numeric_Literals is
  Pi : constant := 3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510;
  Pi2 : constant := 3.14159265358979323846264338327950288419716939937510;
  Z : constant := Pi - Pi2;
pragma Assert (Z = 0.0);

begin
  use Ada.Text_IO;
  Put_Line ("Z = " & Float’Image (Z));
end Ada_Numeric_Literals;
```

Code block metadata

MD5: 8f6516730fa98f08234b1599488431aaf

Runtime output

```
Z = 0.00000E+00
```

Also, when using based literals, Ada allows any base from 2 to 16. Thus, we can write the decimal number 136 in any one of the following notations:

```ada
with Ada.Text_IO;

procedure Ada_Numeric_Literals is
  Bin_136 : constant := 2#1000_1000#;
  Oct_136 : constant := 8#210#;
  Dec_136 : constant := 10#136#;
  Hex_136 : constant := 16#88#;
pragma Assert (Bin_136 = 136);
pragma Assert (Oct_136 = 136);
pragma Assert (Dec_136 = 136);
pragma Assert (Hex_136 = 136);

begin
  use Ada.Text_IO;
end Ada_Numeric_Literals;
```

\(^{102}\) [https://www.adacore.com/gems/ada-gem-7](https://www.adacore.com/gems/ada-gem-7)
begin
  Put_Line ("Bin_136 = " & Integer'Image (Bin_136));
  Put_Line ("Oct_136 = " & Integer'Image (Oct_136));
  Put_Line ("Dec_136 = " & Integer'Image (Dec_136));
  Put_Line ("Hex_136 = " & Integer'Image (Hex_136));
end Ada_Numeric_Literals;

In other languages

The rationale behind the method to specify based literals in the C programming language
is strange and unintuitive. Here, you have only three possible bases: 8, 10, and 16 (why no
base 2?). Furthermore, requiring that numbers in base 8 be preceded by a zero feels like a
bad joke on us programmers. For example, what values do 0210 and 210 represent in C?

When dealing with microcontrollers, we might encounter I/O devices that are memory
mapped. Here, we have the ability to write:

Lights_On : constant := 2#1000_1000#;
Lights_Off : constant := 2#0111_0111#;

and have the ability to turn on/off the lights as follows:

Output_Devices := Output_Devices or Lights_On;
Output_Devices := Output_Devices and Lights_Off;

Here's the complete example:

Listing 405: ada_numeric_literals.adb

with Ada.Text_IO;

procedure Ada_Numeric_Literals is
  Lights_On : constant := 2#1000_1000#;
  Lights_Off : constant := 2#0111_0111#;

  type Byte is mod 256;
  Output_Devices : Byte := 0;

  -- for Output_Devices'Address
  -- use 16#DEAD_BEEF#
  -- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  -- Memory mapped Output

(continues on next page)
use Ada.Text_IO;
begin
Output_Devices := Output_Devices or Lights_On;
Put_Line ("Output_Devices (lights on ) = "
   & Byte'Image (Output_Devices));
Output_Devices := Output_Devices and Lights_Off;
Put_Line ("Output_Devices (lights off) = "
   & Byte'Image (Output_Devices));
end Ada_Numeric_Literals;

Code block metadata
MD5: c3e72b25366d8d815a1f425f2695ad0b

Runtime output
Output_Devices (lights on ) = 136
Output_Devices (lights off) = 0

Of course, we can also use records with representation clauses (page 367) to do the above, which is even more elegant.

The notion of base in Ada allows for exponents, which is particularly pleasant. For instance, we can write:

Listing 406: literal_binaries.ads

package Literal_Binaries is
   Kilobyte : constant := 2#1#e+10;
   Megabyte : constant := 2#1#e+20;
   Gigabyte : constant := 2#1#e+30;
   Terabyte : constant := 2#1#e+40;
   Petabyte : constant := 2#1#e+50;
   Exabyte : constant := 2#1#e+60;
   Zettabyte : constant := 2#1#e+70;
   Yottabyte : constant := 2#1#e+80;
end Literal_Binaries;

Code block metadata
MD5: 98d971e0f170db570069f8868e442d6d

In based literals, the exponent — like the base — uses the regular decimal notation and specifies the power of the base that the based literal should be multiplied with to obtain the final value. For instance 2#1#e+10 = 1 x 2^10 = 1_024 (in base 10), whereas 16#F#e+2 = 15 x 16^2 = 15 x 256 = 3_840 (in base 10).

Based numbers apply equally well to real literals. We can, for instance, write:

One_Third : constant := 3#0.1#;
-- ^^^^^^^
-- same as 1.0/3

Whether we write 3#0.1# or 1.0 / 3, or even 3#1.0#e-1, Ada allows us to specify exactly rational numbers for which decimal literals cannot be written.
Learning Ada

The last nice feature is that Ada has an open-ended set of integer and real types. As a result, numeric literals in Ada do not carry with them their type as, for example, in C. The actual type of the literal is determined from the context. This is particularly helpful in avoiding overflows, underflows, and loss of precision.

In other languages

In C, a source of confusion can be the distinction between 321 and 321. Although both look similar, they're actually very different from each other.

And this is not all: all constant computations done at compile time are done in infinite precision, be they integer or real. This allows us to write constants with whatever size and precision without having to worry about overflow or underflow. We can for instance write:

```
Zero : constant := 1.0 - 3.0 * One_Third;
```

and be guaranteed that constant Zero has indeed value zero. This is very different from writing:

```
One_Third_Approx : constant :=
0.33333333333333333333333333333;
Zero_Approx := constant :=
1.0 - 3.0 * One_Third_Approx;
```

where Zero_Approx is really 1.0e-29 — and that will show up in your numerical computations. The above is quite handy when we want to write fractions without any loss of precision. Here's the complete code:

Listing 407: ada_numeric_literals.adb

```
with Ada.Text_IO;

procedure Ada_Numeric_Literals is
   One_Third : constant := 3#1.0#e-1;
   Zero : constant := 1.0 - 3.0 * One_Third;
   pragma Assert (Zero = 0.0);

   One_Third_Approx : constant :=
0.33333333333333333333333333333;
   Zero_Approx := constant :=
1.0 - 3.0 * One_Third_Approx;

   use Ada.Text_IO;

begin
   Put_Line ("Zero = ", Float'Image (Zero));
   Put_Line ("Zero_Approx = ", Float'Image (Zero_Approx));
end Ada_Numeric_Literals;
```

Code block metadata

MD5: ee604245b34e8cb878a8ebdb21cd564e

Runtime output
Learning Ada

Zero = 0.00000E+00
Zero_Approx = 1.00000E-29

Along these same lines, we can write:

Listing 408: ada_numeric_literals.adb

with Ada.Text_IO;

with Literal_Binaries; use Literal_Binaries;

procedure Ada_Numeric_Literals is

Big_Sum : constant := 1 + Kilobyte + Megabyte + Gigabyte + Terabyte + Petabyte + Exabyte + Zettabyte;

Result : constant := (Yottabyte - 1) / (Kilobyte - 1);

Nil : constant := Result - Big_Sum;
pragma Assert (Nil = 0);

use Ada.Text_IO;

begin
  Put_Line ("Nil = " & Integer'Image (Nil));
end Ada_Numeric_Literals;

and be guaranteed that Nil is equal to zero.

25.7.3 Floating-Point Types

In this section, we discuss various attributes related to floating-point types.

In the Ada Reference Manual

- 3.5.8 Operations of Floating Point Types
- A.5.3 Attributes of Floating Point Types


25.7. Numerics
Learning Ada

**Representation-oriented attributes**

In this section, we discuss attributes related to the representation of floating-point types.

**Attribute: Machine_Radix**

Machine_Radix is an attribute that returns the radix of the hardware representation of a type. For example:

```
Listing 409: show_machine_radix.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Machine_Radix is
begin
  Put_Line ("Float'Machine_Radix: " & Float'Machine_Radix'image);
  Put_Line ("Long_Float'Machine_Radix: " & Long_Float'Machine_Radix'image);
  Put_Line ("Long_Long_Float'Machine_Radix: " & Long_Long_Float'Machine_Radix'image);
end Show_Machine_Radix;
```

**Code block metadata**


MD5: 88680df680f1db4ff803912bb50370551

**Runtime output**

```
Float'Machine_Radix: 2
Long_Float'Machine_Radix: 2
Long_Long_Float'Machine_Radix: 2
```

Usually, this value is two, as the radix is based on a binary system.

**Attributes: Machine_Mantissa**

Machine_Mantissa is an attribute that returns the number of bits reserved for the mantissa of the floating-point type. For example:

```
Listing 410: show_machine_mantissa.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Machine_Mantissa is
begin
  Put_Line ("Float'Machine_Mantissa: " & Float'Machine_Mantissa'image);
  Put_Line ("Long_Float'Machine_Mantissa: " & Long_Float'Machine_Mantissa'image);
end Show_Machine_Mantissa;
```

(continues on next page)
On a typical desktop PC, as indicated by Machine_Mantissa, we have 24 bits for the floating-point mantissa of the Float type.

**Machine_Emin and Machine_Emax**

The Machine_Emin and Machine_Emax attributes return the minimum and maximum value, respectively, of the machine exponent the floating-point type. Note that, in all cases, the returned value is a universal integer. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Machine_Emin_Emax is
begin
  Put_Line ("Float'Machine_Emin: " & Float'Machine_Emin'Image);
  Put_Line ("Float'Machine_Emax: " & Float'Machine_Emax'Image);
  Put_Line ("Long_Float'Machine_Emin: " & Long_Float'Machine_Emin'Image);
  Put_Line ("Long_Float'Machine_Emax: " & Long_Float'Machine_Emax'Image);
  Put_Line ("Long_Long_Float'Machine_Emin: " & Long_Long_Float'Machine_Emin'Image);
  Put_Line ("Long_Long_Float'Machine_Emax: " & Long_Long_Float'Machine_Emax'Image);
end Show_Machine_Emin_Emax;
```

**Runtime output**

Float'Machine_Emin: 24
Float'Machine_Emax: 253
Long_Float'Machine_Emin: 53
Long_Float'Machine_Emax: 532
Long_Long_Float'Machine_Emin: 64
Long_Long_Float'Machine_Emax: 645
Learning Ada

Float'Machine_Emin: -125  
Float'Machine_Emax:  128  
Long_Float'Machine_Emin: -1021  
Long_Float'Machine_Emax:  1024  
Long_Long_Float'Machine_Emin: -16381  
Long_Long_Float'Machine_Emax:  16384

On a typical desktop PC, the value of Float'Machine_Emin and Float'Machine_Emax is -125 and 128, respectively.

To get the actual minimum and maximum value of the exponent for a specific type, we need to use the Machine_Radix attribute that we've seen previously. Let's calculate the minimum and maximum value of the exponent for the Float type on a typical PC:

- Value of minimum exponent: Float'Machine_Radix ** Float'Machine_Emin.
  - In our target platform, this is $2^{-125} = 2.35098870164457501594 \times 10^{-38}$.
- Value of maximum exponent: Float'Machine_Radix ** Float'Machine_Emax.
  - In our target platform, this is $2^{128} = 3.40282366920938463463 \times 10^{38}$.

**Attribute: Digits**

Digits is an attribute that returns the requested decimal precision of a floating-point subtype. Let's see an example:

Listing 412: show_digits.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Digits is
begin
  Put_Line ("Float'Digits: ", Float'Digits'Image);
  Put_Line ("Long_Float'Digits: ", Long_Float'Digits'Image);
  Put_Line ("Long_Long_Float'Digits: ", Long_Long_Float'Digits'Image);
end Show_Digits;
```

Here, the requested decimal precision of the Float type is six digits.

Note that we said that Digits is the requested level of precision, which is specified as part of declaring a floating point type. We can retrieve the actual decimal precision with Base'Digits. For example:
Listing 413: show_base_digits.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Base_Digits is
  type Float_D3 is new Float digits 3;
begin
  Put_Line ("Float_D3'Digits: " & Float_D3'Digits'Image);
  Put_Line ("Float_D3'Base'Digits: " & Float_D3'Base'Digits'Image);
end Show_Base_Digits;

Code block metadata

MD5: a2deb352f93511ab2a39d41f0b3f9512

Runtime output

Float_D3'Digits: 3
Float_D3'Base'Digits: 6

The requested decimal precision of the Float_D3 type is three digits, while the actual decimal precision is six digits (on a typical desktop PC).

Attributes: Denorm, Signed_Zeros, Machine_Rounds, Machine_Overflows

In this section, we discuss attributes that return Boolean values indicating whether a feature is available or not in the target architecture:

- Denorm is an attribute that indicates whether the target architecture uses denormalized numbers\(^{105}\).
- Signed_Zeros is an attribute that indicates whether the type uses a sign for zero values, so it can represent both -0.0 and 0.0.
- Machine_Rounds is an attribute that indicates whether rounding-to-nearest is used, rather than some other choice (such as rounding-toward-zero).
- Machine_Overflows is an attribute that indicates whether a Constraint_Error exception is (or is not) guaranteed to be raised when an operation with that type produces an overflow or divide-by-zero.

Listing 414: show_boolean_attributes.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Boolean_Attributes is
begin
  Put_Line ("Float'Denorm: " & Float'Denorm'Image);
  Put_Line ("Long_Float'Denorm: " & Long_Float'Denorm'Image);
  Put_Line ("Long_Long_Float'Denorm: " & Long_Long_Float'Denorm'Image);
end Show_Boolean_Attributes;

\(^{105}\) https://en.wikipedia.org/wiki/Subnormal_number

25.7. Numerics 545
On a typical PC, we have the following information:

- **Denorm** is true (i.e. the architecture uses denormalized numbers);
- **Signed_Zeros** is true (i.e. the standard floating-point types use a sign for zero values);
- **Machine_Rounds** is true (i.e. rounding-to-nearest is used for floating-point types);
- **Machine_Overflows** is false (i.e. there's no guarantee that a Constraint_Error exception is raised when an operation with a floating-point type produces an overflow or divide-by-zero).
Primitive function attributes

In this section, we discuss attributes that we can use to manipulate floating-point values.

Attributes: Fraction, Exponent and Compose

The Exponent and Fraction attributes return "parts" of a floating-point value:

- Exponent returns the machine exponent, and
- Fraction returns the mantissa part.

Compose is used to return a floating-point value based on a fraction (the mantissa part) and the machine exponent.

Let's see some examples:

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Exponent_Fraction_Compose is
begin
  Put_Line ("Float'Fraction (1.0): " & Float'Fraction (1.0)'Image);
  Put_Line ("Float'Fraction (0.25): " & Float'Fraction (0.25)'Image);
  Put_Line ("Float'Fraction (1.0e-25): " & Float'Fraction (1.0e-25)'Image);
  Put_Line ("Float'Exponent (1.0): " & Float'Exponent (1.0)'Image);
  Put_Line ("Float'Exponent (0.25): " & Float'Exponent (0.25)'Image);
  Put_Line ("Float'Exponent (1.0e-25): " & Float'Exponent (1.0e-25)'Image);
  Put_Line ("Float'Compose (5.00000e-01, 1): " & Float'Compose (5.00000e-01, 1)'Image);
  Put_Line ("Float'Compose (5.00000e-01, -1): " & Float'Compose (5.00000e-01, -1)'Image);
  Put_Line ("Float'Compose (9.67141E-01, -83): " & Float'Compose (9.67141E-01, -83)'Image);
end Show_Exponent_Fraction_Compose;
```

Code block metadata

MD5: d2e61b6b9a7a50861145f6b65e9fac39

Runtime output

```
Float'Fraction (1.0): 5.00000E-01
Float'Fraction (0.25): 5.00000E-01
```

(continues on next page)
Learning Ada

Float'Fraction (1.0e-25): 9.67141E-01
Float'Exponent (1.0): 1
Float'Exponent (0.25): -1
Float'Exponent (1.0e-25): -83
Float'Compose (5.00000e-01, 1): 1.00000E+00
Float'Compose (5.00000e-01, -1): 2.50000E-01
Float'Compose (9.67141E-01, -83): 1.00000E-25

To understand this code example, we have to take this formula into account:

Value = Fraction x Machine_Radix^{Exponent}

Considering that the value of Float'Machine_Radix on a typical PC is two, we see that the value 1.0 is composed by a fraction of 0.5 and a machine exponent of one. In other words:

0.5 x 2^1 = 1.0

For the value 0.25, we get a fraction of 0.5 and a machine exponent of -1, which is the result of 0.5 x 2^{-1} = 0.25. We can use the Compose attribute to perform this calculation. For example, Float'Compose (0.5, -1) = 0.25.

Note that Fraction is always between 0.5 and 0.999999 (i.e < 1.0), except for denormalized numbers, where it can be < 0.5.

**Attribute: Scaling**

Scaling is an attribute that scales a floating-point value based on the machine radix and a machine exponent passed to the function. For example:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Scaling is
begin
  Put_Line ("Float'Scaling (0.25, 1): " & Float'Scaling (0.25, 1)'Image);
  Put_Line ("Float'Scaling (0.25, 2): " & Float'Scaling (0.25, 2)'Image);
  Put_Line ("Float'Scaling (0.25, 3): " & Float'Scaling (0.25, 3)'Image);
end Show_Scaling;
```

**Code block metadata**

MD5: 9fa821d32911b74ee4b4fde3f3adafd8

**Runtime output**

Float'Scaling (0.25, 1): 5.00000E-01
Float'Scaling (0.25, 2): 1.00000E+00
Float'Scaling (0.25, 3): 2.00000E+00

The scaling is calculated with this formula:

scaling = value x Machine_Radix^{machine exponent}

For example, on a typical PC with a machine radix of two, Float'Scaling (0.25, 3) = 2.0 corresponds to

0.25 x 2^3 = 2.0
Round-up and round-down attributes

Floor and Ceiling are attributes that returned the rounded-down or rounded-up value, respectively, of a floating-point value. For example:

Listing 417: show_floor_ceiling.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Floor_Ceiling is
begin
  Put_Line ("Float'Floor (0.25): ", Float'Floor (0.25)'Image);
  Put_Line ("Float'Ceiling (0.25): ", Float'Ceiling (0.25)'Image);
end Show_Floor_Ceiling;
```

Code block metadata

MD5: 1344d54ae86b9fd4831d5f078eb655d4

Runtime output

```
Float'Floor (0.25): 0.00000E+00
Float'Ceiling (0.25): 1.00000E+00
```

As we can see in this example, the rounded-down value (floor) of 0.25 is 0.0, while the rounded-up value (ceiling) of 0.25 is 1.0.

Round-to-nearest attributes

In this section, we discuss three attributes used for rounding: Rounding, Unbiased_Rounding, Machine_Rounding. In all cases, the rounding attributes return the nearest integer value (as a floating-point value). For example, the rounded value for 4.8 is 5.0 because 5 is the closest integer value.

Let's see a code example:

Listing 418: show_roundings.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Roundings is
begin
  Put_Line ("Float'Rounding (0.5): ", Float'Rounding (0.5)'Image);
  Put_Line ("Float'Rounding (1.5): ", Float'Rounding (1.5)'Image);
  Put_Line ("Float'Rounding (4.5): ", Float'Rounding (4.5)'Image);
  Put_Line ("Float'Rounding (-4.5): ", Float'Rounding (-4.5)'Image);
  Put_Line ("Float'Unbiased_Rounding (0.5): ", Float'Unbiased_Rounding (0.5)'Image);
end Show_Roundings;
```

(continues on next page)
The difference between these attributes is the way they handle the case when a value is exactly in between two integer values. For example, 4.5 could be rounded up to 5.0 or rounded down to 4.0. This is the way each rounding attribute works in this case:

- **Rounding** rounds away from zero. Positive floating-point values are rounded up, while negative floating-point values are rounded down when the value is between two integer values. For example:
  - 4.5 is rounded up to 5.0, i.e. `Float'Round`ing (4.5) = `Float'Ceiling` (4.5) = 5.0.
  - -4.5 is rounded down to -5.0, i.e. `Float'Round`ing (-4.5) = `Float'Floor` (-4.5) = -5.0.

- **Unbiased_Rounding** rounds toward the even integer. For example,
  - `Float'Unbiased_Rounding` (0.5) = 0.0 because zero is the closest even integer, while
  - `Float'Unbiased_Rounding` (1.5) = 2.0 because two is the closest even integer.

- **Machine_Rounding** uses the most appropriate rounding instruction available on the target platform. While this rounding attribute can potentially have the best performance, its result may be non-portable. For example, whether the rounding of 4.5 becomes 4.0 or 5.0 depends on the target platform.
  - If an algorithm depends on a specific rounding behavior, it's best to avoid the Machine_Rounding attribute. On the other hand, if the rounding behavior won't have a significant impact on the results, we can safely use this attribute.
Attributes: Truncation, Remainder, Adjacent

The Truncation attribute returns the truncated value of a floating-point value, i.e. the value corresponding to the integer part of a number rounded toward zero. This corresponds to the number before the radix point. For example, the truncation of 1.55 is 1.0 because the integer part of 1.55 is 1.

The Remainder attribute returns the remainder part of a division. For example, \texttt{Float’Remainder (1.25, 0.5) = 0.25}. Let’s briefly discuss the details of this operation. The result of the division 1.25 / 0.5 is 2.5. Here, 1.25 is the dividend and 0.5 is the divisor. The quotient and remainder of this division are 2 and 0.25, respectively. (Here, the quotient is an integer number, and the remainder is the floating-point part that remains.)

Note that the relation between quotient and remainder is defined in such a way that we get the original dividend back when we use the formula: "quotient x divisor + remainder = dividend". For the previous example, this means \(2 \times 0.5 + 0.25 = 1.25\).

The Adjacent attribute is the next machine value towards another value. For example, on a typical PC, the adjacent value of a small value — say, \(1.0 \times 10^{-83}\) — towards zero is +0.0, while the adjacent value of this small value towards 1.0 is another small, but greater value — in fact, it’s \(1.40130 \times 10^{-45}\). Note that the first parameter of the Adjacent attribute is the value we want to analyze and the second parameter is the Towards value.

Let’s see a code example:

```
1 with Ada.Text_IO; use Ada.Text_IO;

2 procedure Show_Truncation_Remainder_Adjacent is
3 begin
4   Put_Line
5      (**Float’Truncation (1.55): "
6         & Float’Truncation (1.55)’Image);
7   Put_Line
8      (**Float’Truncation (-1.55): "
9         & Float’Truncation (-1.55)’Image);
10   Put_Line
11      (**Float’Remainder (1.25, 0.25): "
12         & Float’Remainder (1.25, 0.25)’Image);
13   Put_Line
14      (**Float’Remainder (1.25, 0.5): "
15         & Float’Remainder (1.25, 0.5)’Image);
16   Put_Line
17      (**Float’Remainder (1.25, 1.0): "
18         & Float’Remainder (1.25, 1.0)’Image);
19   Put_Line
20      (**Float’Remainder (1.25, 2.0): "
21         & Float’Remainder (1.25, 2.0)’Image);
22   Put_Line
23      (**Float’Adjacent (1.0e-83, 0.0): "
24         & Float’Adjacent (1.0e-83, 0.0)’Image);
25   Put_Line
26      (**Float’Adjacent (1.0e-83, 1.0): "
27         & Float’Adjacent (1.0e-83, 1.0)’Image);
28 end Show_Truncation_Remainder_Adjacent;
```
Attributes: Copy_Sign and Leading_Part

Copy_Sign is an attribute that returns a value where the sign of the second floating-point argument is multiplied by the magnitude of the first floating-point argument. For example, Float'Copy_Sign (1.0, -10.0) is -1.0. Here, the sign of the second argument (-10.0) is multiplied by the magnitude of the first argument (1.0), so the result is -1.0.

Leading_Part is an attribute that returns the approximated version of the mantissa of a value based on the specified number of leading bits for the mantissa. Let's see some examples:

- Float'Leading_Part (3.1416, 1) is 2.0 because that's the value we can represent with one leading bit.
  - Note that Float'Fraction (2.0) = 0.5 — which can be represented with one leading bit in the mantissa — and Float'Exponent (2.0) = 2.

- If we increase the number of leading bits of the mantissa to two — by writing Float'Leading_Part (3.1416, 2) —, we get 3.0 because that's the value we can represent with two leading bits.

- If we increase again the number of leading bits to five — Float'Leading_Part (3.1416, 5) —, we get 3.125.
  - Note that, in this case Float'Fraction (3.125) = 0.78125 and Float'Exponent (3.125) = 2.
  - The binary mantissa is actually 2#110_0100_0000_0000_0000_0000#, which can be represented with five leading bits as expected: 2#110_01#.
    * We can get the binary mantissa by calculating Float'Fraction (3.125) * Float (Float'Machine_Radix) ** (Float'Machine_Mantissa - 1) and converting the result to binary format. The -1 value in the formula corresponds to the sign bit.

Attention

In this explanation about the Leading_Part attribute, we're talking about leading bits. Strictly speaking, however, this is actually a simplification, and it's only correct if Machine_Radix is equal to two — which is the case for most machines. Therefore, in most cases, the explanation above is perfectly acceptable.

However, if Machine_Radix is not equal to two, we cannot use the term "bits" anymore, but rather digits of the Machine_Radix.

Let's see some examples:

Listing 420: show_copy_sign_leading_part_machine.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Copy_Sign_Leading_Part_Machine is
begin
  Put_Line ("Float'Copy_Sign (1.0, -10.0): "
            & Float'Copy_Sign (1.0, -10.0)'Image);
  Put_Line ("Float'Copy_Sign (-1.0, -10.0): "
            & Float'Copy_Sign (-1.0, -10.0)'Image);
  Put_Line ("Float'Copy_Sign (1.0, 10.0): "
            & Float'Copy_Sign (1.0, 10.0)'Image);
```
Attribute: Machine

Not every real number is directly representable as a floating-point value on a specific machine. For example, let's take a value such as $1.0 \times 10^{15}$ (or $1,000,000,000,000,000$):

Listing 421: show_float_value.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Float_Value is
    package F_IO is new Ada.Text_IO.Float_IO (Float);
    V : Float;
    begin
        F_IO.Default_Fore := 3;
        F_IO.Default_Aft := 1;
        F_IO.Default_Exp := 0;
        V := 1.0E+15;
        put ("1.0E+15 = ");
        F_IO.Put (Item => V);
        New_Line;
    end Show_Float_Value;
```

If we run this example on a typical PC, we see that the expected value $1,000,000,000,000,000.0$ was displayed as $999,999,986,991,000.0$. This is because $1.0 \times 10^{15}$ isn't directly representable on this machine, so it has to be modified to a value that is actually representable (on the machine).

This automatic modification we've just described is actually hidden, so to say, in the assignment. However, we can make it more visible by using the Machine (X) attribute,
which returns a version of X that is representable on the target machine. The Machine (X) attribute rounds (or truncates) X to either one of the adjacent machine numbers for the specific floating-point type of X. (Of course, if the real value of X is directly representable on the target machine, no modification is performed.)

In fact, we could rewrite the V := 1.0E+15 assignment of the code example as V := Float'Machine (1.0E+15), as we're never assigning a real value directly to a floating-pointing variable — instead, we're first converting it to a version of the real value that is representable on the target machine. In this case, 99999986991000.0 is a representable version of the real value 1.0 x 10^{15}. Of course, writing V := 1.0E+15 or V := Float'Machine (1.0E+15) doesn't make any difference to the actual value that is assigned to V (in the case of this specific target architecture), as the conversion to a representable value happens automatically during the assignment to V.

There are, however, instances where using the Machine attribute does make a difference in the result. For example, let's say we want to calculate the difference between the original real value in our example (1.0 x 10^{15}) and the actual value that is assigned to V. We can do this by using the Machine attribute in the calculation:

```
Listing 422: show_machine_attribute.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Machine_Attribute is
  package F_IO is new Ada.Text_IO.Float_IO (Float);
  V : Float;
begin
  F_IO.Default_Fore := 3;
  F_IO.Default_Aft := 1;
  F_IO.Default_Exp := 0;
  Put_Line ("Original value: 1_000_000_000_000_000.0");
  V := 1.0E+15;
  Put ("Machine value: ");
  F_IO.Put (Item => V);
  New_Line;
  V := 1.0E+15 - Float'Machine (1.0E+15);
  Put ("Difference: ");
  F_IO.Put (Item => V);
  New_Line;
end Show_Machine_Attribute;
```

When we run this example on a typical PC, we see that the difference is roughly 1.3009 x 10^7. (Actually, the value that we might see is 1.300896 x 10^7, which is a version of 1.3009 x 10^7 that is representable on the target machine.)
When we write 1.0E+15 - Float'Machine (1.0E+15):

- the first value in the operation is the universal real value 1.0 x 10^{15}, while
- the second value in the operation is a version of the universal real value 1.0 x 10^{15} that is representable on the target machine.

This also means that, in the assignment to V, we're actually writing V := Float'Machine (1.0E+15 - Float'Machine (1.0E+15)), so that:

1. we first get the intermediate real value that represents the difference between these values; and then
2. we get a version of this intermediate real value that is representable on the target machine.

This is the reason why we see 1.3008896 x 10^7 instead of 1.3009 x 10^7 when we run this application.

### 25.7.4 Fixed-Point Types

In this section, we discuss various attributes and operations related to fixed-point types.

#### Attributes of fixed-point types

**Attribute: Machine_Radix**

Machine_Radix is an attribute that returns the radix of the hardware representation of a type. For example:

Listing 423: show_fixed_machine_radix.adb

```ada
with Ada.Text_IO;  use Ada.Text_IO;

procedure Show_Fixed_Machine_Radix is
  type T3_D3 is delta 10.0 ** (-3) digits 3;
  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;
begin
  Put_Line ("T3_D3'Machine_Radix: " & T3_D3'Machine_Radix'Image);
  Put_Line ("TQ31'Machine_Radix: " & TQ31'Machine_Radix'Image);
end Show_Fixed_Machine_Radix;
```

#### Code block metadata


MD5: a90d860a58f76550e948a8245ff5fde

106 http://www.ada-auth.org/standards/22rm/html/RM-3-5-10.html
Learning Ada

Runtime output

<table>
<thead>
<tr>
<th>Type</th>
<th>Machine_Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3_D3</td>
<td>2</td>
</tr>
<tr>
<td>TQ31</td>
<td>2</td>
</tr>
</tbody>
</table>

Usually, this value is two, as the radix is based on a binary system.

**Attribute: Machine_Rounds and Machine_Overflows**

In this section, we discuss attributes that return *Boolean* values indicating whether a feature is available or not in the target architecture:

- **Machine_Rounds** is an attribute that indicates what happens when the result of a fixed-point operation is inexact:
  - T'Machine_Rounds = *True*: inexact result is rounded;
  - T'Machine_Rounds = *False*: inexact result is truncated.

- **Machine_Overflows** is an attribute that indicates whether a *Constraint_Error* is guaranteed to be raised when a fixed-point operation with that type produces an overflow or divide-by-zero.

Listing 424: show_boolean_attributes.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Boolean_Attributes is
  type T3_D3 is delta 10.0 ** (-3) digits 3;
  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;

begin
  Put_Line ("T3_D3'Machine_Rounds: "
            & T3_D3'Machine_Rounds'Image);
  Put_Line ("TQ31'Machine_Rounds: "
            & TQ31'Machine_Rounds'Image);
  Put_Line ("T3_D3'Machine_Overflows: "
            & T3_D3'Machine_Overflows'Image);
  Put_Line ("TQ31'Machine_Overflows: "
            & TQ31'Machine_Overflows'Image);
end Show_Boolean_Attributes;
```

**Attribute: Small and Delta**

The Small and *Delta* attributes return numbers that indicate the numeric precision of a fixed-point type. In many cases, the Small of a type T is equal to the *Delta* of that type — i.e. T'Small = T'Delta. Let’s discuss each attribute and how they distinguish from each other.

The *Delta* attribute returns the value of the *delta* that was used in the type definition. For example, if we declare `type T3_D3 is delta 10.0 ** (-3) digits D`, then the value of T3_D3'Delta is the 10.0 ** (-3) that we used in the type definition.

The Small attribute returns the "small" of a type, i.e. the smallest value used in the machine representation of the type. The small must be at least equal to or smaller than the *delta* — in other words, it must conform to the T'Small <= T'Delta rule.

For further reading...
The Small and the Delta need not actually be small numbers. They can be arbitrarily large. For instance, they could be 1.0, or 1000.0. Consider the following example:

Listing 425: fixed_point defs.ads

```ada
package Fixed_Point_Defs is
  S  : constant := 32;
  Exp : constant := 128;
  D  : constant := 2.0 ** (-S + Exp + 1);

  type Fixed is delta D
    range -1.0 * 2.0 ** Exp ..
          1.0 * 2.0 ** Exp - D;

  pragma Assert (Fixed'Size = S);
end Fixed_Point_Defs;
```

Listing 426: show fixed type info.adb

```ada
with Fixed_PointDefs; use Fixed_PointDefs;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Fixed_Type_Info is
  begin
    Put_Line ("Size : ");
    Put_Line ("Small : ");
    Put_Line ("Delta : ");
    Put_Line ("First : ");
    Put_Line ("Last : ");
  end Show_Fixed_Type_Info;
```

In this example, the small of the Fixed type is actually quite large: \(1.58456325028528675 \times 10^{29}\). (Also, the first and the last values are large: \(-340,282,366,920,938,463,463,374,607,431,768,211,456.0\) and \(340,282,366,762,128,434,845,932,244,680,310,784.0\), or approximately \(-3.4028^{38}\) and \(3.4028^{38}\).)

In this case, if we assign 1 or 1,000 to a variable F of this type, the actual value stored in F is zero. Feel free to try this out!

When we declare an ordinary fixed-point data type, we must specify the delta. Specifying the small, however, is optional:

25.7. Numerics
• If the *small* isn't specified, it is automatically selected by the compiler. In this case, the actual value of the *small* is an implementation-defined power of two — always following the rule that says: \( T'\text{Small} \leq T'\Delta \).

• If we want, however, to specify the *small*, we can do that by using the Small aspect. In this case, it doesn't need to be a power of two.

For decimal fixed-point types, we cannot specify the *small*. In this case, it's automatically selected by the compiler, and it's always equal to the *delta*.

Let's see an example:

Listing 427: fixed_small_delta.ads

```ada
package Fixed_Small_Delta is
  D3 : constant := 10.0 ** (-3);

  type T3_D3 is delta D3 digits 3;
  type TD3 is delta D3 range -1.0 .. 1.0 - D3;
  D31 : constant := 2.0 ** (-31);
  D15 : constant := 2.0 ** (-15);

  type TQ31 is delta D31 range -1.0 .. 1.0 - D31;
  type TQ15 is delta D15 range -1.0 .. 1.0 - D15
    with Small => D31;
end Fixed_Small_Delta;
```

Listing 428: show_fixed_small_delta.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Fixed_Small_Delta; use Fixed_Small_Delta;

procedure Show_Fixed_Small_Delta is
begin
  Put_Line ("T3_D3'Small: " & T3_D3'Small'Image);
  Put_Line ("T3_D3'Delta: " & T3_D3'Delta'Image);
  Put_Line ("T3_D3'Size: " & T3_D3'Size'Image);
  Put_Line ("--------------------");

  Put_Line ("TD3'Small: " & TD3'Small'Image);
  Put_Line ("TD3'Delta: " & TD3'Delta'Image);
  Put_Line ("TD3'Size: " & TD3'Size'Image);
  Put_Line ("--------------------");

  Put_Line ("TQ31'Small: " & TQ31'Small'Image);
  Put_Line ("TQ31'Delta: " & TQ31'Delta'Image);
  Put_Line ("TQ31'Size: " & TQ31'Size'Image);
  Put_Line ("--------------------");

  Put_Line ("TQ15'Small: " & TQ15'Small'Image);
  Put_Line ("TQ15'Delta: " & TQ15'Delta'Image);
  Put_Line ("TQ15'Size: " & TQ15'Size'Image);
  Put_Line ("--------------------");
```

(continues on next page)
As we can see in the output of the code example, the **Delta** attribute returns the value we used for **delta** in the type definition of the T3_D3, TD3, TQ31 and TQ15 types.

The TD3 type is an ordinary fixed-point type with the the same delta as the decimal T3_D3 type. In this case, however, TD3`Small` is not the same as the TD3`Delta`. On a typical desktop PC, TD3`Small` is 2\(^{-10}\), while the delta is 10\(^{-3}\). (Remember that, for ordinary fixed-point types, if we don’t specify the **small**, it’s automatically selected by the compiler as a power of two smaller than or equal to the **delta**.)

In the case of the TQ15 type, we’re specifying the **small** by using the Small aspect. In this case, the underlying size of the TQ15 type is 32 bits, while the precision we get when operating with this type is 16 bits. Let’s see a specific example for this type:

```
with Ada.Text_IO; use Ada.Text_IO;
with Fixed_Small_Delta; use Fixed_Small_Delta;

procedure Show_Fixed_Small_Delta is
  V : TQ15;
begin
  Put_Line ("V'Size: " & V'Size'Image);
  V := TQ15'Small;
  Put_Line ("V: " & V'Image);
  V := TQ15'Delta;
  Put_Line ("V: " & V'Image);
end Show_Fixed_Small_Delta;
```
In the first assignment, we assign `TQ15'Small (2^31)` to `V`. This value is smaller than the type's `delta` \(2^{-15}\). Even though `V'Size` is 32 bits, `V'Delta` indicates 16-bit precision, and `TQ15'Small` requires 32-bit precision to be represented correctly. As a result, `V` has a value of zero after this assignment.

In contrast, after the second assignment — where we assign `TQ15'Delta (2^{-15})` to `V` — we see, as expected, that `V` has the same value as the `delta`.

**Attributes: Fore and Aft**

The Fore and Aft attributes indicate the number of characters or digits needed for displaying a value in decimal representation. To be more precise:

- The Fore attribute refers to the digits before the decimal point and it returns the number of digits plus one for the sign indicator (which is either `-` or space), and it's always at least two.
- The Aft attribute returns the number of decimal digits that is needed to represent the delta after the decimal point.

Let's see an example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Fixed_Fore_Aft is
  type T3_D3 is delta 10.0 ** (-3) digits 3;
  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;
  Dec : constant T3_D3 := -0.123;
  Fix : constant TQ31 := -TQ31'Delta;
begin
  Put_Line ("T3_D3'Fore: " & T3_D3'Fore'Image);
  Put_Line ("T3_D3'Aft: " & T3_D3'Aft'Image);
  Put_Line ("TQ31'Fore: " & TQ31'Fore'Image);
  Put_Line ("TQ31'Aft: " & TQ31'Aft'Image);
  Put_Line ("----");
  Put_Line ("Dec: " & Dec'Image);
  Put_Line ("Fix: " & Fix'Image);
end Show_Fixed_Fore_Aft;
```
As we can see in the output of the Dec and Fix variables at the bottom, the value of Fore is two for both T3_D3 and TQ31. This value corresponds to the length of the string "-0" displayed in the output for these variables (the first two characters of "-0.123" and "-0.0000000005").

The value of Dec'Aft is three, which matches the number of digits after the decimal point in "-0.123". Similarly, the value of Fix'Aft is 10, which matches the number of digits after the decimal point in "-0.0000000005".

**Attributes of decimal fixed-point types**

The attributes presented in this subsection are only available for decimal fixed-point types.

**Attribute: Digits**

**Digits** is an attribute that returns the number of significant decimal digits of a decimal fixed-point subtype. This corresponds to the value that we use for the *digits* in the definition of a decimal fixed-point type.

Let's see an example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Decimal_Digits is
    type T3_D6 is delta 10.0 ** (-3) digits 6;
    subtype T3_D2 is T3_D6 digits 2;
begin
    Put_Line ("T3_D6'Digits: ");
    Put_Line ("& T3_D6'Digits'Image");
    Put_Line ("T3_D2'Digits: ");
    Put_Line ("& T3_D2'Digits'Image");
end Show_Decimal_Digits;
```

25.7. Numerics
In this example, T3_D6′Digits is six, which matches the value that we used for digits in the type definition of T3_D6. The same logic applies for subtypes, as we can see in the value of T3_D2′Digits. Here, the value is two, which was used in the declaration of the T3_D2 subtype.

**Attribute: Scale**

According to the Ada Reference Manual, the Scale attribute "indicates the position of the point relative to the rightmost significant digits of values" of a decimal type. For example:

- If the value of Scale is two, then there are two decimal digits after the decimal point.
- If the value of Scale is negative, that implies that the Delta is a power of 10 greater than 1, and it would be the number of zero digits that every value would end in.

The Scale corresponds to the N used in the delta 10.0 ** (-N) expression of the type declaration. For example, if we write delta 10.0 ** (-3) in the declaration of a type T, then the value of T′Scale is three.

Let's look at this complete example:

`Listing 432: show_decimal_scale.adb`

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Decimal_Scale is
  type TM3_D6 is delta 10.0 ** 3 digits 6;
  type T3_D6 is delta 10.0 ** (-3) digits 6;
  type T9_D12 is delta 10.0 ** (-9) digits 12;

begin
  Put_Line ("TM3_D6'Scale: ", TM3_D6'Image);
  Put_Line ("T3_D6'Scale: ", T3_D6'Image);
  Put_Line ("T9_D12'Scale: ", T9_D12'Image);
end Show_Decimal_Scale;
```

In this example, we get the following values for the scales:

- TM3_D6′Scale = -3,
- T3_D6′Scale = 3,
- T9_D12′Scale = 9.

As you can see, the value of Scale is directly related to the delta of the corresponding type declaration.
**Attribute: Round**

The Round attribute rounds a value of any real type to the nearest value that is a multiple of the *delta* of the decimal fixed-point type, rounding away from zero if exactly between two such multiples.

For example, if we have a type T with three digits, and we use a value with 10 digits after the decimal point in a call to T'Round, the resulting value will have three digits after the decimal point.

Note that the X input of an S'Round (X) call is a universal real value, while the returned value is of S'Base type.

Let's look at this example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Decimal_Round is
  type T3_D3 is delta 10.0 ** (-3) digits 3;
begin
  Put_Line ("T3_D3'Round (0.2774): ", T3_D3'Round (0.2774)'Image);
  Put_Line ("T3_D3'Round (0.2777): ", T3_D3'Round (0.2777)'Image);
end Show_Decimal_Round;
```

Here, the T3_D3 has a precision of three digits. Therefore, to fit this precision, 0.2774 is rounded to 0.277, and 0.2777 is rounded to 0.278.

### 25.7.5 Big Numbers

As we've seen before, we can define numeric types in Ada with a high degree of precision. However, these normal numeric types in Ada are limited to what the underlying hardware actually supports. For example, any signed integer type — whether defined by the language or the user — cannot have a range greater than that of System.Min_Int .. System.Max_Int because those constants reflect the actual hardware's signed integer types. In certain applications, that precision might not be enough, so we have to rely on arbitrary-precision arithmetic\(^{108}\). These so-called "big numbers" are limited conceptually only by available memory, in contrast to the underlying hardware-defined numeric types.

Ada supports two categories of big numbers: big integers and big reals — both are specified in child packages of the Ada.Numerics.Big_Numbers package:

<table>
<thead>
<tr>
<th>Category</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Integers</td>
<td>Ada.Numerics.Big_Numbers.Big_Integers</td>
</tr>
<tr>
<td>Big Reals</td>
<td>Ada.Numerics.Big_Numbers.Big_Real</td>
</tr>
</tbody>
</table>

In the Ada Reference Manual

- Big Numbers\(^{109}\)
- Big Integers\(^{110}\)
- Big Reals\(^{111}\)

Overview

Let’s start with a simple declaration of big numbers:

Listing 434: show_simple_big_numbers.adb

```ada
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
with Ada.Numerics.Big_Numbers.Big_Reals;
use Ada.Numerics.Big_Numbers.Big_Reals;

procedure Show_Simple_Big_Numbers is
  BI : Big_Integer;
  BR : Big_Real;
begin
  BI := 12345678901234567890;
  BR := 2.0 ** 1234;
  Put_Line ("BI: " & BI’Image);
  Put_Line ("BR: " & BR’Image);
  BI := BI + 1;
  BR := BR + 1.0;
  Put_Line ("BI: " & BI’Image);
  Put_Line ("BR: " & BR’Image);
end Show_Simple_Big_Numbers;
```

Code block metadata

MD5: d25e0c73ef04b6c950f2ab63fc96a353

Runtime output

- BI: 12345678901234567890
- BR: 29581122460099629060044695716103590786339687135372992239556207050657350796238924261053837248 37000
- BI: 12345678901234567891
- BR: 29581122460099629060044695716103590786339687135372992239556207050657350796238924261053837248 37000

In this example, we're declaring the big integer BI and the big real BR, and we're incrementing them by one.

Naturally, we're not limited to using the + operator (such as in this example). We can use the same operators on big numbers that we can use with normal numeric types. In fact, the common unary operators (+, -, abs) and binary operators (+, -, *, /, **, Min and Max) are available to us. For example:

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
procedure Show_Simple_Big_Numbers_Operators is
  BI : Big_Integer;
begin
  BI := 12345678901234567890;
  Put_Line ("BI: " & BI'Image);
  BI := -BI + BI / 2;
  BI := BI - BI * 2;
  Put_Line ("BI: " & BI'Image);
end Show_Simple_Big_Numbers_Operators;
```

In this example, we're applying the four basic operators (+, -, *, /) on big integers.

**Factorial**

A typical example is the factorial\(^{112}\): a sequence of the factorial of consecutive small numbers can quickly lead to big numbers. Let's take this implementation as an example:

```ada
function Factorial (N : Integer) return Long_Long_Integer;
```

```ada
function Factorial (N : Integer)
  return Long_Long_Integer is
    Fact := Long_Long_Integer := 1;
begin
  (continues on next page)
```

\(^{112}\) [https://en.wikipedia.org/wiki/Factorial](https://en.wikipedia.org/wiki/Factorial)
Learning Ada

(continued from previous page)

```ada
for I in 2 .. N loop
    Fact := Fact * Long_Long_Integer (I);
end loop;

return Fact;
end Factorial;
```

Listing 438: show_factorial.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Factorial;

procedure Show_Factorial is
begin
    for I in 1 .. 50 loop
        Put_Line (I'Image & "! = " & Factorial (I)'Image);
    end loop;
end Show_Factorial;
```

Code block metadata

MD5: 9b20469533706ef025a03b506a07b920

Runtime output

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000

raised CONSTRAINT_ERROR : factorial.adb:6 overflow check failed

Here, we're using `Long_Long_Integer` for the computation and return type of the Factorial function. (We're using `Long_Long_Integer` because its range is probably the biggest possible on the machine, although that is not necessarily so.) The last number we're able to calculate before getting an exception is $20!$, which basically shows the limitation of standard integers for this kind of algorithm. If we use big integers instead, we can easily display all numbers up to $50!$ (and more!):
Listing 439: factorial.ads

```ada
pragma Ada_2022;

with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;

function Factorial (N : Integer)
return Big_Integer;
```

Listing 440: factorial.adb

```ada
function Factorial (N : Integer)
return Big_Integer is
Fact : Big_Integer := 1;
begin
for I in 2 .. N loop
  Fact := Fact * To_Big_Integer (I);
end loop;
return Fact;
end Factorial;
```

Listing 441: show_big_number_factorial.adb

```ada
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;
with Factorial;

procedure Show_Big_Number_Factorial is
begin
for I in 1 .. 50 loop
  Put_Line (I'Image & "! = 
  & Factorial (I)'Image);
end loop;
end Show_Big_Number_Factorial;
```

Code block metadata

MD5: 18b6e168dac40422a1f0334fe5e4486e

Runtime output

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
```

(continues on next page)
As we can see in this example, replacing the *Long_Long_Integer* type by the *Big_Integer* type fixes the problem (the runtime exception) that we had in the previous version. (Note that we're using the *To_Big_Integer* function to convert from *Integer* to *Big_Integer*: we discuss these conversions next.)

Note that there is a limit to the upper bounds for big integers. However, this limit isn't dependent on the hardware types — as it's the case for normal numeric types —, but rather compiler specific. In other words, the compiler can decide how much memory it wants to use to represent big integers.
Conversions

Most probably, we want to mix big numbers and *standard* numbers (i.e. integer and real numbers) in our application. In this section, we talk about the conversion between big numbers and standard types.

Validity

The package specifications of big numbers include subtypes that ensure that a actual value of a big number is valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Subtype for valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Integers</td>
<td>Valid_Big_Integer</td>
</tr>
<tr>
<td>Big Reals</td>
<td>Valid_Big_Real</td>
</tr>
</tbody>
</table>

These subtypes include a contract for this check. For example, this is the definition of the Valid_Big_Integer subtype:

```ada
subtype Valid_Big_Integer is Big_Integer
  with Dynamic_Predicate =>
    Is_Valid (Valid_Big_Integer),
  Predicate_Failure =>
    (raise Program_Error);
```

Any operation on big numbers is actually performing this validity check (via a call to the Is_Valid function). For example, this is the addition operator for big integers:

```ada
function "+" (L, R : Valid_Big_Integer)
return Valid_Big_Integer;
```

As we can see, both the input values to the operator as well as the return value are expected to be valid — the Valid_Big_Integer subtype triggers this check, so to say. This approach ensures that an algorithm operating on big numbers won't be using invalid values.

Conversion functions

These are the most important functions to convert between big number and *standard* types:

<table>
<thead>
<tr>
<th>Category</th>
<th>To big number</th>
<th>From big number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Integers</td>
<td>• To_Big_Integer</td>
<td>• To_Integer (Integer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• From_Big_Integer (other integer types)</td>
</tr>
<tr>
<td>Big Reals</td>
<td>• To_Big_Real (floating-point types or fixed-point types)</td>
<td>• From_Big_Real</td>
</tr>
<tr>
<td></td>
<td>• To_Big_Real (Valid_Big_Integer)</td>
<td>• Numerator,Denominator (Integer)</td>
</tr>
<tr>
<td></td>
<td>• To_Real (Integer)</td>
<td></td>
</tr>
</tbody>
</table>
In the following sections, we discuss these functions in more detail.

**Big integer to integer**

We use the `To_Big_Integer` and `To_Integer` functions to convert back and forth between `Big_Integer` and `Integer` types:

```
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;

with Ada.Numerics.Big_Numbers.Big_Integers;

procedure Show_Simple_Big_Integer_Conversion is
  BI : Big_Integer;
  I : Integer := 10000;
begin
  BI := To_Big_Integer (I);
  Put_Line ("BI: " & BI'Image);
  I := To_Integer (BI + 1);
  Put_Line ("I: " & I'Image);
end Show_Simple_Big_Integer_Conversion;
```

Code block metadata

MD5: 84f55568b26bf6c61c6f0b06391e8ac0f

Runtime output

BI: 10000
I: 10001

In addition, we can use the generic `Signed_Conversions` and `Unsigned_Conversions` packages to convert between `Big_Integer` and any signed or unsigned integer types:

```
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;

with Ada.Numerics.Big_Numbers.Big_Integers;

procedure Show_Arbitrary_Big_Integer_Conversion is
  type Mod_32_Bit is mod 2 ** 32;

  package Long_Long_Integer_Conversions is new
    Signed_Conversions (Long_Long_Integer);
  use Long_Long_Integer_Conversions;

  package Mod_32_Bit_Conversions is new
    Unsigned_Conversions (Mod_32_Bit);
  use Mod_32_Bit_Conversions;
```

(continues on next page)
In this examples, we declare the **Long_Long_Integer_Conversions** and the **Mod_32_Bit_Conversions** to be able to convert between big integers and the **Long_Long_Integer** and the **Mod_32_Bit** types, respectively.

Note that, when converting from big integer to integer, we used the **To_Integer** function, while, when using the instances of the generic packages, the function is named **From_Big_Integer**.

**Big real to floating-point types**

When converting between big real and floating-point types, we have to instantiate the generic **Float_Conversions** package:
package D10_Conversions is new
  Float_Conversions (D10);
use D10_Conversions;

package Long_Float_Conversions is new
  Float_Conversions (Long_Float);
use Long_Float_Conversions;

BR : Big_Real;
LF : Long_Float := 2.0;
F10 : D10 := 1.999;

begin
  BR := To_Big_Real (LF);
  Put_Line ("BR: " & BR'String);
  LF := From_Big_Real (BR + 1.0);
  Put_Line ("LF: " & LF'String);
  BR := To_Big_Real (F10);
  Put_Line ("BR: " & BR'String);
  F10 := From_Big_Real (BR + 0.1);
  Put_Line ("F10: " & F10'String);
end Show_Big_Real_Floating_Point_Conversion;

In this example, we declare the D10_Conversions and the Long_Float_Conversions to
be able to convert between big reals and the custom floating-point type D10 and the
Long_Float type, respectively. To do that, we use the To_Big_Real and the From_Big_Real
functions.

Big real to fixed-point types

When converting between big real and ordinary fixed-point types, we have to instantiate
the generic Fixed_Conversions package:
procedure Show_Big_Real_Fixed_Point_Conversion is
  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;
  package TQ31_Conversions is new
    Fixed_Conversions (TQ31);
  use TQ31_Conversions;
  BR : Big_Real;
  FQ31 : TQ31 := 0.25;
begin
  BR := To_Big_Real (FQ31);
  Put_Line ("BR: " & BR'Image);
  FQ31 := From_Big_Real (BR * 2.0);
  Put_Line ("FQ31: " & FQ31'Image);
end Show_Big_Real_Fixed_Point_Conversion;

In this example, we declare the TQ31_Conversions to be able to convert between big reals and the custom fixed-point type TQ31 type. Again, we use the To_Big_Real and the From_Big_Real functions for the conversions.

Note that there's no direct way to convert between decimal fixed-point types and big real types. (Of course, you could perform this conversion indirectly by using a floating-point or an ordinary fixed-point type in between.)

Big reals to (big) integers

We can also convert between big reals and big integers (or standard integers):

Listing 446: show_big_real_big_integer_conversion.adb

pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
with Ada.Numerics.Big_Numbers.Big_Reals;
use Ada.Numerics.Big_Numbers.Big_Reals;
procedure Show_Big_Real_Big_Integer_Conversion is
  I : Integer;
Learning Ada

(continued from previous page)

begin
I := 12345;
BR := To_Real (I);
Put_Line ("BR (from I): " & BR'Image);

BI := 123456;
BR := To_Big_Real (BI);
Put_Line ("BR (from BI): " & BR'Image);
end Show_Big_Real_Big_Integer_Conversion;

Code block metadata

MD5: 9a217c0551bc80269596d7217d2be879

Runtime output

BR (from I): 12345.000
BR (from BI): 123456.000

Here, we use the To_Real and the To_Big_Real and functions for the conversions.

String conversions

In addition to that, we can use string conversions:

Listing 447: show_big_number_string_conversion.adb

pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
with Ada.Numerics.Big_Numbers.Big_Reals;
use Ada.Numerics.Big_Numbers.Big_Reals;
procedure Show_Big_Number_String_Conversion
is
BI : Big_Integer;
BR : Big_Real;
begin
BI := From_String ("12345678901234567890");
BR := From_String ("12345678901234567890.0");
Put_Line ("BI: 
& To_String (Arg => BI,
Width => 5,
Base => 2));
Put_Line ("BR: 
& To_String (Arg => BR,
Fore => 2,
Aft => 6,
(continues on next page)
In this example, we use the From_String to convert a string to a big number. Note that the From_String function is actually called when converting a literal — because of the corresponding aspect for user-defined literals in the definitions of the Big_Integer and the Big_Real types.

For further reading...

Big numbers are implemented using user-defined literals (page 339), which we discussed previously. In fact, these are the corresponding type declarations:

```
-- Declaration from
-- Ada.Numerics.Big_Numbers.Big_Integers;

type Big_Integer is private
  with Integer_Literal => From_Universal_Image,
                   Put_Image => Put_Image;

function From_Universal_Image
  (Arg : String)
  return Valid_Big_Integer
renames From_String;

-- Declaration from
-- Ada.Numerics.Big_Numbers.Big_Reals;

type Big_Real is private
  with Real_Literal => From_Universal_Image,
                 Put_Image => Put_Image;

function From_Universal_Image
  (Arg : String)
  return Valid_Big_Real
renames From_String;
```

As we can see in these declarations, the From_String function renames the From_Universal_Image function, which is being used for the user-defined literals.

Also, we call the To_String function to get a string for the big numbers. Naturally, using the To_String function instead of the Image attribute — as we did in previous examples — allows us to customize the format of the string that we display in the user message.
**Other features of big integers**

Now, let's look at two additional features of big integers:
- the natural and positive subtypes, and
- other available operators and functions.

**Big positive and natural subtypes**

Similar to integer types, big integers have the Big_Natural and Big_Positive subtypes to indicate natural and positive numbers. However, in contrast to the Natural and Positive subtypes, the Big_Natural and Big_Positive subtypes are defined via predicates rather than the simple ranges of normal (ordinary) numeric types:

```ada
subtype Natural is
  Integer range 0 .. Integer'Last;

subtype Positive is
  Integer range 1 .. Integer'Last;

subtype Big_Natural is Big_Integer
  with Dynamic_Predicate =>
    (if Is_Valid (Big_Natural)
      then Big_Natural >= 0),
    Predicate_Failure =>
      (raise Constraint_Error);

subtype Big_Positive is Big_Integer
  with Dynamic_Predicate =>
    (if Is_Valid (Big_Positive)
      then Big_Positive > 0),
    Predicate_Failure =>
      (raise Constraint_Error);
```

Therefore, we cannot simply use attributes such as Big_Natural'First. However, we can use the subtypes to ensure that a big integer is in the expected (natural or positive) range:

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;

procedure Show_Big_Positive_Natural is
  BI, D, N : Big_Integer;
begin
  D := 3;
  N := 2;
  BI := Big_Natural (D / Big_Positive (N));
  Put_Line ("BI: " & BI'Image);
end Show_Big_Positive_Natural;
```

**Code block metadata**


MD5: 6debfb86e11c7bfa3dbaf2d81eb24360
### Runtime output

BI: 1

By using the Big_Natural and Big_Positive subtypes in the calculation above (in the assignment to BI), we ensure that we don't perform a division by zero, and that the result of the calculation is a natural number.

### Other operators for big integers

We can use the **mod** and **rem** operators with big integers:

#### Listing 449: show_big_integer_rem_mod.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
procedure Show_Big_Integer_Rem_Mod is
  BI : Big_Integer;
begín
  BI := 145 mod (-4);
  Put_Line ("BI (mod): " & BI'Image);
  BI := 145 rem (-4);
  Put_Line ("BI (rem): " & BI'Image);
end Show_Big_Integer_Rem_Mod;
```

In this example, we use the **mod** and **rem** operators in the assignments to BI.

Moreover, there's a **Greatest_Common_Divisor** function for big integers which, as the name suggests, calculates the greatest common divisor of two big integer values:

#### Listing 450: show_big_integer_greatest_common_divisor.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
procedure Show_Big_Integer_Greatest_Common_Divisor is
  BI : Big_Integer;
begín
  BI := Greatest_Common_Divisor (145, 25);
  Put_Line ("BI: " & BI'Image);
end Show_Big_Integer_Greatest_Common_Divisor;
```

(continues on next page)
end Show_Big_Integer_Greatest_Common_Divisor;

Code block metadata

MD5: b2d0098fcca6f949f228276b4d862b56

Runtime output

BI: 5

In this example, we retrieve the greatest common divisor of 145 and 25 (i.e.: 5).

Big real and quotients

An interesting feature of big reals is that they support quotients. In fact, we can simply assign 2/3 to a big real variable. (Note that we're able to omit the decimal points, as we write 2/3 instead of 2.0 / 3.0.) For example:

Listing 451: show_big_real_quotient_conversion.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Big_Real_Quotient_Conversion is
   BR : Big_Real;
begin
   BR := 2 / 3;
   -- Same as:
   -- BR := From_Quotient_String ("2 / 3");
   Put_Line ("BR: " & BR'Image);
   Put_Line ("Q: "
      & To_Quotient_String (BR));
   Put_Line ("Q numerator: "
      & Numerator (BR)'Image);
   Put_Line ("Q denominator: "
      & Denominator (BR)'Image);
end Show_Big_Real_Quotient_Conversion;
```

Code block metadata

MD5: 4ef8355332e73a1f7da036b8e1e4b898

Runtime output

BR: 0.666
Q: 2 / 3
In this example, we assign \( \frac{2}{3} \) to \( BR \) — we could have used the \text{From\_Quotient\_String} function as well. Also, we use the \text{To\_Quotient\_String} to get a string that represents the quotient. Finally, we use the \text{Numerator} and \text{Denominator} functions to retrieve the values, respectively, of the numerator and denominator of the quotient (as big integers) of the big real variable.

**Range checks**

Previously, we've talked about the \text{Big\_Natural} and \text{Big\_Positive} subtypes. In addition to those subtypes, we have the \text{In\_Range} function for big numbers:

```
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
with Ada.Numerics.Big_Numbers.Big_Reals;
use Ada.Numerics.Big_Numbers.Big_Reals;
procedure Show_Big_Numbers_In_Range is
  BI : Big_Integer;
  BR : Big_Real;
  BI_From : constant Big_Integer := 0;
  BI_To : constant Big_Integer := 1024;
  BR_From : constant Big_Real := 0.0;
  BR_To : constant Big_Real := 1024.0;
begin
  BI := 1023;
  BR := 1023.9;
  if In_Range (BI, BI_From, BI_To) then
    Put_Line ("BI (" & BI'Image & ") is in the " & BI_From'Image & ". . . " & BI_To'Image & ") range");
  end if;
  if In_Range (BR, BR_From, BR_To) then
    Put_Line ("BR (" & BR'Image & ") is in the " & BR_From'Image & ". . . " & BR_To'Image & ") range");
```

(continues on next page)
In this example, we call the `In_Range` function to check whether the big integer number (BI) and the big real number (BR) are in the range between 0 and 1024.
26.1 Expressions

26.1.1 Expressions: Definition

According to the Ada Reference Manual, an expression "is a formula that defines the computation or retrieval of a value." Also, when an expression is evaluated, the computed or retrieved value always has an associated type known at compile-time.

Even though the definition above is very simple, Ada expressions are actually very flexible — and they can also be very complex. In fact, if you read the corresponding section of the Ada Reference Manual, you'll quickly discover that they include elements such as relations, membership choices, terms and primaries. Some of these are classic elements of expressions in programming languages, although some of their forms are unique to Ada. In this section, we present examples of just some of these elements. For a complete overview, please refer to the Reference Manual.

In the Ada Reference Manual
- 4.4 Expressions

Relations and simple expressions

Expressions usually consist of relations, which in turn consist of simple expressions. (There are more details to this, but we'll keep it simple for the moment.) Let's see a code example with a few expressions, which we dissect into the corresponding grammatical elements (we're going to discuss them later):

```
procedure Show_Expression_Elements is
  type Mode is (Off, A, B, C, D);
  pragma Unreferenced (B, C, D);
  subtype Active_Mode is Mode
    range Mode' Succ (Off) .. Mode' Last;
  M1, M2 : Mode;
  Dummy : Boolean;
begin
```
M1 := A;

Dummy :=
M1 in Active_Mode
and then M2 in Off | A;

-- ^^^^^^^^^^^^^^ relation
-- ^^^^^^^^^^^^^ relation
-- ^^^^^^^^^^^^^^^^^ expression

Dummy :=
M1 in Active_Mode;

-- ^^ name
-- ^^ primary
-- ^^ factor
-- ^^ term
-- ^^ simple expression
-- ^^ membership choice
-- ^^ membership choice list
-- ^ expression

Dummy :=
M2 in Off | A;

-- ^^ name
-- ^^ primary
-- ^^ factor
-- ^^ term
-- ^^ simple expression
-- ^^ membership choice
-- ^ membership choice
-- ^^^^^ membership choice list
-- ^ expression

end Show_Expression_Elements;

Code block metadata

Expression_Elements
MD5: a22e6f2d2bc181ce77097a1de204eb62

Build output

show_expression_elements.adb:9:08: warning: variable "M2" is read but never assigned [-gnatwv]

In this code example, we see three expressions. As we mentioned earlier, every expression has a type; here, the type of each expression is Boolean.

The first expression (M1 in Active_Mode and then M2 in Off | A) consists of two relations: M1 in Active_Mode and M2 in Off | A. Let's discuss some of the details.

The M1 in Active_Mode relation consists of the simple expression M1 and the membership choice list Active_Mode. (Here, the in keyword is part of the relation definition.) Also, as
we see in the comments of the source code, the simple expression $M_1$ is, at the same time, a term, a factor, a primary and a name.

Let's briefly talk about this chain of syntactic elements for simple expressions. Very roughly said, this is how we can break up simple expressions:

- a simple expression consists of terms;
- a term consists of factors;
- a factor consists of primaries;
- a primary can be one of those:
  - a numeric literal;
  - $\texttt{null}$;
  - a string literal;
  - an aggregate (page 435);
  - a name;
  - an allocator (like $\texttt{new Integer}$);
  - a parenthesized expression (page 585);
  - a conditional expression (page 588);
  - a quantified expression (page 590);
  - a declare expression (page 594).

For further reading...
The definition of simple expressions we've just seen is very simplified. In actuality, these are the grammatical elements specified in the Ada Reference Manual:

```ada
simple_expression ::=  
  [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=  
  numeric_literal | null | string_literal | aggregate 
  | name | allocator | (expression) 
  | (conditional_expression) | (quantified_expression) 
  | (declare_expression)
```

Later on in this chapter, we discuss conditional expressions (page 588), quantified expressions (page 590) and declare expressions (page 594) in more details.

In the relation $M_2 \texttt{in Off | A}$ from the code example, Off \texttt{| A} is a membership choice list, and Off and A are membership choices.

For further reading...
Relations can actually be much more complicated than the one we just saw. In fact, this is the definition from the Ada Reference Manual:

```ada
expression ::=  
  relation {and relation} 
  | relation {and then relation}
```

(continues on next page)
Learning Ada

(continued from previous page)

| relation {or relation} |
| relation {or else relation} |
| relation {xor relation} |

relation ::=  
simple_expression  
[relational_operator simple_expression]  
| simple_expression [not] in  
membership_choice_list  
| raise_expression

Again, for more details, please refer to the section on expressions\[^{115}\] of the Ada Reference Manual.

In the Ada Reference Manual

- 4.4 Expressions\[^{116}\]
- 4.5.2 Relational Operators and Membership Tests\[^{117}\]

**Numeric expressions**

The expressions we’ve seen so far had the Boolean type. Although much of the grammar described in the Manual exists exclusively for Boolean operations, we can also write numeric expressions such as the following one:

```
procedure Show_Numeric_Expressions is
  C1 : constant Integer := 5;
  Dummy : Integer;
begin
  Dummy := -2 ** 4 + 3 * C1 ** 8;
```

```
| -^ numeric literal  |
| -- primary          |
| -- ^^ name           |
| -- ^^ primary        |
| -- ^^^^^^^ factor    |
| -- ^ multiplying operator |
| -- ^ numeric literal |
| -- ^ primary        |
| -- ^ factor         |
| -- ^^^^^^^^^^^ term |
| -- ^ numeric literal |
| -- ^ primary        |
| -- ^ numeric literal |
| -- ^ primary        |
| -- ^^^^^ factor     |
| -- ^ binary adding operator |
| -- ^ unary adding operator |
```

(continues on next page)

\[^{115}\] http://www.ada-auth.org/standards/22rm/html/RM-4-4.html
\[^{117}\] http://www.ada-auth.org/standards/22rm/html/RM-4-5-2.html
In this code example, the expression \(- 2 \times 4 + 3 \times C1 \times 8\) consists of just a single simple expression. (Note that simple expressions do not have to be "simple"). This simple expression consists of two terms: \(2 \times 4\) and \(3 \times C1 \times 8\). While the \(2 \times 4\) term is also a single factor, the \(3 \times C1 \times 8\) term consists of two factors: \(3\) and \(C1 \times 8\). Both the \(2 \times 4\) and the \(C1 \times 8\) factors consist of two primaries each:

- the \(2 \times 4\) factor has the primaries 2 and 4,
- the \(C1 \times 8\) factor has the primaries C1 and 8.

### In the Ada Reference Manual

- 4.4 Expressions\(^ {118}\)

### Other expressions

Expressions aren't limited to the **Boolean** type or to numeric types. Indeed, expressions can be of any type, and the definition of primaries we've seen earlier on already hints in this direction — as it includes elements such as allocators. Because expressions are very flexible, covering all possible variations and combinations in this section is out of scope. Again, please refer to the section on expressions\(^ {119}\) of the Ada Reference Manual for further details.

### Parenthesized expression

An interesting aspect of primaries is that, by using parentheses, we can embed an expression inside another expression. As an example, let's discuss the following expression and its elements:

```
procedure Show_Parenthesized_Expressions is
  C1 : constant Integer := 4;
  C2 : constant Integer := 5;
begin
  Dummy : Integer;
  begin
    Dummy :=
      (2 + C1) \times C2;
    -- ^^ name
    -- ^^ primary
  end
end Show_Parenthesized_Expressions;
```


In this example, we first start with the single expression \((2 + C1) \ast C2\), which is also a simple expression consisting of just one term, which consists of two factors: \((2 + C1)\) and \(C2\). The \((2 + C1)\) factor is also a primary. Now, because of the parentheses, we identify that the primary \((2 + C1)\) is an expression that is embedded in another expression.

**Important**

To be fair, the existence of parentheses in a primary could also indicate other kinds of expressions, such as conditional or quantified expressions. However, differentiating between them is straightforward, as we'll see later on in this chapter.

We then proceed to parse the \((2 + C1)\) expression, which consists of the terms \(2\) and \(C1\). As we've seen in the comments of the code example, each of these terms consists of one factor, which consists of one primary. In the end, after parsing the primaries, we identify that \(2\) is a numeric literal and \(C1\) is a name.

Note that the usage of parentheses might lead to situations where we have expressions in potentially unsuspected places. For example, consider the following code example:
Here, the case statement expects a selecting expression. In this case, \( M_1 \) is identified as a name — after being identified as a relation, a simple expression, a term, a factor and a primary.

However, if we replace `case M1` is by `case (M1)` is, (M1) is identified as a parenthesized expression, not as a name! This parenthesized expression is first parsed and evaluated, which might have implications in case statements, as we'll see in another chapter (page 612).

Let's look at another example, this time with a subprogram call:

```
Listing 5: increment_by_one.ads

procedure Increment_By_One (I : in out Integer);
```

```
Listing 6: increment_by_one.adb

procedure Increment_By_One (I : in out Integer) is
  begin
    I := I + 1;
  end Increment_By_One;
```

```
Listing 7: show_name_in_expression.adb

with Increment_By_One;

procedure Show_Name_In_Expression is
  V : Integer := 0;
  begin
    Increment_By_One ((V));
  end Show_Name_In_Expression;
```

The `Increment_By_One` procedure from this example expects a variable as an actual parameter because the parameter mode is `in out`. However, the \((V)\) in the call to the procedure is interpreted as an expression, so we end up providing a value — the result of the expression — as the actual parameter instead of the \( V \) variable. Naturally, this is a compilation error. (Of course, writing `Increment_By_One (V)` fixes the error.)
26.1.2 Conditional Expressions

As we've seen before, we can write simple expressions such as \( I = 0 \) or \( D.\text{Valid} \). A conditional expression, as the name implies, is an expression that contains a condition. This might be an "if-expression" (in the \texttt{if ... then ... else} form) or a "case-expression" (in the \texttt{case ... is when} \Rightarrow form).

The \texttt{Max} function in the following code example is an expression function implemented with a conditional expression — an if-expression, to be more precise:

```
package Expr_Func is
  function Max (A, B : Integer) return Integer is
    (if A \geq B then A else B);
end Expr_Func;
```

Let's say we have a system with four states Off, On, Waiting, and Invalid. For this system, we want to implement a function named Toggled that returns the toggled value of a state \( S \). If the current value of \( S \) is either Off or On, the function toggles from Off to On (or from On to Off). For other values, the state remains unchanged — i.e. the returned value is the same as the input value. This is the implementation using a conditional expression:

```
package Expr_Func is
  type State is (Off, On, Waiting, Invalid);
  function Toggled (S : State) return State is
    (if S = Off then On
    elsif S = On then Off
    else S);
end Expr_Func;
```

As you can see, if-expressions may contain an \texttt{elsif} branch (and therefore be more complicated).

The code above corresponds to this more verbose version:

```
package Expr_Func is
  type State is (Off, On, Waiting, Invalid);
  function Toggled (S : State) return State;
end Expr_Func;
```
package body Expr_Func is
    function Toggled (S : State) return State is
        begin
            if S = Off then
                return On;
            elsif S = On then
                return Off;
            else
                return S;
            end if;
        end Toggled;
end Expr_Func;

Code block metadata
  Conditional_If_Expressions_2
MD5: 9e6cdf53c9c934f37e5717e1d230615a

If we compare the if-block of this code example to the if-expression of the previous example, we notice that the if-expression is just a simplified version without the return keyword and the end if;. In fact, converting an if-block to an if-expression is quite straightforward.

We could also replace the if-expression used in the Toggled function above with a case-expression. For example:

package Expr_Func is
    type State is (Off, On, Waiting, Invalid);
    function Toggled (S : State) return State is
        (case S is
            when Off => On,
            when On => Off,
            when others => S);
end Expr_Func;

Code block metadata
  Conditional_Case_Expressions_1
MD5: 0dd3a86f0872d1e0c3a81f7a17c44bd5

Note that we use commas in case-expressions to separate the alternatives (the when expressions). The code above corresponds to this more verbose version:
Learning Ada

Listing 14: expr_func.adb

```ada
package body Expr_Func is

  function Toggled (S : State) return State is
  begin
    case S is
      when Off => return On;
      when On  => return Off;
      when others => return S;
    end case;
  end Toggled;

end Expr_Func;
```

Code block metadata

.Conditional_Case_Expressions_2
MD5: db6a0737e3931c83c31f53e4da3d8a2b

If we compare the case block of this code example to the case-expression of the previous example, we notice that the case-expression is just a simplified version of the case block without the `return` keyword and the `end case;`, and with alternatives separated by commas instead of semicolons.

In the Ada Reference Manual

- 4.5.7 Conditional Expressions

26.1.3 Quantified Expressions

Quantified expressions are *for* expressions using a quantifier — which can be either *all* or *some* — and a predicate. This kind of expressions let us formalize statements such as:

- "all values of array \( A \) must be zero" into *for all* \( I \) in \( A'\text{Range} \) \( \Rightarrow \) \( A \ (I) = 0 \), and
- "at least one value of array \( A \) must be zero" into *for some* \( I \) in \( A'\text{Range} \) \( \Rightarrow \) \( A \ (I) = 0 \).

In the quantified expression *for all* \( I \) in \( A'\text{Range} \) \( \Rightarrow \) \( A \ (I) = 0 \), the quantifier is *all* and the predicate is \( A \ (I) = 0 \). In the second expression, the quantifier is *some*. The result of a quantified expression is always a Boolean value.

For example, we could use the quantified expressions above and implement these two functions:

- Is_Zero, which checks whether all components of an array \( A \) are zero, and
- Has_Zero, which checks whether array \( A \) has at least one component of the array \( A \) is zero.

This is the complete code:

package Int_Arrays is

  type Integer_Arr is
  array (Positive range <>) of Integer;

  function Is_Zero (A : Integer_Arr)
  return Boolean is
  (for all I in A'Range => A (I) = 0);

  function Has_Zero (A : Integer_Arr)
  return Boolean is
  (for some I in A'Range => A (I) = 0);

  procedure Display_Array (A : Integer_Arr;
          Name : String);

end Int_Arrays;

with Ada.Text_IO; use Ada.Text_IO;

package body Int_Arrays is

  procedure Display_Array (A : Integer_Arr;
          Name : String) is
  begin
    Put (Name & ": ");
    for E of A loop
        Put (E'Image & " ");
    end loop;
    New_Line;
  end Display_Array;

end Int_Arrays;

with Ada.Text_IO; use Ada.Text_IO;

with Int_Arrays; use Int_Arrays;

procedure Test_Int_Arrays is
  A : Integer_Arr := (0, 0, 1);
  begin
    Display_Array (A, "A");
    Put_Line ("Is Zero: "
               & Boolean'Image (Is_Zero (A)));
    Put_Line ("Has Zero: "
              & Boolean'Image (Has_Zero (A)));
    A := (0, 0, 0);
    Display_Array (A, "A");
    Put_Line ("Is Zero: "
              & Boolean'Image (Is_Zero (A)));
    Put_Line ("Has Zero: "
              & Boolean'Image (Has_Zero (A)));
  end Test_Int_Arrays;
As you might have expected, we can rewrite a quantified expression as a loop in the \texttt{for I in A'Range loop if ... return ...} form. In the code below, we're implementing \texttt{Is_Zero} and \texttt{Has_Zero} using loops and conditions instead of quantified expressions:

\begin{verbatim}
package Int_Arrays is
  type Integer_Arr is
    array (Positive range <>) of Integer;
  function Is_Zero (A : Integer_Arr) return Boolean;
  function Has_Zero (A : Integer_Arr) return Boolean;
  procedure Display_Array (A : Integer_Arr;
                           Name : String);
end Int_Arrays;
\end{verbatim}

\begin{verbatim}
with Ada.Text_IO; use Ada.Text_IO;
package body Int_Arrays is
  function Is_Zero (A : Integer_Arr) return Boolean is
    begin
      for I in A'Range loop
        if A (I) /= 0 then
          return False;
        end if;
      end loop;
      return True;
    end Is_Zero;

  function Has_Zero (A : Integer_Arr) return Boolean is
    begin
      for I in A'Range loop
        if A (I) = 0 then
          return True;
        end if;
      end loop;
      return True;
    end Has_Zero;
end Int_Arrays;
\end{verbatim}
end loop;

return False;
end Has_Zero;

procedure Display_Array (A : Integer_Arr;
                          Name : String) is
begin
  Put (Name & ": ");
  for E of A loop
    Put (E’Image & " ");
  end loop;
  New_Line;
end Display_Array;
end Int_Arrays;

Listing 20: test_int_arrays.adb

with Ada.Text_IO; use Ada.Text_IO;
with Int_Arrays; use Int_Arrays;

procedure Test_Int_Arrays is
  A : Integer_Arr := (0, 0, 1);
begin
  Display_Array (A, "A");
  Put_Line ("Is_Zero: "
            & Boolean’Image (Is_Zero (A)));
  Put_Line ("Has_Zero: "
            & Boolean’Image (Has_Zero (A)));
  A := (0, 0, 0);
  Display_Array (A, "A");
  Put_Line ("Is_Zero: "
            & Boolean’Image (Is_Zero (A)));
  Put_Line ("Has_Zero: "
            & Boolean’Image (Has_Zero (A)));
end Test_Int_Arrays;

Code block metadata

Quantified Expression_2
MD5: a957a8fd60e1849248e6e1a84ee6a6a

Runtime output

A: 0 0 1
Is_Zero: FALSE
Has_Zero: TRUE
A: 0 0 0
Is_Zero: TRUE
Has_Zero: TRUE

So far, we’ve seen quantified expressions using indices — e.g. for all I in A’Range => .... We could avoid indices in quantified expressions by simply using the E of A form. In this case, we can just write for all E of A => .... Let’s adapt the implementation of Is_Zero and Has_Zero using this form:
package Int_Arrays is

  type Integer_Arr is array (Positive range <> ) of Integer;

  function Is_Zero (A : Integer_Arr) return Boolean is
    (for all E of A => E = 0);

  function Has_Zero (A : Integer_Arr) return Boolean is
    (for some E of A => E = 0);

end Int_Arrays;

Here, we’re checking the components E of the array A and comparing them against zero.

In the Ada Reference Manual
  • 4.5.8 Quantified Expressions

26.1.4 Declare Expressions

So far, we’ve seen expressions that make use of existing objects declared outside of the expression. Sometimes, we might want to declare constant objects inside the expression, so we can use them locally in the expression. Similarly, we might want to rename an object and use the renamed object in an expression. In those cases, we can use a declare expression.

A declare expression allows for declaring or renaming objects within an expression:

Listing 22: p.ads

pragma Ada_2022;

package P is

  function Max (A, B : Integer) return Integer is
    (declare
      Bigger_A : constant Boolean := (A >= B);
      begin
        (if Bigger_A then A else B));

end P;

The declare expression starts with the **declare** keyword and the usual object declarations, and it's followed by the **begin** keyword and the body. In this example, the body of the declare expression is a conditional expression.

Of course, the code above isn't really useful, so let's look at a more complete example:

Listing 23: integer_arrays.ads

```ada
pragma Ada_2022;
package Integer_Arrays is
  type Integer_Array is array (Positive range <>) of Integer;
  function Sum (Arr : Integer_Array) return Integer is
    begin
    end Integer_Arrays;
```

Listing 24: integer_arrays.adb

```ada
package body Integer_Arrays is
  function Sum (Arr : Integer_Array) return Integer is
    begin
    end Integer_Arrays;
```

Listing 25: show_integer_arrays.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Integer_Arrays; use Integer_Arrays;
```

(continues on next page)
procedure Show_Integer_Arrays is
    Arr : constant Integer_Array := [1, 2, 3];
begin
    Put_Line ("Sum: " & Sum (Arr)'Image);
    Put_Line ("Avg: " & Avg (Arr)'Image);
end Show_Integer_Arrays;

Code block metadata

MD5: 8e96d49b1676f0aaf95437e271060690

Runtime output

Sum: 6
Avg: 2.00000E+00

In this example, the Avg function is implemented using a declare expression. In this expression, A renames the Arr array, and S is a constant initialized with the value returned by the Sum function.

In the Ada Reference Manual

- 4.5.9 Declare Expressions

Restrictions in the declarative part

The declarative part of a declare expression is more restricted than the declarative part of a subprogram or declare block. In fact, we cannot:

- declare variables;
- declare constants of limited types;
- rename an object of limited type that is constructed within the declarative part;
- declare aliased constants;
- declare constants that make use of the Access or Unchecked_Access attributes in the initialization;
- declare constants of anonymous access type.

Let's see some examples of erroneous declarations:

Listing 26: integer_arrays.ads

pragma Ada_2022;

package Integer_Arrays is

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Integer_Sum is limited private;

(continues on next page)

type Const_Integer_Access is
access constant Integer;

function Sum (Arr : Integer_Array)
return Integer;

function Sum (Arr : Integer_Array)
return Integer_Sum;

-- Expression function using
-- declare expression:
--
function Avg (Arr : Integer_Array)
return Float is
(declare
A : Integer_Array renames Arr;

S1 : aliased constant Integer := Sum (A);
-- ERROR: aliased constant
S : Float := Float (S1);
L : Float := Float (A'Length);
-- ERROR: declaring variables
S2 : constant Integer_Sum := Sum (A);
-- ERROR: declaring constant of
-- limited type
A1 : Const_Integer_Access :=
S1'Unchecked_Access;
-- ERROR: using 'Unchecked_Access
-- attribute
A2 : access Integer := null;
-- ERROR: declaring object of
-- anonymous access type
begin
S / L);

private

type Integer_Sum is new Integer;

end Integer_Arrays;

package body Integer_Arrays is

function Sum (Arr : Integer_Array)
return Integer is
begin
return Acc := 0 do
for V of Arr loop
Acc := Acc + V;
end loop;
end return;
end Sum;

(continues on next page)
Learning Ada

function Sum (Arr : Integer_Array) return Integer_Sum is (Integer_Sum (Integer' (Sum (Arr))));
end Integer_Arrays;

Code block metadata

MD5: e1f72f817baea87f66fb34b6aa8d1949

Build output

integer_arrays.ads:28:10: error: "aliased" not allowed in declare_expression
integer_arrays.ads:31:10: error: object renaming or constant declaration expected
integer_arrays.ads:32:10: error: object renaming or constant declaration expected
integer_arrays.ads:35:10: error: object renaming or constant declaration expected
integer_arrays.ads:40:19: error: "Unchecked_Access" attribute cannot occur in a
→ declare_expression
integer_arrays.ads:44:15: error: anonymous access type not allowed in declare_expression
gprbuild: *** compilation phase failed

In this version of the Avg function, we see many errors in the declarative part of the declare expression. If we convert the declare expression into an actual function implementation, however, those declarations won't trigger compilation errors. (Feel free to try this out!)

26.1.5 Reduction Expressions

Note: This feature was introduced in Ada 2022.

A reduction expression reduces a list of values into a single value. For example, we can reduce the list \([2, 3, 4]\) to a single value:

- by adding the values of the list: \(2 + 3 + 4 = 9\), or
- by multiplying the values of the list: \(2 \times 3 \times 4 = 24\).

We write a reduction expression by using the Reduce attribute and providing the reducer and its initial value:

- the reducer is the operator (e.g.: + or \(^\star\)) that we use to combine the values of the list;
- the initial value is the value that we use before all other values of the list.

For example, if we use + as the operator and 0 as the initial value, we get the reduction expression: \(0 + 2 + 3 + 4 = 9\). This can be implemented using an array:

Listing 28: show_reduction_expression.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Reduction_Expression is
    A : array (1 .. 3) of Integer;
    I : Integer;
begin
    A := [2, 3, 4];
    I := A'Reduce ("+", 0);
```

(continues on next page)
Learning Ada

Put_Line ("A = 
  & A'Image);
Put_Line ("I = 
  & I'Image);
end Show_Reduction_Expression;

Code block metadata
Simple_Reduction_Expression
MD5: 1a0164b3c4768125c8dbbe8a0f4955a1

Runtime output
A =  
[ 2, 3, 4]
I =  9

Here, we have the array A with a list of values. The A'Reduce ("+", 0) expression reduces
the list of values of A into a single value — in this case, an integer value that is stored in I. This statement is equivalent to:

I := 0;
for E of A loop
  I := I + E;
end loop;

Naturally, we can reduce the array using the * operator:

Listing 29: show_reduction_expression.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Reduction_Expression is
  A : array (1 .. 3) of Integer;
  I : Integer;
begin
  A := [2, 3, 4];
  I := A'Reduce ("**", 1);
  Put_Line ("A = 
    & A'Image);
  Put_Line ("I = 
    & I'Image);
end Show_Reduction_Expression;

Code block metadata
Simple_Reduction_Expression
MD5: 415b1ee8b21cca6d2438a34c88e7e2df

Runtime output
A =  
[ 2, 3, 4]
I =  24

In this example, we call A'Reduce ("**", 1) to reduce the list. (Note that we use an
Learning Ada

Initial value of one because it is the identity element of a multiplication, so the complete operation is: \(1 \times 2 \times 3 \times 4 = 24\).

In the Ada Reference Manual

- Reduction Expressions

Value sequences

In addition to arrays, we can apply reduction expression to value sequences, which consist of an iterated element association— for example, \([\text{for } I \text{ in } 1 \ldots 3 \Rightarrow I + 1]\). We can simply append the reduction expression to a value sequence:

Listing 30: show_reduction_expression.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Reduction_Expression is
    I : Integer;
begin
    I := [\text{for } I \text{ in } 1 \ldots 3 \Rightarrow I + 1]\ 'Reduce (+, 0);
    Put_Line ("I = " & I'Image);
    I := [\text{for } I \text{ in } 1 \ldots 3 \Rightarrow I + 1]\ 'Reduce (*, 1);
    Put_Line ("I = " & I'Image);
end Show_Reduction_Expression;
```

Code block metadata

Reduction_Expression_Value_Sequences
MD5: e714f69700e3f0387314ee0e531620c4

Runtime output

I = 9
I = 24

In this example, we create the value sequence \([\text{for } I \text{ in } 1 \ldots 3 \Rightarrow I + 1]\) and reduce it using the + and * operators. (Note that the operations in this example have the same results as in the previous examples using arrays.)

Custom reducers

In the previous examples, we've used standard operators such as + and * as the reducer. We can, however, write our own reducers and pass them to the Reduce attribute. For example:

Listing 31: show_reduction_expression.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Reduction_Expression is
    type Integer_Array is
        array (Positive range <>) of Integer;
    A : Integer_Array (1 .. 3);
    I : Long_Integer;

    procedure Accumulate
        (Accumulator : in out Long_Integer;
         Value      : Integer) is
    begin
        Accumulator := Accumulator + Long_Integer (Value);
    end Accumulate;

    begin
        A := [2, 3, 4];
        I := A'Reduce (Accumulate, 0);
        Put_Line ("A = " & A'Image);
        Put_Line ("I = " & I'Image);
    end Show_Reduction_Expression;
```

Code block metadata

CustomReducerProcedure
MD5: 3190a1ff6a8027268ca96a75cf214714

Runtime output

```
A = [2, 3, 4]
I = 9
```

In this example, we implement the Accumulate procedure as our reducer, which is called to accumulate the individual elements (integer values) of the list. We pass this procedure to the Reduce attribute in the `I := A'Reduce (Accumulate, 0)` statement, which is equivalent to:

```ada
I := 0;
for E of A loop
    Accumulate (I, E);
end loop;
```

A custom reducer must have the following parameters:

1. The accumulator parameter, which stores the interim result — and the final result as well, once all elements of the list have been processed.
2. The value parameter, which is a single element from the list.
Note that the accumulator type doesn't need to match the type of the individual components. In this example, we're using \texttt{Integer} as the component type, while the accumulator type is \texttt{Long_Integer}. (For this kind of reducers, using \texttt{Long_Integer} instead of \texttt{Integer} for the accumulator type makes lots of sense due to the risk of triggering overflows while the reducer is accumulating values — e.g. when accumulating a long list with larger numbers.)

In the example above, we've implemented the reducer as a procedure. However, we can also implement it as a function. In this case, the accumulated value is returned by the function:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Reduction_Expression is
  type Integer_Array is
    array (Positive range <>) of Integer;

  A : Integer_Array (1 .. 3);
  I : Long_Integer;

  function Accumulate
      (Accumulator : Long_Integer;
       Value : Integer)
  return Long_Integer is
  begin
    return Accumulator + Long_Integer (Value);
  end Accumulate;

begin
  A := [2, 3, 4];
  I := A'Reduce (Accumulate, 0);
  Put_Line ("A = ",
            & A'Image);
  Put_Line ("I = ",
            & I'Image);
  end Show_Reduction_Expression;
```

In this example, we converted the \texttt{Accumulate} procedure into a function (while the core implementation is essentially the same).

Note that the reduction expression remains the same, independently of whether we're using a procedure or a function as the reducer. Therefore, the statement with the reduction expression in this example is the same as in the previous example: \( I := A'\text{Reduce} (\text{Accumulate}, 0) \). Now that we're using a function, this statement is equivalent to:

\[
I := 0;
for E of A loop
\]

(continues on next page)
Other accumulator types

The accumulator type isn't restricted to scalars: in fact, we could use record types as well. For example:

Listing 33: show_reduction_expression.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Reduction_Expression is
  type Integer_Array is
    array (Positive range <>) of Integer;
  A : Integer_Array (1 .. 3);

type Integer_Accumulator is record
  Value : Long_Integer;
  Count : Integer;
end record;

function Accumulate
  (Accumulator : Integer_Accumulator;
   Value : Integer)
  return Integer_Accumulator is
begin
  return (Value => Accumulator.Value + Long_Integer (Value),
          Count => Accumulator.Count + 1);
end Accumulate;

function Zero return Integer_Accumulator is
  (Value => 0, Count => 0);

function Average (Acc : Integer_Accumulator)
  return Float is
  (Float (Acc.Value) / Float (Acc.Count));

Acc : Integer_Accumulator;

begin
  A := [2, 3, 4];

  Acc := A'Reduce (Accumulate, Zero);
  Put_Line ("Acc = ", Acc'Image);
  Put_Line ("Avg = ", Average (Acc)'Image);
end Show_Reduction_Expression;
```

In this example, we're using the Integer_Accumulator record type in our reducer — the Accumulate function. In this case, we're not only accumulating the values, but also counting the number of elements in the list. (Of course, we could have used A'Length for that as well.)

Also, we're not limited to numeric types: we can also create a reducer using strings as the accumulator type. In fact, we can display the initial value and the elements of the list by using unbounded strings:
Listing 34: show_reduction_expression.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

procedure Show_Reduction_Expression is
  type Integer_Array is
    array (Positive range <>) of Integer;
  A : Integer_Array (1 .. 3);

  function Unbounded_String_List
    (Accumulator : Unbounded_String;
     Value : Integer)
    return Unbounded_String is
    begin
      return Accumulator
        & " " & Value'Image;
    end Unbounded_String_List;

  begin
    A := [2, 3, 4];
    Put_Line ("A = ": A'Image);
    Put_Line ("L = ": To_String (A'Reduce
                 (Unbounded_String_List,
                  To_Unbounded_String ("0"))));
  end Show_Reduction_Expression;
```

Code block metadata

Reducer_String_Accumulator
MD5: 43c54e93e404a235c8721db7c691a864

Runtime output

```text
A = [2, 3, 4]
L = 0, 2, 3, 4
```

In this case, the "accumulator" is concatenating the initial value and individual values of the list into a string.

### 26.2 Statements

#### 26.2.1 Simple and Compound Statements

We can classify statements as either simple or compound. Simple statements don't contain other statements; think of them as "atomic units" that cannot be further divided. Compound statements, on the other hand, may contain other — simple or compound — statements.

Here are some examples from each category:
Learning Ada

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple statements</td>
<td>Null statement, assignment, subprogram call, etc.</td>
</tr>
<tr>
<td>Compound statements</td>
<td>If statement, case statement, loop statement, block statement</td>
</tr>
</tbody>
</table>

In the Ada Reference Manual

- 5.1 Simple and Compound Statements - Sequences of Statements

26.2.2 Labels

We can use labels to identify statements in the code. They have the following format: `<<Some_Label>>`. We write them right before the statement we want to apply it to. Let’s see an example of labels with simple statements:

Listing 35: show_statement_identifier.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Statement_Identifier is
    pragma Warnings (Off, "is not referenced");
begin
    <<Show_Hello>> Put_Line ("Hello World!");
    <<Show_Test>>  Put_Line ("This is a test.");

    <<Show_Separator>>
    <<Show_Block_Separator>>
    Put_Line ("====================");
end Show_Statement_Identifier;
```

Code block metadata

MD5: 820f5963b476af5c04314fd4373d2286

Runtime output

Hello World!
This is a test.
====================

Here, we’re labeling each statement. For example, we use the Show_Hello label to identify the Put_Line ("Hello World!"); statement. Note that we can use multiple labels a single statement. In this code example, we use the Show_Separator and Show_Block_Separator labels for the same statement.

In the Ada Reference Manual

- 5.1 Simple and Compound Statements - Sequences of Statements

Labels and goto statements

Labels are mainly used in combination with goto statements. (Although pretty much uncommon, we could potentially use labels to indicate important statements in the code.) Let's see an example where we use a goto label; statement to jump to a specific label:

Listing 36: show_cleanup.adb

```ada
procedure Show_Cleanup is
  pragmaWarnings (Off, "always false");
begin
  Some_Error : Boolean;
  if Some_Error then
    goto Cleanup;
  end if;
  <<Cleanup>> null;
end Show_Cleanup;
```

Use-case: Continue

Another use-case is that of a Continue label in a loop. Consider a loop where we want to skip further processing depending on a condition:

Listing 37: show_continue.adb

```ada
procedure Show_Continue is
  function Is_Further_Processing_Needed (Dummy : Integer) return Boolean is begin return False; end Is_Further_Processing_Needed;
  A : constant array (1 .. 10) of Integer := (others => 0);
begin
  for E of A loop
    -- Some stuff here...
    if Is_Further_Processing_Needed (E) then
      -- Do more stuff...
      null;
    end if;
  end loop;
end Show_Continue;
```
In this example, we call the `Is_Further_Processing_Needed (E)` function to check whether further processing is needed or not. If it’s needed, we continue processing in the `if` statement. We could simplify this code by just using a Continue label at the end of the loop and a `goto` statement:

```
procedure Show_Continue is
  function Is_Further_Processing_Needed
    (Dummy : Integer)
    return Boolean
  is
    -- Dummy implementation
    return False;
  end Is_Further_Processing_Needed;

  A : constant array (1 .. 10) of Integer :=
  (others => 0);
begin
  for E of A loop
    -- Some stuff here...
    if not Is_Further_Processing_Needed (E) then
      goto Continue;
    end if;
    -- Do more stuff...
    <<Continue>>
  end loop;
end Show_Continue;
```

Here, we use a Continue label at the end of the loop and jump to it in the case that no further processing is needed. Note that, in this example, we don’t have a statement after the `Continue` label because the label itself is at the end of a statement — to be more specific, at the end of the loop statement. In such cases, there’s an implicit `null` statement.

**Historically**

Since Ada 2012, we can simply write:

```
loop
  -- Some statements...
  <<Continue>>
end loop;
```

If a label is used at the end of a sequence of statements, a `null` statement is implied. In previous versions of Ada, however, that is not the case. Therefore, when using those versions of the language, we must write at least a `null` statement.
Learning Ada

```
loop
   -- Some statements...
   <<Continue>> null;
end loop;
```

Labels and compound statements

We can use labels with compound statements as well. For example, we can label a `for` loop:

Listing 39: show_statement_identifier.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Statement_Identifier is
   pragma Warnings (Off, "is not referenced");
   Arr : constant array (1 .. 5) of Integer :=
      (1, 4, 6, 42, 49);
   Found : Boolean := False;
begin
   <<Find_42>> for E of Arr loop
      if E = 42 then
         Found := True;
         exit;
      end if;
   end loop;
   Put_Line ("Found: " & Found'Image);
end Show_Statement_Identifier;
```

Code block metadata

MD5: 5ca80b5a379ba0b08ccfaa4c6eab64d5

Runtime output

Found: TRUE

For further reading...

In addition to labels, loops and block statements allow us to use a statement identifier. In simple terms, instead of writing `<<Some_Label>>`, we write `Some_Label :.

We could rewrite the previous code example using a loop statement identifier:

Listing 40: show_statement_identifier.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Statement_Identifier is
   Arr : constant array (1 .. 5) of Integer :=
      (1, 4, 6, 42, 49);
   Found : Boolean := False;
begin
   Find_42 : for E of Arr loop
```
if E = 42 then
    Found := True;
    exit Find_42;
end if;
end loop Find_42;

Put_Line ("Found: " & Found'Image);
end Show_Statement_Identifier;

26.2.3 Exit loop statement

We've introduced bare loops back in the Introduction to Ada course (page 15). In this section, we'll briefly discuss loop names and exit loop statements.

A bare loop has this form:

```ada
loop
    exit when Some_Condition;
end loop;
```

We can name a loop by using a loop statement identifier:

```ada
Loop_Name:
    loop
        exit Loop_Name when Some_Condition;
    end loop Loop_Name;
```

In this case, we have to use the loop's name after end loop. Also, having a name for a loop allows us to indicate which loop we're exiting from: exit Loop_Name when.

Let's see a complete example:

Listing 41: show_vector_cursor_iteration.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Vectors;

procedure Show_Vector_Cursor_Iteration is
    package Integer_Vectors is new
        Ada.Containers.Vectors
            (Index_Type => Positive,
             Element_Type => Integer);
    use Integer_Vectors;
```
Learning Ada

(continued from previous page)

12
13 V : constant Vector := 20 & 10 & 0 & 13;
14 C : Cursor;
15 begin
16 C := V.First;
17 Put_Line ("Vector elements are: ");
18
19 Show_Elements :
20 loop
21 exit Show_Elements when C = No_Element;
22
23 Put_Line ("Element: " & Integer'Image (V (C)));
24 C := Next (C);
25 end loop Show_Elements;
26 end Show_Vector_Cursor_Iteration;

Code block metadata

MD5: b77353f6ed98f8ddb32c73c47d249020

Runtime output

Vector elements are:
Element: 20
Element: 10
Element: 0
Element: 13

Naming a loop is particularly useful when we have nested loops and we want to exit directly from the inner loop:

Listing 42: show_inner_loop_exit.adb

1 procedure Show_Inner_Loop Exit is
2 pragma Warnings (Off);
3 Cond : Boolean := True;
4 begin
5 Outer_Processing : loop
6 Inner_Processing : loop
7 exit Outer_Processing when Cond;
8 end loop Inner_Processing;
9 end loop Outer_Processing;
10 end Show_Inner_Loop Exit;

Code block metadata

MD5: b5c7434f1bf23c2cb8f81e4c13a31386

Here, we indicate that we exit from the Outer_Processing loop in case a condition Cond is met, even if we're actually within the inner loop.
In the Ada Reference Manual

- 5.7 Exit Statements

26.2.4 If, case and loop statements

In the Introduction to Ada course, we talked about if statements (page 11), loop statements (page 13), and case statements (page 16). This is a very simple code example with these statements:

Listing 43: show_if_case_loop_statements.adb

```ada
procedure Show_If_Case_Loop_Statements is
  pragma Warnings (Off);
  Reset : Boolean := False;
  Increment : Boolean := True;
  Val : Integer := 0;
begin
  -- If statement
  --
  if Reset then
    Val := 0;
  elsif Increment then
    Val := Val + 1;
  else
    Val := Val - 1;
  end if;
  -- Loop statement
  --
  for I in 1 .. 5 loop
    Val := Val * 2 - I;
  end loop;
  -- Case statement
  --
  case Val is
    when 0 .. 5 =>
      null;
    when others =>
      Val := 5;
  end case;
end Show_If_Case_Loop_Statements;
```

In this section, we'll look into a more advanced detail about the case statement.

Case statements and expressions

As we know, the case statement has a choice expression (case Choice_Expression is), which is expected to be a discrete type. Also, this expression can be a function call or a type conversion, for example — in addition to being a variable or a constant.

As we discussed earlier on (page 585), if we use parentheses, the contents between those parentheses is parsed as an expression. In the context of case statements, the expression is first evaluated before being used as a choice expression. Consider the following code example:

```ada
package Scales is
  type Satisfaction_Scale is (Very_Dissatisfied, Dissatisfied, OK, Satisfied, Very_Satisfied);
  type Scale is range 0 .. 10;
  function To_Satisfaction_Scale (S : Scale) return Satisfaction_Scale;
end Scales;
```

```ada
package body Scales is
  function To_Satisfaction_Scale (S : Scale) return Satisfaction_Scale is
    Satisfaction : Satisfaction_Scale;
    begin
      case S is
        when 0 .. 2 => Satisfaction := Very_Dissatisfied;
        when 3 .. 4 => Satisfaction := Dissatisfied;
        when 5 .. 6 => Satisfaction := OK;
        when 7 .. 8 => Satisfaction := Satisfied;
    end case;
end Scales;
```

(continues on next page)

---

When we try to compile this code example, the compiler complains about missing values in the `To_Satisfaction_Scale` function. As we mentioned in the *Introduction to Ada course* (page 16), every possible value for the choice expression needs to be covered by a unique branch of the case statement. In principle, it *seems* that we’re actually covering all possible values of the `Scale` type, which ranges from 0 to 10. However, we’ve written `case (S)` *is* instead of `case S is`. Because of the parentheses, `(S)` is evaluated as an expression. In this case, the expected range of the case statement is not `Scale'Range`, but the range of its `base type` (page 283) `Scale'Base'Range`.

**In other languages**

In C, the switch-case statement requires parentheses for the choice expression:

```
Listing 47: main.c
#include <stdio.h>
int main(int argc, const char * argv[])
{
    int s = 0;
    switch (s)
```

(continues on next page)
9
{ 
  case 0: 
  case 1: 
    printf("Value in the 0 -- 1 range\n"); 
  default: 
    printf("Value > 1\n"); 
} 
}

26.2.5 Block Statements

We've introduced block statements back in the Introduction to Ada course (page 19). They have this simple form:

Listing 48: show_block_statement.adb

procedure Show_Block_Statement is 
  pragma Warnings (Off); 
begin 
  -- BLOCK STARTS HERE: 
  declare 
    I : Integer; 
  begin 
    I := 0; 
  end; 
end Show_Block_Statement; 

We can use an identifier when writing a block statement. (This is similar to loop statement identifiers that we discussed in the previous section.) In this example, we implement a block called Simple_Block:

Listing 49: show_block_statement.adb

(continues on next page)
begin

  Simple_Block : declare
    I : Integer;
  begin
    I := 0;
  end Simple_Block;
end Show_Block_Statement;

Code block metadata

MD5: b327b7675931d9b994637671c806c7c3

Note that we must write end Simple_Block; when we use the Simple_Block identifier. Block statement identifiers are useful:

- to indicate the begin and the end of a block — as some blocks might be long or nested in other blocks;
- to indicate the purpose of the block (i.e. as code documentation).

In the Ada Reference Manual

- 5.6 Block Statements\(^{131}\)

26.2.6 Extended return statement

A common idiom in Ada is to build up a function result in a local object, and then return that object:

Listing 50: show_return.adb

```ada
procedure Show_Return is
  type Array_Of_Natural is
    array (Positive range <>) of Natural;
  function Sum (A : Array_Of_Natural) return Natural is
    Result : Natural := 0;
  begin
    for Index in A'Range loop
      Result := Result + A (Index);
    end loop;
    return Result;
  end Sum;
begin
  null;
end Show_Return;
```

Code block metadata

\(^{131}\) http://www.ada-auth.org/standards/22rm/html/RM-5-6.html
Since Ada 2005, a notation called the extended return statement is available: this allows you to declare the result object and return it as part of one statement. It looks like this:

```ada
procedure Show_Extended_Return is
  type Array_Of_Natural is
    array (Positive range <>) of Natural;
  function Sum (A : Array_Of_Natural) return Natural is
    begin
      return Result : Natural := 0 do
        for Index in A'Range loop
          Result := Result + A (Index);
        end loop;
      end return;
    end Sum;
begin
  null;
end Show_Extended_Return;
```

The return statement here creates `Result`, initializes it to 0, and executes the code between `do` and `end return`. When `end return` is reached, `Result` is automatically returned as the function result.

**In the Ada Reference Manual**

- 6.5 Return Statements\(^{132}\)

**Other usages of extended return statements**

**Note:** This section was originally written by Robert A. Duff and published as Gem #10: Limited Types in Ada 2005\(^{133}\).

While the `extended_return` statement was added to the language specifically to support `limited constructor functions` (page 958), it comes in handy whenever you want a local name for the function result:

\(^{132}\) http://www.ada-auth.org/standards/22rm/html/RM-6-5.html
\(^{133}\) https://www.adacore.com/gems/ada-gem-10
Listing 52: show_string_construct.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_String_Construct is

  function Make_String
    (S     : String;
     Prefix : String;
     Use_Prefix : Boolean) return String
  is
    Length : Natural := S'Length;
  begin
    if Use_Prefix then
      Length := Length + Prefix'Length;
    end if;

    return Result : String (1 .. Length) do
      -- fill in the characters
      if Use_Prefix then
        Result (1 .. Prefix'Length) := Prefix;
        Result (Prefix'Length + 1 .. Length) := S;
      else
        Result := S;
      end if;
    end return;
  end Make_String;

  S1 : String := "Ada";
  S2 : String := "Make_With_";
begin
  Put_Line ("No prefix: ": & Make_String (S1, S2, False));
  Put_Line ("With prefix: ": & Make_String (S1, S2, True));
end Show_String_Construct;
```

Code block metadata

Extended_Return_Other_Usages
MD5: a2b26ceed06a0ab66aff6c2b59c02003

Runtime output

No prefix: Ada
With prefix: Make_With_Ada

In this example, we first calculate the length of the string and store it in Length. We then use this information to initialize the return object of the Make_String function.
26.3 Subprograms

26.3.1 Parameter Modes and Associations

In this section, we discuss some details about parameter modes and associations. First of all, as we know, parameters can be either formal or actual:

- Formal parameters are the ones we see in a subprogram declaration and implementation;
- Actual parameters are the ones we see in a subprogram call.
  - Note that actual parameters are also called subprogram arguments in other languages.

We define parameter associations as the connection between an actual parameter in a subprogram call and its declaration as a formal parameter in a subprogram specification or body.

In the Ada Reference Manual

- 6.2 Formal Parameter Modes
- 6.4.1 Parameter Associations

Formal Parameter Modes

We already discussed formal parameter modes in the Introduction to Ada (page 28) course:

<table>
<thead>
<tr>
<th>Parameter Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Parameter can only be read, not written</td>
</tr>
<tr>
<td>out</td>
<td>Parameter can be written to, then read</td>
</tr>
<tr>
<td>in out</td>
<td>Parameter can be both read and written</td>
</tr>
</tbody>
</table>

As this topic was already discussed in that course — and we used parameter modes extensively in all code examples from that course —, we won't introduce the topic again here. Instead, we'll look into some of the more advanced details.

By-copy and by-reference

In the Introduction to Ada (page 28) course, we saw that parameter modes don't correspond directly to how parameters are actually passed. In fact, an in out parameter could be passed by copy. For example:

Listing 53: check_param_passing.ads

```ada
with System;

procedure Check_Param_Passing
  (Formal : System.Address;
  Actual : System.Address);
```

134 http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
135 http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
Listing 54: check_param_passing.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System.Address_Image;

procedure Check_Param_Passing
( Formal : System.Address;
  Actual : System.Address ) is
begin
  Put_Line ("Formal parameter at 
            & System.Address_Image (Formal));
  Put_Line ("Actual parameter at 
            & System.Address_Image (Actual));
  if System.Address_Image (Formal) =
      System.Address_Image (Actual)
  then
    Put_Line ("Parameter is passed by reference.");
  else
    Put_Line ("Parameter is passed by copy.");
  end if;
end Check_Param_Passing;
```

Listing 55: machine_x.ads

```ada
with System;

package Machine_X is

  procedure Update_Value
  ( V : in out Integer;
    AV : System.Address );

end Machine_X;
```

Listing 56: machine_x.adb

```ada
with Check_Param_Passing;

package body Machine_X is

  procedure Update_Value
  ( V : in out Integer;
    AV : System.Address ) is
begin
  V := V + 1;
  Check_Param_Passing (Formal => V'Address,
                       Actual => AV);
end Update_Value;
end Machine_X;
```

Listing 57: show_by_copy_by_ref_params.adb

```ada
with Machine_X; use Machine_X;

procedure Show_By_Copy_By_Ref_Params is
  A : Integer := 5;
begin
end Show_By_Copy_By_Ref_Params;
```

(continues on next page)
As we can see by running this example,

- the integer variable `A` in the `Show_By_Copy_By_Ref_Params` procedure

and

- the `V` parameter in the `Update_Value` procedure

have different addresses, so they are different objects. Therefore, we conclude that this parameter is being passed by value, even though it has the `in out` mode. (We talk more about addresses and the `'Address` attribute later on (page 401)).

As we know, when a parameter is passed by copy, it is first copied to a temporary object. In the case of a parameter with `in out` mode, the temporary object is copied back to the original (actual) parameter at the end of the subprogram call. In our example, the temporary object indicated by `V` is copied back to `A` at the end of the call to `Update_Value`.

In Ada, it's not the parameter mode that determines whether a parameter is passed by copy or by reference, but rather its type. We can distinguish between three categories:

1. By-copy types;
2. By-reference types;
3. *Unspecified* types.

Obviously, parameters of by-copy types are passed by copy and parameters of by-reference type are passed by reference. However, if a category isn't specified — i.e. when the type is neither a by-copy nor a by-reference type —, the decision is essentially left to the compiler.

As a rule of thumb, we can say that;

- elementary types — and any type that is essentially elementary, such as a private type whose full view is an elementary type — are passed by copy;
- tagged and explicitly limited types — and other types that are essentially tagged, such as task types — are passed by reference.

The following table provides more details:
Learning Ada

<table>
<thead>
<tr>
<th>Type category</th>
<th>Parameter passing</th>
<th>List of types</th>
</tr>
</thead>
</table>
| By copy       | By copy           | • Elementary types  
|               |                   | • Descendant of a private type whose full type is a by-copy type |
| By reference  | By reference      | • Tagged types  
|               |                   | • Task and protected types  
|               |                   | • Explicitly limited record types  
|               |                   | • Composite types with at least one subcomponent of a by-reference type  
|               |                   | • Private types whose full type is a by-reference type  
|               |                   | • Any descendant of the types mentioned above |
| Unspecified   | Either by copy or by reference | • Any type not mentioned above |

Note that, for parameters of limited types, only those parameters whose type is explicitly limited are always passed by reference. We discuss this topic in more details in another chapter (page 967).

Let’s see an example:

Listing 58: machine_x.ads

```ada
with System;

package Machine_X is

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Rec is record
    A : Integer;
  end record;

  type Rec_Array is record
    A : Integer;
    Arr : Integer_Array (1 .. 100);
  end record;

  type Tagged_Rec is tagged record
    A : Integer;
  end record;

  procedure Update_Value
    (R : in out Rec;
```

(continues on next page)
Learning Ada

Listing 59: machine_x.adb

```ada
with Check_Param_Passing;

package body Machine_X is

procedure Update_Value
(R : in out Rec;  
AR : System.Address)
is
begin
  R.A := R.A + 1;
  Check_Param_Passing (Formal => R'Address,  
                        Actual => AR);
end Update_Value;

procedure Update_Value
(RA : in out Rec_Array;  
ARA : System.Address)
is
begin
  RA.A := RA.A + 1;
  Check_Param_Passing (Formal => RA'Address,  
                        Actual => ARA);
end Update_Value;

procedure Update_Value
(R : in out Tagged_Rec;  
AR : System.Address)
is
begin
  R.A := R.A + 1;
  Check_Param_Passing (Formal => R'Address,  
                        Actual => AR);
end Update_Value;

end Machine_X;
```

Listing 60: show_by_copy_by_ref_params.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Machine_X; use Machine_X;

procedure Show_By_Copy_By_Ref_Params is
  TR : Tagged_Rec := (A => 5);
  R : Rec := (A => 5);
  RA : Rec_Array := (A => 5,
                     Arr => (others => 0));
```

(continues on next page)
begin
  Put_Line ("Tagged record");
  Update_Value (TR, TR'Address);
  Put_Line ("Untagged record");
  Update_Value (R, R'Address);
  Put_Line ("Untagged record with array");
  Update_Value (RA, RA'Address);
end Show_By_Copy_By_Ref_Params;

Code block metadata
MD5: 3ca46380c4df36af9393041181ff2f17

Runtime output
Tagged record
Formal parameter at 00007FFD9D7686B0
Actual parameter at 00007FFD9D7686B0
Parameter is passed by reference.
Untagged record
Formal parameter at 00007FFD9D7684FC
Actual parameter at 00007FFD9D7686AC
Parameter is passed by copy.
Untagged record with array
Formal parameter at 00007FFD9D768510
Actual parameter at 00007FFD9D768510
Parameter is passed by reference.

When we run this example, we see that the object of tagged type (Tagged_Rec) is passed by reference to the Update_Value procedure. In the case of the objects of untagged record types, you might see this:

- the parameter of Rec type — which is an untagged record with a single component of integer type —, the parameter is passed by copy;
- the parameter of Rec_Array type — which is an untagged record with a large array of 100 components —, the parameter is passed by reference.

Because Rec and Rec_Array are neither by-copy nor by-reference types, the decision about how to pass them to the Update_Value procedure is made by the compiler. (Thus, it is possible that you see different results when running the code above.)

Bounded errors

When we use parameters of types that are neither by-copy nor by-reference types, we might encounter the situation where we have the same object bound to different names in a subprogram. For example, if:

- we use a global object Global_R of a record type Rec
  and
- we have a subprogram with an in-out parameter of the same record type Rec
  and
- we pass Global_R as the actual parameter for the in-out parameter of this subprogram,
then we have two access paths to this object: one of them using the global variable directly, and the other one using it indirectly via the in-out parameter. This situation could lead to undefined behavior or to a program error. Consider the following code example:

**Listing 61: machine_x.ads**

```ada
with System;
package Machine_X is
type Rec is record
  A : Integer;
end record;
Global_R : Rec := (A => 0);
procedure Update_Value
  (R : in out Rec;
   AR : System.Address);
end Machine_X;
```

**Listing 62: machine_x.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Check_Param_Passing;
package body Machine_X is
  procedure Update_Value
    (R : in out Rec;
     AR : System.Address)
  is
    procedure Show_Vars is
    begin
      Put_Line ("Global_R.A: ",
                & Integer'Image (Global_R.A));
      Put_Line ("R.A: ",
                & Integer'Image (R.A));
    end Show_Vars;
  begin
    Check_Param_Passing (Formal => R'Address,
                         Actual => AR);
    Put_Line ("Incrementing Global_R.A...");
    Global_R.A := Global_R.A + 1;
    Show_Vars;
    Put_Line ("Incrementing R.A...");
    R.A := R.A + 5;
    Show_Vars;
  end Update_Value;
end Machine_X;
```

**Listing 63: show_by_copy_by_ref_params.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Machine_X; use Machine_X;
procedure Show_By_Copy_By_Ref_Params is
  (continues on next page)
```
begin
  Put_Line ("Calling Update_Value...");
  Update_Value (Global_R, Global_R'Address);
  Put_Line ("After call to Update_Value...");
  Put_Line ("Global_R.A: " & Integer'Image (Global_R.A));
end Show_By_Copy_By_Ref_Params;

Code block metadata

| Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_Associations.By_Copy_By_Ref_Params | MD5: 96be7054b7ff64a304705edf6b15f031 |

Runtime output

Calling Update_Value...
Formal parameter at 00007FFC5B7D4A9C
Actual parameter at 00000000008003BC
Parameter is passed by copy.
Incrementing Global_R.A...
Global_R.A: 1
R.A: 0
Incrementing R.A...
Global_R.A: 1
R.A: 5
After call to Update_Value...
Global_R.A: 5

In the Update_Value procedure, because Global_R and R have a type that is neither a by-pass nor a by-reference type, the language does not specify whether the old or the new value would be read in the calls to Put_Line. In other words, the actual behavior is undefined. Also, this situation might raise the Program_Error exception.

Important

As a general advice:

- you should be very careful when using global variables and
- you should avoid passing them as parameters in situations such as the one illustrated in the code example above.

Aliased parameters

When a parameter is specified as aliased, it is always passed by reference, independently of the type we’re using. In this sense, we can use this keyword to circumvent the rules mentioned so far. (We discuss more about aliasing (page 777) and aliased parameters (page 785) later on.)

Let's rewrite a previous code example that has a parameter of elementary type and change it to aliased:

Listing 64: machine_x.ads

```ada
with System;
```
package Machine_X is

procedure Update_Value
  (V : aliased in out Integer;
   AV : System.Address);

end Machine_X;

Listing 65: machine_x.adb

with Check_Param_Passing;
package body Machine_X is

procedure Update_Value
  (V : aliased in out Integer;
   AV : System.Address)
is
begin
  V := V + 1;
  Check_Param_Passing (Formal => V'Address,
                        Actual => AV);
end Update_Value;

end Machine_X;

Listing 66: show_by_copy_by_ref_params.adb

with Machine_X; use Machine_X;
procedure Show_By_Copy_By_Ref_Params is
  A : aliased Integer := 5;
begin
  Update_Value (A, A'Address);
end Show_By_Copy_By_Ref_Params;

Code block metadata
MD5: c066af3a7081815d0a7598733f9e6aee

Runtime output
Formal parameter at 00007FFD697FCACC
Actual parameter at 00007FFD697FCACC
Parameter is passed by reference.

As we can see, A is now passed by reference.

Note that we can only pass aliased objects to aliased parameters. If we try to pass a non-aliased object, we get a compilation error:

Listing 67: show_by_copy_by_ref_params.adb

with Machine_X; use Machine_X;
procedure Show_By_Copy_By_Ref_Params is
  A : Integer := 5;
begin
(continues on next page)
Update_Value (A, A'Address);
end Show_By_Copy_By_Ref_Params;

Code block metadata

MD5: 9e6586e0b771de68040131cae81799b8

Build output

show_by_copy_by_ref_params.adb:6:18: error: actual for aliased formal "V" must be an aliased object
gprbuild: *** compilation phase failed

Again, we discuss more about aliased parameters (page 785) and aliased objects (page 778) later on in the context of access types.

Parameter Associations

When actual parameters are associated with formal parameters, some rules are checked. As a typical example, the type of each actual parameter must match the type of the corresponding actual parameter. In this section, we see some details about how this association is made and some of the potential errors.

In the Ada Reference Manual

- 6.4.1 Parameter Associations

Parameter order and association

As we already know, when calling subprograms, we can use positional or named parameter association — or a mixture of both. Also, parameters can have default values. Let's see some examples:

Listing 68: operations.ads

```ada
package Operations is
  procedure Add (Left : in out Integer;
                  Right : Float := 1.0);
end Operations;
```

Listing 69: operations.adb

```ada
package body Operations is
  procedure Add (Left : in out Integer;
                 Right : Float := 1.0) is
    begin
      Left := Left + Integer (Right);
    end Add;
```

136 http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html

26.3. Subprograms

627
Listing 70: show_param_association.adb

```ada
with Operations; use Operations;

procedure Show_Param_Association is
  A : Integer := 5;
begin
  -- Positional association
  Add (A, 2.0);

  -- Positional association (using default value)
  Add (A);

  -- Named association
  Add (Left => A, Right => 2.0);

  -- Named association (inversed order)
  Add (Right => 2.0, Left => A);

  -- Mixed positional / named association
  Add (A, Right => 2.0);
end Show_Param_Association;
```

This code snippet has examples of positional and name parameter association. Also, it has an example of mixed positional / named parameter association. In most cases, the actual A parameter is associated with the formal Left parameter, and the actual 2.0 parameter is associated with the formal Right parameter.

In addition to that, parameters can have default values, so, when we write Add (A), the A variable is associated with the Left parameter and the default value (1.0) is associated with the Right parameter.

Also, when we use named parameter association, the parameter order is irrelevant: we can, for example, write the last parameter as the first one. Therefore, we can write Add (Right => 2.0, Left => A) instead of Add (Left => A, Right => 2.0).

### Ambiguous calls

Ambiguous calls can be detected by the compiler during parameter association. For example, when we have both default values in parameters and subprogram overloading, the compiler might be unable to decide which subprogram we're calling:

Listing 71: operations.ads

```ada
package Operations is
  procedure Add (Left : in out Integer);
end Operations;
```
As we see in this example, the Add procedure is overloaded. The first instance has one parameter, and the second instance has two parameters, where the second parameter has a default value. When we call Add with just one parameter, the compiler cannot decide whether we intend to call

• the first instance of Add with one parameter

or

• the second instance of Add using the default value for the second parameter.

In this specific case, there are multiple options to solve the issue, but all of them involve redesigning the package specification:

26.3. Subprograms
Learning Ada

- we could just rename one of Add procedures (thereby eliminating the subprogram overloading);
- we could rename the first parameter of one of the Add procedures and use named parameter association in the call to the procedure;
  - For example, we could rename the parameter to Value and call Add (Value => A).
- remove the default value from the second parameter of the second instance of Add.

Overlapping actual parameters

When we have more than one out or in out parameters in a subprogram, we might run into the situation where the actual parameter overlaps with another parameter. For example:

Listing 74: machine_x.ads

```ada
package Machine_X is

  procedure Update_Value (V1 : in out Integer;
                          V2 :     out Integer);

end Machine_X;
```

Listing 75: machine_x.adb

```ada
package body Machine_X is

  procedure Update_Value (V1 : in out Integer;
                          V2 :     out Integer) is
  begin
    V1 := V1 + 1;
    V2 := V2 + 1;
  end Update_Value;

end Machine_X;
```

Listing 76: show_by_copy_by_ref_params.adb

```ada
with Machine_X; use Machine_X;

procedure Show_By_Copy_By_Ref_Params is
  A : Integer := 5;
begin
  Update_Value (A, A);
end Show_By_Copy_By_Ref_Params;
```

Build output

```
show_by_copy_by_ref_params.adb:6:18: error: writable actual for "V1" overlaps with actual for "V2"
gprbuild: *** compilation phase failed
```
In this case, we’re using \texttt{A} for both output parameters in the call to \texttt{Update_Value}. Passing one variable to more than one output parameter in a given call is forbidden in Ada, so this triggers a compilation error. Depending on the specific context, you could solve this issue by using temporary variables for the other output parameters.

### 26.3.2 Operators

Operators are commonly used for variables of scalar types such as \texttt{Integer} and \texttt{Float}. In these cases, they replace \textit{usual} function calls. (To be more precise, operators are function calls, but written in a different format.) For example, we simply write \texttt{A := A + B + C}; when we want to add three integer variables. A hypothetical, non-intuitive version of this operation could be \texttt{A := Add (Add (A, B), C)}; in such cases, operators allow for expressing function calls in a more intuitive way.

Many primitive operators exist for scalar types. We classify them as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>\texttt{and, or, xor}</td>
</tr>
<tr>
<td>Relational</td>
<td>\texttt{=, /=, &lt;, &lt;=, &gt;, &gt;=}</td>
</tr>
<tr>
<td>Unary adding</td>
<td>\texttt{+, -}</td>
</tr>
<tr>
<td>Binary adding</td>
<td>\texttt{+, -, &amp;, &amp;}</td>
</tr>
<tr>
<td>Multiplying</td>
<td>\texttt{*}, \texttt{/}, \texttt{mod}, \texttt{rem}</td>
</tr>
<tr>
<td>Highest precedence</td>
<td>\texttt{**, abs, not}</td>
</tr>
</tbody>
</table>

---

**In the Ada Reference Manual**

- 4.5 Operators and Expression Evaluation\textsuperscript{137}

---

**User-defined operators**

For non-scalar types, not all operators are defined. For example, it wouldn't make sense to expect a compiler to include an addition operator for a record type with multiple components. Exceptions to this rule are the equality and inequality operators (\texttt{=} and \texttt{/=}), which are defined for any type (be it scalar, record types, and array types).

For array types, the concatenation operator (\&\&) is a primitive operator:

Listing 77: integer_arrays.ads

```ada
package Integer_Arrays is
type Integer_Array is
  array (Positive range <>) of Integer;
end Integer_Arrays;
```

Listing 78: show_array_concatenation.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Integer_Arrays; use Integer_Arrays;
procedure Show_Array_Concatenation is
(continues on next page)
```

\textsuperscript{137} http://www.ada-auth.org/standards/22rm/html/RM-4-5.html
In this example, we're using the primitive & operator to concatenate the A and B arrays in the assignment to R. Similarly, we're concatenating individual components (integer values) to create an aggregate that we assign to A and B.

In contrast to this, the addition operator is not available for arrays:
We can, however, define custom operators for any type. For example, if a specific type doesn't have a predefined addition operator, we can define our own + operator for it.

Note that we're limited to the operator symbols that are already defined by the Ada language (see the previous table for the complete list of operators). In other words, the operator we define must be selected from one of those existing symbols; we cannot use new symbols for custom operators.

### In other languages
Some programming languages — such as Haskell — allow you to define and use custom operator symbols. For example, in Haskell, you can create a new "broken bar" (¦) operator for integer values:

```haskell
(¦) :: Int -> Int -> Int
a ¦ b = a + a + b
main = putStrLn $ show (2 ¦ 3)
```

This is not possible in Ada.

Let's define a custom addition operator that adds individual components of the Integer_Array type:

```ada
package Integer_Arrays is

  type Integer_Array is
    array (Positive range <>) of Integer;

  function "+" (Left, Right : Integer_Array)
    return Integer_Array
  with Post =>
    (for all I in "+"'Result'Range =>
    "+"'Result (I) = Left (I) + Right (I));

end Integer_Arrays;
```

```ada
package body Integer_Arrays is

  function "+" (Left, Right : Integer_Array)
    return Integer_Array
  is
    R : Integer_Array (Left'Range);
    begin
      for I in Left'Range loop
        R (I) := Left (I) + Right (I);
      end loop;
```

(continues on next page)
Learning Ada

(continued from previous page)

```
11 return R;
12 end "+";
13 end Integer_Arrays;
```

Listing 83: show_array_addition.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Addition is
  A, B, R : Integer_Array (1 .. 5);
begin
  A := (1 & 2 & 3 & 4 & 5);
  B := (6 & 7 & 8 & 9 & 10);
  R := A + B;
  for E of R loop
    Put (E'Image & ' ');
  end loop;
  New_Line;
end Show_Array_Addition;
```

Code block metadata

- MD5: 6f50fa47270d97d3fb50379b6275777d

Runtime output

```
7 9 11 13 15
```

Now, the R := A + B line doesn't trigger a compilation error anymore because the + operator is defined for the Integer_Array type.

In the implementation of the +, we return an array with the range of the Left array where each component is the sum of the Left and Right arrays. In the declaration of the + operator, we're defining the expected behavior in the postcondition. Here, we're saying that, for each index of the resulting array (for all I in "+"'Result'Range), the value of each component of the resulting array at that specific index is the sum of the components from the Left and Right arrays at the same index ("+"'Result (I) = Left (I) + Right (I)). (for all denotes a quantified expression (page 590).)

Note that, in this implementation, we assume that the range of Right is a subset of the range of Left. If that is not the case, the Constraint_Error exception will be raised at runtime in the loop. (You can test this by declaring B as Integer_Array (5 .. 10), for example.)

We can also define custom operators for record types. For example, we could declare two + operators for a record containing the name and address of a person:

```
package Addresses is
  type Person is private;
  function "+" (Name : String;
```

Listing 84: addresses.ads
Listing 85: addresses.adb

```ada
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO; use Ada.Text_IO;

package body Addresses is

function "+" (Name : String;
  Address : String)
  return Person
is
begin
  return (Name => Head (Name,
    Name_String'Length),
  Address => Head (Address,
    Address_String'Length));
end "+";

function "+" (Left, Right : Person)
  return Person
is
begin
  return (Name => Left.Name,
  Address => Right.Address);
end "+";

procedure Display (P : Person) is
begin
  Put_Line ("Name: " & P.Name);
  Put_Line ("Address: " & P.Address);
  New_Line;
end Display;
end Addresses;
```

Listing 86: show_address_addition.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Addresses; use Addresses;
```

26.3. Subprograms
procedure Show_Address_Addition is
  John : Person := "John" + "4 Main Street";
  Jane : Person := "Jane" + "7 High Street";
begin
  Display (John);
  Display (Jane);
  Put_Line ("----------------");
  Jane := Jane + John;
  Display (Jane);
end Show_Address_Addition;

In this example, the first + operator takes two strings — with the name and address of a person — and returns an object of Person type. We use this operator to initialize the John and Jane variables.

The second + operator in this example brings two people together. Here, the person on the left side of the + operator moves to the home of the person on the right side. In this specific case, Jane is moving to John's house.

As a small remark, we usually expect that the + operator is commutative. In other words, changing the order of the elements in the operation doesn't change the result. However, in our definition above, this is not the case, as we can confirm by comparing the operation in both orders:

Listing 87: show_address_addition.adb

with Ada.Text_IO; use Ada.Text_IO;
with Addresses; use Addresses;

procedure Show_Address_Addition is
  John : constant Person := "John" + "4 Main Street";
  Jane : constant Person := "Jane" + "7 High Street";
begin
  if Jane + John = John + Jane then
    Put_Line ("It's commutative!");
  else
    Put_Line ("It's not commutative!");
  end if;
end Show_Address_Addition;
In this example, we're using the primitive = operator for the Person to assess whether the result of the addition is commutative.

In the Ada Reference Manual

- 6.1 Subprogram Declarations

26.3.3 Expression functions

Usually, we implement Ada functions with a construct like this: begin return X; end; In other words, we create a begin ... end; block and we have at least one return statement in that block. An expression function, in contrast, is a function that is implemented with a simple expression in parentheses, such as (X);. In this case, we don't use a begin ... end; block or a return statement.

As an example of an expression, let's say we want to implement a function named Is_Zero that checks if the value of the integer parameter I is zero. We can implement this function with the expression I = 0. In the usual approach, we would create the implementation by writing is begin return I = 0; end Is_Zero;. When using expression functions, however, we can simplify the implementation by just writing is (I = 0);. This is the complete code of Is_Zero using an expression function:

```
package Expr_Func is
  function Is_Zero (I : Integer) return Boolean is
    (I = 0);
  end Expr_Func;
end
```

An expression function has the same effect as the usual version using a block. In fact, the code above is similar to this implementation of the Is_Zero function using a block:

```
package Expr_Func is
  function Is_Zero (I : Integer) return Boolean;
end
```

(continues on next page)
The only difference between these two versions of the Expr_Func packages is that, in the first version, the package specification contains the implementation of the Is_Zero function, while, in the second version, the implementation is in the body of the Expr_Func package.

An expression function can be, at same time, the specification and the implementation of a function. Therefore, in the first version of the Expr_Func package above, we don't have a separate implementation of the Is_Zero function because \((I = 0)\) is the actual implementation of the function. Note that this is only possible for expression functions; you cannot have a function implemented with a block in a package specification. For example, the following code is wrong and won't compile:

```
package Expr_Func is

    function Is_Zero (I : Integer) return Boolean is
        begin
            return I = 0;
        end Is_Zero;

end Expr_Func;
```

We can, of course, separate the function declaration from its implementation as an expression function. For example, we can rewrite the first version of the Expr_Func package and move the expression function to the body of the package:

```
package Expr_Func is

    function Is_Zero (I : Integer)

end Expr_Func;
```
Learning Ada

(continued from previous page)

```ada
return Boolean;
end Expr_Func;
```

Listing 93: expr_func.adb

```ada
package body Expr_Func is
  function Is_Zero (I : Integer) return Boolean is
    (I = 0);
end Expr_Func;
```

Code block metadata

MD5: 491a491da92636a35579f870969aaf08

In addition, we can use expression functions in the private part of a package specification. For example, the following code declares the Is_Valid function in the specification of the My_Data package, while its implementation is an expression function in the private part of the package specification:

```ada
package My_Data is
  type Data is private;

  function Is_Valid (D : Data) return Boolean;

private

  type Data is record
    Valid : Boolean;
  end record;

  function Is_Valid (D : Data) return Boolean is
    (D.Valid);
end My_Data;
```

Code block metadata

MD5: beb57eca67b3954097e0f7ac00ea70c9

Naturally, we could write the function implementation in the package body instead:

```ada
package My_Data is
  type Data is private;

  function Is_Valid (D : Data) return Boolean;
end My_Data;
```

(continues on next page)
function Is_Valid (D : Data) return Boolean;

private

type Data is record
   Valid : Boolean;
end record;
end My_Data;

Listing 96: my_data.adb

package body My_Data is

   function Is_Valid (D : Data) return Boolean is
      (D.Valid);
   end My_Data;

end My_Data;

Code block metadata


Private Expression Function 2

MD5: 3c6e2a3c53c7c8e1a7b86efccdc3bf8d

In the Ada Reference Manual

- 6.8 Expression functions

26.3.4 Overloading

Note: This section was originally written by Robert A. Duff and published as Gem #50: Overload Resolution.

Ada allows overloading of subprograms, which means that two or more subprogram declarations with the same name can be visible at the same place. Here, "name" can refer to operator symbols, like "+". Ada also allows overloading of various other notations, such as literals and aggregates.

In most languages that support overloading, overload resolution is done "bottom up" — that is, information flows from inner constructs to outer constructs. As usual, computer folks draw their trees upside-down, with the root at the top. For example, if we have two procedures Print:

Listing 97: show_overloading.adb

procedure Show_Overloading is

   package Types is
      type Sequence is null record;
   end Types;

   procedure Print (S : Sequence) is

end Show_Overloading;

https://www.adacore.com/gems/gem-50
type Set is null record;

procedure Print (S : Sequence) is null;
procedure Print (S : Set) is null;
end Types;

use Types;

X : Sequence;
begin
  -- Compiler selects Print (S : Sequence)
  Print (X);
end Show_Overloading;

Code block metadata

MD5: 020c4f04285c80c1050d8edbaa2dbca

the type of X determines which Print is meant in the call.
Ada is unusual in that it supports top-down overload resolution as well:

Listing 98: show_top_down_overloading.adb

procedure Show_Top_Down_Overloading is

package Types is
  type Sequence is null record;
  type Set is null record;

  function Empty return Sequence is
    ((others => <>));

  function Empty return Set is
    ((others => <>));

  procedure Print_Sequence (S : Sequence) is
    null;

  procedure Print_Set (S : Set) is
    null;
end Types;

use Types;

X : Sequence;
begin
  -- Compiler selects function
  -- Empty return Sequence
  -- Print_Sequence (Empty);
end Show_Top_Down_Overloading;

Code block metadata

MD5: 3b776a3efdee3d7e583dddbf5159c9a1b

The type of the formal parameter S of Print_Sequence determines which Empty is meant in the call. In C++, for example, the equivalent of the Print (X) example would resolve, but the Print_Sequence (Empty) would be illegal, because C++ does not use top-down
If we overload things too heavily, we can cause ambiguities:

Listing 99: show_overloading_error.adb

```ada
procedure Show_Overloading_Error is
package Types is
type Sequence is null record;
type Set is null record;
function Empty return Sequence is
  ((others => <>));
function Empty return Set is
  ((others => <>));
procedure Print (S : Sequence) is
  null;
procedure Print (S : Set) is
  null;
end Types;
use Types;
x : Sequence;
beg
  Print (Empty); -- Illegal!
end Show_Overloading_Error;
```

The call is ambiguous, and therefore illegal, because there are two possible meanings. One way to resolve the ambiguity is to use a qualified expression to say which type we mean:

Print (Sequence'(Empty));

Note that we're now using both bottom-up and top-down overload resolution: Sequence' determines which Empty is meant (top down) and which Print is meant (bottom up). You can qualify an expression, even if it is not ambiguous according to Ada rules — you might want to clarify the type because it might be ambiguous for human readers.

Of course, you could instead resolve the Print (Empty) example by modifying the source code so the names are unique, as in the earlier examples. That might well be the best solution, assuming you can modify the relevant sources. Too much overloading can be confusing. How much is "too much" is in part a matter of taste.

Ada really needs to have top-down overload resolution, in order to resolve literals. In some
languages, you can tell the type of a literal by looking at it, for example appending L (letter l) means "the type of this literal is long int". That sort of kludge won't work in Ada, because we have an open-ended set of integer types:

Listing 100: show_literal_resolution.adb

```ada
procedure Show_Literal_Resolution is
  type Apple_Count is range 0 .. 100;
  procedure Peel (Count : Apple_Count) is null;
begin
  Peel (20);
end Show_Literal_Resolution;
```

You can't tell by looking at the literal 20 what its type is. The type of formal parameter Count tells us that 20 is an Apple_Count, as opposed to some other type, such as Standard.Long_Integer.

Technically, the type of 20 is universal_integer, which is implicitly converted to Apple_Count — it's really the result type of that implicit conversion that is at issue. But that's an obscure point — you won't go too far wrong if you think of the integer literal notation as being overloaded on all integer types.

Developers sometimes wonder why the compiler can't resolve something that seems obvious. For example:

Listing 101: show_literal_resolution_error.adb

```ada
procedure Show_Literal_Resolution_Error is
  type Apple_Count is range 0 .. 100;
  procedure Slice (Count : Apple_Count) is null;
  type Orange_Count is range 0 .. 10_000;
  procedure Slice (Count : Orange_Count) is null;
begin
  Slice (Count => (10_000)); -- Illegal!
end Show_Literal_Resolution_Error;
```

This call is ambiguous, and therefore illegal. But why? Clearly the developer must have meant the Orange_Count one, because 10_000 is out of range for Apple_Count. And all the relevant expressions happen to be static.
Well, a good rule of thumb in language design (for languages with overloading) is that the overload resolution rules should not be "too smart". We want this example to be illegal to avoid confusion on the part of developers reading the code. As usual, a qualified expression fixes it:

\[
\text{Slice (Count} \Rightarrow \text{Orange\_Count'(10\_000))};
\]

Another example, similar to the literal, is the aggregate. Ada uses a simple rule: the type of an aggregate is determined top down (i.e., from the context in which the aggregate appears). Bottom-up information is not used; that is, the compiler does not look inside the aggregate in order to determine its type.

Listing 102: show\_record\_resolution\_error.adb

```ada
procedure Show\_Record\_Resolution\_Error is
  type Complex is record
    Re, Im : Float;
  end record;

  procedure Grind (X : Complex) is null;
  procedure Grind (X : String) is null;

begin
  Grind (X => (Re => 1.0, Im => 1.0));
  -- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ illegal!
end Show\_Record\_Resolution\_Error;
```

There are two \texttt{Grind} procedures visible, so the type of the aggregate could be \texttt{Complex} or \texttt{String}, so it is ambiguous and therefore illegal. The compiler is not required to notice that there is only one type with components \texttt{Re} and \texttt{Im}, of some real type — in fact, the compiler is not \textit{allowed} to notice that, for overloading purposes.

We can qualify as usual:

\[
\text{Grind (X} \Rightarrow \text{Complex'((Re} => \text{1.0}, \text{Im} => \text{1.0}))};
\]

Only after resolving that the type of the aggregate is \texttt{Complex} can the compiler look inside and make sure \texttt{Re} and \texttt{Im} make sense.

This not-too-smart rule for aggregates helps prevent confusion on the part of developers reading the code. It also simplifies the compiler, and makes the overload resolution algorithm reasonably efficient.
26.3.5 Operator Overloading

We've seen previously (page 631) that we can define custom operators for any type. We've also seen that subprograms can be overloaded (page 640). Since operators are functions, we're essentially talking about operator overloading, as we're defining the same operator (say + or -) for different types.

As another example of operator overloading, in the Ada standard library, operators are defined for the Complex type of the Ada.Numerics.Generic_Complex_Types package. This package contains not only the definition of the + operator for two objects of Complex type, but also for combination of Complex and other types. For instance, we can find these declarations:

```ada
function "+" (Left, Right : Complex) return Complex;
function "+" (Left : Complex; Right : Real'Base) return Complex;
function "+" (Left : Real'Base; Right : Complex) return Complex;
```

This example shows that the + operator — as well as other operators — are being overloaded in the Generic_Complex_Types package.

In the Ada Reference Manual

- 6.6 Overloading of Operators\(^{141}\)
- G.1.1 Complex Types\(^{142}\)

26.3.6 Operator Overriding

We can also override operators of derived types. This allows for modifying the behavior of operators for the corresponding derived types.

To override an operator of a derived type, we simply implement a function for that operator. This is the same as how we implement custom operators (as we've seen previously).

As an example, when adding two fixed-point values, the result might be out of range, which causes an exception to be raised. A common strategy to avoid exceptions in this case is to saturate the resulting value. This strategy is typically employed in signal processing algorithms, for example.

In this example, we declare and use the 32-bit fixed-point type TQ31:

```ada
package Fixed_Point is
   D : constant := 2.0 ** (-31);
   type TQ31 is delta D range -1.0 .. 1.0 - D;
end Fixed_Point;
```

\(^{141}\) http://www.ada-auth.org/standards/22rm/html/RM-6-6.html
\(^{142}\) http://www.ada-auth.org/standards/22rm/html/RM-G-1-1.html
Learning Ada

Listing 104: show_sat_op.adb

with Ada.Text_IO; use Ada.Text_IO;
with Fixed_Point; use Fixed_Point;

procedure Show_Sat_Op is
  A, B, C : TQ31;
begin
  A := TQ31'Last;
  B := TQ31'Last;
  C := A + B;
  Put_Line (A'Image & " + "
            & B'Image & " = "
            & C'Image);
  A := TQ31'First;
  B := TQ31'First;
  C := A + B;
  Put_Line (A'Image & " + "
            & B'Image & " = "
            & C'Image);
end Show_Sat_Op;

Code block metadata

MD5: 15d8860773ec7c0e505d0ee94781ae14

Runtime output

raised CONSTRAINT_ERROR : show_sat_op.adb:9 overflow check failed

Here, we're using the standard + operator, which raises a Constraint_Error exception in the C := A + B; statement due to an overflow. Let's now override the addition operator and enforce saturation when the result is out of range:

Listing 105: fixed_point.ads

package Fixed_Point is
  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;
  function "+" (Left, Right : TQ31)
    return TQ31;
end Fixed_Point;

Listing 106: fixed_point.adb

package body Fixed_Point is
  function "+" (Left, Right : TQ31)
    return TQ31
  is
    type TQ31_2 is
      delta TQ31'Delta
  (continues on next page)
Learning Ada

range TQ31'First * 2.0 .. TQ31'Last * 2.0;

L : constant TQ31_2 := TQ31_2 (Left);
R : constant TQ31_2 := TQ31_2 (Right);
Res : TQ31_2;
begin
  Res := L + R;
  if Res > TQ31_2 (TQ31'Last) then
    return TQ31'Last;
  elsif Res < TQ31_2 (TQ31'First) then
    return TQ31'First;
  else
    return TQ31 (Res);
  end if;
end "+";
end Fixed_Point;

Listing 107: show_sat_op.adb

with Ada.Text_IO; use Ada.Text_IO;
with Fixed_Point; use Fixed_Point;
procedure Show_Sat_Op is
  A, B, C : TQ31;
begin
  A := TQ31'Last;
  B := TQ31'Last;
  C := A + B;
  Put_Line (A'Image & " + "
                & B'Image & " = "
                & C'Image);
  A := TQ31'First;
  B := TQ31'First;
  C := A + B;
  Put_Line (A'Image & " + "
                & B'Image & " = "
                & C'Image);
end Show_Sat_Op;

Code block metadata

MD5: 6317bcf9c278c01f86dbdc761d86240

Runtime output

0.9999999995 + 0.9999999995 = 0.9999999995
-1.0000000000 + -1.0000000000 = -1.0000000000

In the implementation of the overridden + operator of the TQ31 type, we declare another type (TQ31_2) with a wider range than TQ31. We use variables of the TQ31_2 type to perform the actual addition, and then we verify whether the result is still in TQ31's range. If it is, we simply convert the result back to the TQ31 type. Otherwise, we saturate it — using either the first or last value of the TQ31 type.

26.3. Subprograms
When overriding operators, the overridden operator replaces the original one. For example, in the \( A + B \) operation of the `Show_Sat_Op` procedure above, we’re using the overridden version of the `+` operator, which performs saturation. Therefore, this operation doesn’t raise an exception (as it was the case with the original `+` operator).

### 26.3.7 Nonreturning procedures

Usually, when calling a procedure \( P \), we expect that it returns to the caller’s *thread of control* after performing some action in the body of \( P \). However, there are situations where a procedure never returns. We can indicate this fact by using the `No_Return` aspect in the subprogram declaration.

A typical example is that of a server that is designed to run forever until the process is killed or the machine where the server runs is switched off. This server can be implemented as an endless loop. For example:

```ada
package Servers is
    procedure Run_Server
        with No_Return;
end Servers;
```

```ada
package body Servers is
    procedure Run_Server is
        begin
            pragma Warnings (Off, "implied return after this statement");
            while True loop
                -- Processing happens here...
                null;
            end loop;
            end Run_Server;
        end Servers;
```

```ada
with Servers; use Servers;
procedure Show_Endless_Loop is
    begin
        Run_Server;
    end Show_Endless_Loop;
```

In this example, `Run_Server` doesn't exit from the `while True` loop, so it never returns to the `Show_Endless_Loop` procedure.
The same situation happens when we call a procedure that raises an exception unconditionally. In that case, exception handling is triggered, so that the procedure never returns to the caller. An example is that of a logging procedure that writes a message before raising an exception internally:

```ada
package Loggers is

 Logged_Failure : exception;

  procedure Log_And_Raise (Msg : String)
  with No_Return;

end Loggers;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Loggers is

  procedure Log_And_Raise (Msg : String) is
  begin
    Put_Line (Msg);
    raise Logged_Failure;
  end Log_And_Raise;

end Loggers;
```

```ada
with Ada.Text_IO; use Ada.Text_IO; with Loggers; use Loggers;

procedure Show_No_Return_Exception is
  Check_Passed : constant Boolean := False;
begin
  if not Check_Passed then
    Log_And_Raise ("Check failed!");
    Put_Line ("This line will not be reached!");
  end if;
end Show_No_Return_Exception;
```

In this example, Log_And_Raise writes a message to the user and raises the Logged_Failure, so it never returns to the Show_No_Return_Exception procedure.

We could implement exception handling in the Show_No_Return_Exception procedure, so that the Logged_Failure exception could be handled there after it's raised in Log_And_Raise. However, this wouldn't be considered a normal return to the procedure because it wouldn't return to the point where it should (i.e. to the point where Put_Line is about to be called, right after the call to the Log_And_Raise procedure).

If a nonreturning procedure returns nevertheless, this is considered a program error, so that the Program_Error exception is raised. For example:

```
26.3. Subprograms
```
### Listing 114: loggers.ads

```ada
package Loggers is
 Logged_Failure : exception;

procedure Log_And_Raise (Msg : String) with No_Return;
end Loggers;
```

### Listing 115: loggers.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Loggers is
  procedure Log_And_Raise (Msg : String) is
  begin
    Put_Line (Msg);
  end Log_And_Raise;
end Loggers;
```

### Listing 116: show_no_return_exception.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Loggers; use Loggers;

procedure Show_No_Return_Exception is
  Check_Passed : constant Boolean := False;
  begin
    if not Check_Passed then
      Log_And_Raise ("Check failed!");
      Put_Line ("This line will not be reached!");
    end if;
  end Show_No_Return_Exception;
```

---

#### Code block metadata


**Erroneous Log Exception**

**MD5:** e44fd8df0529dda5749e85b9e300a999

---

#### Build output

loggers.adb:7:07: warning: implied return after this statement will raise Program_Error [enabled by default]

loggers.adb:7:07: warning: procedure "Log_And_Raise" is marked as No_Return [enabled by default]

---

#### Runtime output

Check failed!

raised PROGRAM_ERROR : loggers.adb:7 implicit return with No_Return

---

Here, Program_Error is raised when Log_And_Raise returns to the Show_No_Return_Exception procedure.

---

#### In the Ada Reference Manual

650 Chapter 26. Control Flow
26.3.8 Inline subprograms

Inlining\(^{144}\) refers to a kind of optimization where the code of a subprogram is expanded at the point of the call in place of the call itself.

In modern compilers, inlining depends on the optimization level selected by the user. For example, if we select the higher optimization level, the compiler will perform automatic inlining aggressively.

**In the GNAT toolchain**

The highest optimization level (-O3) of GNAT performs aggressive automatic inlining. This could mean that this level inlines too much rather than not enough. As a result, the cache may become an issue and the overall performance may be worse than the one we would achieve by compiling the same code with optimization level 2 (-O2). Therefore, the general recommendation is to not just select -O3 for the optimized version of an application, but instead compare it the optimized version built with -O2.

It's important to highlight that the inlining we're referring above happens automatically, so the decision about which subprogram is inlined depends entirely on the compiler. However, in some cases, it's better to reduce the optimization level and perform manual inlining instead of automatic inlining. We do that by using the Inline aspect.

Let's look at this example:

**Listing 117: float_arrays.ads**

```ada
package Float_Arrays is

  type Float_Array is array (Positive range <>) of Float;

  function Average (Data : Float_Array) return Float
    with Inline;

end Float_Arrays;
```

**Listing 118: float_arrays.adb**

```ada
package body Float_Arrays is

  function Average (Data : Float_Array) return Float
    is
      Total : Float := 0.0;
    begin
      for Value of Data loop
        Total := Total + Value;
      end loop;
      return Total / Float (Data'Length);
      end Average;

end Float_Arrays;
```

\(^{143}\) [http://www.ada-auth.org/standards/22rm/html/RM-6-5-1.html](http://www.ada-auth.org/standards/22rm/html/RM-6-5-1.html)

\(^{144}\) [https://en.wikipedia.org/wiki/Inline_expansion](https://en.wikipedia.org/wiki/Inline_expansion)
```ada
with Ada.Text_IO; use Ada.Text_IO;

with Float_Arrays; use Float_Arrays;

procedure Compute_Average is
   Values : constant Float_Array :=
     (10.0, 11.0, 12.0, 13.0);
   Average_Value : Float;
begin
   Average_Value := Average (Values);
   Put_Line ("Average = "
             & Float'Image (Average_Value));
end Compute_Average;
```

### Code block metadata

MD5: 246bc11e8a969d69873f416f583f450e

### Runtime output

Average = 1.15000E+01

When compiling this example, the compiler will most probably inline `Average` in the `Compute_Average` procedure. Note, however, that the Inline aspect is just a recommendation to the compiler. Sometimes, the compiler might not be able to follow this recommendation, so it won't inline the subprogram.

These are some examples of situations where the compiler might not be able to inline a subprogram:

- when the code is too large,
- when it's too complicated — for example, when it involves exception handling —, or
- when it contains tasks, etc.

### In the GNAT toolchain

In order to effectively use the Inline aspect, we need to set the optimization level to at least `-O1` and use the `-gnatn` switch, which instructs the compiler to take the Inline aspect into account.

In addition to the Inline aspect, in GNAT, we also have the (implementation-defined) Inline_Always aspect. In contrast to the former aspect, however, the Inline_Always aspect isn't primarily related to performance. Instead, it should be used when the functionality would be incorrect if inlining was not performed by the compiler. Examples of this are procedures that insert Assembly instructions that only make sense when the procedure is inlined, such as memory barriers.

Similar to the Inline aspect, there might be situations where a subprogram has the Inline_Always aspect, but the compiler is unable to inline it. In this case, we get a compilation error from GNAT.

Note that we can use the Inline aspect for generic subprograms as well. When we do this, we indicate to the compiler that we wish it inlines all instances of that generic subprogram.

### In the Ada Reference Manual

652
26.3.9 Null Procedures

Null procedures are procedures that don't have any effect, as their body is empty. We declare a null procedure by simply writing `is null` in its declaration. For example:

```ada
package Null_procs is

    procedure Do_Nothing (Msg : String) is null;

end Null_procs;
```

As expected, calling a null procedure doesn't have any effect. For example:

```ada
with Null_procs; use Null_procs;

procedure Show_Null_Proc is
begin
    Do_Nothing ("Hello");
end Show_Null_Proc;
```

Null procedures are equivalent to implementing a procedure with a body that only contains `null`. Therefore, the `Do_Nothing` procedure above is equivalent to this:

```ada
package Null_procs is

    procedure Do_Nothing (Msg : String);

end Null_procs;
```

```ada
package body Null_procs is

    procedure Do_Nothing (Msg : String) is
        begin
            null;
        end Do_Nothing;

end Null_procs;
```

145 http://www.ada-auth.org/standards/22rm/html/RM-6-3-2.html
Null procedures and overriding

We can use null procedures as a way to simulate interfaces for non-tagged types — similar to what actual interfaces do for tagged types. For example, we may start by declaring a type and null procedures that operate on that type. For example, let’s model a very simple API:

```adacode
package Simple_Storage is
  type Storage_Model is null record;
  procedure Set (S : in out Storage_Model; V : String) is null;
  procedure Display (S : Storage_Model) is null;
end Simple_Storage;
```

Here, the API of the `Storage_Model` type consists of the `Set` and `Display` procedures. Naturally, we can use objects of the `Storage_Model` type in an application, but this won’t have any effect:

```adacode
procedure Show_Null_Proc is
  S : Storage_Model;
begin
  Put_Line ("Setting 24...");
  Set (S, "24");
  Display (S);
end Show_Null_Proc;
```

By itself, the `Storage_Model` type is not very useful. However, we can derive other types from it and override the null procedures. Let’s say we want to implement the `Integer_Storage` type to store an integer value:

```adacode
package Integer_Storage is
  type Integer_Model is
    new private with record
      Value : Integer;
  procedure Set (I : Integer_Model; V : Integer) is new private (I.Value := V);
  procedure Display (I : Integer_Model) is new private (Put (I.Value));
end Integer_Storage;
```
Listing 126: simple_storage.ads

```ada
package Simple_Storage is

  type Storage_Model is null record;

  procedure Set (S : in out Storage_Model; V : String) is null;
  procedure Display (S : Storage_Model) is null;

  type Integer_Storage is private;

  procedure Set (S : in out Integer_Storage; V : String);
  procedure Display (S : Integer_Storage);

private

  type Integer_Storage is record
    V : Integer := 0;
  end record;

end Simple_Storage;
```

Listing 127: simple_storage.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Simple_Storage is

  procedure Set (S : in out Integer_Storage; V : String) is
  begin
    S.V := Integer'Value (V);
  end Set;

  procedure Display (S : Integer_Storage) is
  begin
    Put_Line ("Value: " & S.V'Image);
  end Display;

end Simple_Storage;
```

Listing 128: show_null_proc.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Simple_Storage; use Simple_Storage;

procedure Show_NullProc is
  S : Integer_Storage;
begin
  Put_Line ("Setting 24...");
  Set (S, "24");
  Display (S);
end Show_NullProc;
```

Code block metadata

MD5: 55d491d1ef72fb7be2bf0d2a212f335b
In this example, we can view Storage_Model as a sort of interface for derived non-tagged types, while the derived types — such as Integer_Storage — provide the actual implementation.

The section on null records (page 420) contains an extended example that makes use of null procedures.

### 26.4 Exceptions

#### 26.4.1 Asserts

When we want to indicate a condition in the code that must always be valid, we can use the pragma Assert. As the name implies, when we use this pragma, we're asserting some truth about the source-code. (We can also use the procedural form, as we'll see later.)

**Important**

Another method to assert the truth about the source-code is to use pre and post-conditions.

A simple assert has this form:

```
procedure Show_Pragma_Assert is
  I : constant Integer := 10;
begin
  pragma Assert (I = 10);
end Show_Pragma_Assert;
```

In this example, we're asserting that the value of I is always 10. We could also display a message if the assertion is false:

```
procedure Show_Pragma_Assert is
  I : constant Integer := 11;
begin
  pragma Assert (I = 10, "I is not 10");
```

(continues on next page)
null;
end Show_Pragma_Assert;

Code block metadata
MD5: b70fa67c92542ade39c388964ce12302

Build output
show_pragma_assert.adb:4:19: warning: assertion will fail at run time [-gnatw.a]

Runtime output
raised ADA ASSERTIONS ASSERTION_ERROR : I is not 10

Similarly, we can use the procedural form of Assert. For example, the code above can implemented as follows:

Listing 131: show_procedure_assert.adb

with Ada.Assertions; use Ada.Assertions;

procedure Show_Procedure_Assert is
   I : constant Integer := 11;
begin
   Assert (I = 10, "I is not 10");
end Show_Procedure_Assert;

Code block metadata
MD5: cbab23645ff89d4adffcaaddaeb6f0e3

Runtime output
raised ADA ASSERTIONS ASSERTION_ERROR : I is not 10

Note that a call to Assert is simply translated to a check — and the Assertion_Error exception from the Ada.Assertions package being raised in the case that the check fails. For example, the code above roughly corresponds to this:

Listing 132: show_assertion_error.adb

with Ada.Assertions; use Ada.Assertions;

procedure Show_Assertion_Error is
   I : constant Integer := 11;
begin
   if I /= 10 then
      raise Assertion_Error with "I is not 10";
   end if;
end Show_Assertion_Error;

Code block metadata

26.4. Exceptions
26.4.2 Assertion policies

We can activate and deactivate assertions based on assertion policies. We can do that by using the pragma `Assertion_Policy`. As an argument to this pragma, we indicate whether a specific policy must be checked or ignored.

For example, we can deactivate assertion checks by specifying `Assert => Ignore`. Similarly, we can activate assertion checks by specifying `Assert => Check`. Let’s see a code example:

```ada
procedure Show_Pragma_Assertion_Policy is
  I : constant Integer := 11;
pragma Assertion_Policy (Assert => Ignore);
begin
  pragma Assert (I = 10);
end Show_Pragma_Assertion_Policy;
```

Here, we’re specifying that asserts shall be ignored. Therefore, the call to the pragma `Assert` doesn’t raise an exception. If we replace `Ignore` with `Check` in the call to `Assertion_Policy`, the assert will raise the `Assertion_Error` exception.

The following table presents all policies that we can set:

---

147 http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html
### In the GNAT toolchain

Compilers are free to include policies that go beyond the ones listed above. For example, GNAT includes the following policies — called assertion kinds in this context:

- Assertions
- Assert_And_Cut
- Assume
- Contract_Cases
- Debug
- Ghost
- Initial_Condition
- Invariant
- Invariant'Class
- Loop_Invariant
- Loop_Variant
- Postcondition
- Precondition
- Predicate
- Refined_Post
- Statement_Assertions
- Subprogram_Variant

Also, in addition to Check and Ignore, GNAT allows you to set a policy to Disable and Suppressible.

You can read more about them in the [GNAT Reference Manual][148].

You can specify multiple policies in a single call to Assertion_Policy. For example, you can activate all policies by writing:

---

[148] https://gcc.gnu.org/onlinedocs/gnat_rm/Pragma-Assertion_005fPolicy.html
Listing 134: show_multiple_assertion_policies.adb

```ada
procedure Show_Multiple_Assertion_Policies is
  pragma Assertion_Policy
  (Assert => Check,
   Static_Predicate => Check,
   Dynamic_Predicate => Check,
   Pre => Check,
   Pre'Class => Check,
   Post => Check,
   Post'Class => Check,
   Type_Invariant => Check,
   Type_Invariant'Class => Check);
begin
  null;
end Show_Multiple_Assertion_Policies;
```

In the GNAT toolchain

With GNAT, policies can be specified in multiple ways. In addition to calls to `Assertion_Policy`, you can use configuration pragmas files. You can use these files to specify all pragmas that are relevant to your application, including `Assertion_Policy`. In addition, you can manage the granularity for those pragmas. For example, you can use a global configuration pragmas file for your complete application, or even different files for each source-code file you have.

Also, by default, all policies listed in the table above are deactivated, i.e. they're all set to Ignore. You can use the command-line switch `-gnata` to activate them.

Note that the Assert procedure raises an exception independently of the assertion policy (`Assertion_Policy (Assert => Ignore)`). For example:

Listing 135: show_assert_procedure_policy.adb

```ada
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Assertions; use Ada.Assertions;

procedure Show_Assert_Procedure_Policy is
  pragma Assertion_Policy (Assert => Ignore);
  I : constant Integer := 1;
begin
  Put_Line ("------ Pragma Assert -----");
  pragma Assert (I = 0);
  Put_Line ("----- Procedure Assert -----");
  Assert (I = 0);
  Put_Line ("Finished.");
end Show_Assert_Procedure_Policy;
```

149 https://gcc.gnu.org/onlinedocs/gnat_ugn/The-Configuration-Pragmas-Files.html#The-Configuration-Pragmas-Files
Here, the **pragma Assert** is ignored due to the assertion policy. However, the call to `Assert` is not ignored.

---

### 26.4.3 Checks and exceptions

This table shows all language-defined checks and the associated exceptions:

<table>
<thead>
<tr>
<th>Check</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Discriminant_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Division_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Index_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Length_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Overflow_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Range_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Tag_Check</td>
<td>Constraint_Error</td>
</tr>
<tr>
<td>Accessibility_Check</td>
<td>Program_Error</td>
</tr>
<tr>
<td>Allocation_Check</td>
<td>Program_Error</td>
</tr>
<tr>
<td>Elaboration_Check</td>
<td>Program_Error</td>
</tr>
<tr>
<td>Storage_Check</td>
<td>Storage_Error</td>
</tr>
</tbody>
</table>

In addition, we can use `All_Checks` to refer to all those checks above at once.

Let's discuss each check and see code examples where those checks are performed. Note that all examples are erroneous, so please avoid reusing them elsewhere.

---

| 150 | http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html |
**Access Check**

As you know, an object of an access type might be null. It would be an error to dereference this object, as it doesn't indicate a valid position in memory. Therefore, the access check verifies that an access object is not null when dereferencing it. For example:

```ada
procedure Show_Access_Check is
  type Integer_Access is access Integer;
  AI : Integer_Access;
begin
  AI.all := 10;
end Show_Access_Check;
```

Here, the value of `AI` is null by default, so we cannot dereference it.

The access check also performs this verification when assigning to a subtype that excludes null (**not null access**). (You can find more information about this topic in the section about **not null access** (page 808).) For example:

```ada
procedure Show_Access_Check is
  type Integer_Access is access all Integer;
  type Safe_Integer_Access is not null access all Integer;
  AI : Integer_Access;
  SAI : Safe_Integer_Access := new Integer;
begin
  SAI := Safe_Integer_Access (AI);
end Show_Access_Check;
```
Build output

show_access_check.adb:9:04: warning: variable "AI" is read but never assigned [-\_gnatwv]
show_access_check.adb:13:32: warning: null value not allowed here [enabled by default]
show_access_check.adb:13:32: warning: Constraint_Error will be raised at run time [enabled by default]

Runtime output

raised CONSTRAINT_ERROR : show_access_check.adb:13 access check failed

Here, the value of AI is null (by default), so we cannot assign it to SAI because its type excludes null.

Note that, if we remove the := new Integer assignment from the declaration of SAI, the null exclusion fails in the declaration itself (because the default value of the access type is null).

**Discriminant Check**

As we’ve seen earlier, a variant record is a record with discriminants that allows for changing its structure. In operations such as an assignment, it’s important to ensure that the discriminants of the objects match — i.e. to ensure that the structure of the objects matches. The discriminant check verifies whether this is the case. For example:

```
Listing 138: show_discriminant_check.adb

procedure Show_Discriminant_Check is
  type Rec (Valid : Boolean) is record
    case Valid is
      when True =>
        Counter : Integer;
      when False =>
        null;
    end case;
  end record;
  R : Rec (Valid => False);
begin
  R := (Valid => True,
       Counter => 10);
end Show_Discriminant_Check;
```

Build output
Here, R’s discriminant (Valid) is **False**, so we cannot assign an object whose Valid discriminant is **True**.

Also, when accessing a component, the discriminant check ensures that this component exists for the current discriminant value:

```
procedure Show_Discriminant_Check is
  type Rec (Valid : Boolean) is record
  case Valid is
    when True =>
      Counter : Integer;
    when False =>
      null;
  end case;
end record;

R : Rec (Valid => False);
I : Integer;
begin
  I := R.Counter;
end Show_Discriminant_Check;
```

Here, R’s discriminant (Valid) is **False**, so we cannot access the Counter component, for it only exists when the Valid discriminant is **True**.
Division Check

The division check verifies that we're not trying to divide a value by zero when using the `/`, `rem` and `mod` operators. For example:

Listing 140: ops.ads

```ada
package Ops is
  function Div_Op (A, B : Integer) return Integer is
    (A / B);
  function Rem_Op (A, B : Integer) return Integer is
    (A rem B);
  function Mod_Op (A, B : Integer) return Integer is
    (A mod B);
end Ops;
```

Listing 141: show_division_check.adb

```ada
with Ops; use Ops;

procedure Show_Division_Check is
  I : Integer;
begin
  I := Div_Op (10, 0);
  I := Rem_Op (10, 0);
  I := Mod_Op (10, 0);
end Show_Division_Check;
```

Runtime output

raised CONSTRAINT_ERROR : ops.ads:4 divide by zero

All three calls in the Show_Division_Check procedure — to the Div_Op, Rem_Op and Mod_Op functions — can raise an exception because we're using 0 as the second argument, which makes the division check in those functions fail.

Index Check

We use indices to access components of an array. An index check verifies that the index we're using to access a specific component is within the array's bounds. For example:

Listing 142: show_index_check.adb

```ada
procedure Show_Index_Check is
  type Integer_Array is
    array (Positive range <>) of Integer;
```

(continues on next page)
function Value_Of (A : Integer_Array; I : Integer)
    return Integer is
    type Half_Integer_Array is new Integer_Array (A'First ..
        A'First + A'Length / 2);
    A_2 : Half_Integer_Array := (others => 0);
    begin
        return A_2 (I);
        end Value_Of;
    Arr_1 : Integer_Array (1 .. 10) :=
        (others => 1);
    begin
        Arr_1 (10) := Value_Of (Arr_1, 10);
        end Show_Index_Check;

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Index_Check
MD5: fa791718701c4ac805badf368df9064e

Runtime output

raised CONSTRAINT_ERROR : show_index_check.adb:16 index check failed

The range of A_2 — which is passed as an argument to the Value_Of function — is 1 to 6. However, in that function call, we’re trying to access position 10, which is outside A_2’s bounds.

Length Check

In array assignments, both arrays must have the same length. To ensure that this is the case, a length check is performed. For example:

Listing 143: show_length_check.adb

procedure Show_Length_Check is
    type Integer_Array is
        array (Positive range <>) of Integer;
    procedure Assign (To : out Integer_Array;
        From : Integer_Array) is
    begin
        To := From;
        end Assign;
    Arr_1 : Integer_Array (1 .. 10);
    Arr_2 : Integer_Array (1 .. 9) :=
        (others => 1);
    begin
        (continues on next page)
Assign (Arr_1, Arr_2);
end Show_Length_Check;

Code block metadata
MD5: a521afd0a46a67d260e8b0bd5f046ce4

Runtime output
raised CONSTRAINT_ERROR : show_length_check.adb:9 length check failed

Here, the length of Arr_1 is 10, while the length of Arr_2 is 9, so we cannot assign Arr_2 (From parameter) to Arr_1 (To parameter) in the Assign procedure.

Overflow Check

Operations on scalar objects might lead to overflow, which, if not checked, lead to wrong information being computed and stored. Therefore, an overflow check verifies that the value of a scalar object is within the base range of its type. For example:

Listing 144: show_overflow_check.adb

1 procedure Show_Overflow_Check is
2   A, B : Integer;
3 begin
4   A := Integer'Last;
5   B := 1;
6   A := A + B;
7 end Show_Overflow_Check;

Code block metadata
MD5: baa46d9085cbdb14863aaa7e24dc7b9cc

Build output
show_overflow_check.adb:7:11: warning: value not in range of type "Standard.Integer" [enabled by default]
show_overflow_check.adb:7:11: warning: Constraint_Error will be raised at run time [enabled by default]

Runtime output
raised CONSTRAINT_ERROR : show_overflow_check.adb:7 overflow check failed

In this example, A already has the last possible value of the Integer'Base range, so increasing it by one causes an overflow error.
Range Check

The range check verifies that a scalar value is within a specific range — for instance, the range of a subtype. Let's see an example:

Listing 145: show_range_check.adb

```ada
procedure Show_Range_Check is
  subtype Int_1_10 is Integer range 1 .. 10;
  I : Int_1_10;
begin
  I := 11;
end Show_Range_Check;
```

Build output

```
show_range_check.adb:8:09: warning: value not in range of type "Int_1_10" defined,
  at line 3 [enabled by default]
show_range_check.adb:8:09: warning: Constraint_Error will be raised at run time,
  [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_range_check.adb:8 range check failed
```

In this example, we're trying to assign 11 to the variable I of the Int_1_10 subtype, which has a range from 1 to 10. Since 11 is outside that range, the range check fails.

Tag Check

The tag check ensures that the tag of a tagged object matches the expected tag in a dispatching operation. For example:

Listing 146: p.ads

```ada
package P is
  type T is tagged null record;
  type T1 is new T with null record;
  type T2 is new T with null record;
end P;
```

Listing 147: show_tag_check.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Tags;
with P; use P;
```
procedure Show_Tag_Check is

A1 : T'Class := T1'(null record);
A2 : T'Class := T2'(null record);

begin
  Put_Line ("A1'Tag: ">
         & Ada.Tags.Expanded_Name (A1'Tag));
  Put_Line ("A2'Tag: ">
         & Ada.Tags.Expanded_Name (A2'Tag));
  A2 := A1;
end Show_Tag_Check;

Code block metadata

MD5: 5a685be7804200a884649f54c175ee42

Runtime output

A1'Tag: P.T1
A2'Tag: P.T2
raised CONSTRAINT_ERROR : show_tag_check.adb:17 tag check failed

Here, A1 and A2 have different tags:

- A1'Tag = T1'Tag, while
- A2'Tag = T2'Tag.

Since the tags don't match, the tag check fails in the assignment of A1 to A2.

Accessibility Check

The accessibility check verifies that the accessibility level of an entity matches the expected level. We discuss accessibility levels in a later chapter (page 788).

Let's look at an example that mixes access types and anonymous access types. Here, we use an anonymous access type in the declaration of A1 and a named access type in the declaration of A2:

Listing 148: p.ads

package P is
  type T is tagged null record;
  type T_Class is access all T'Class;
end P;

Listing 149: show_accessibility_check.adb

with P; use P;
procedure Show_Accessibility_Check is

(continues on next page)
Learning Ada

(continued from previous page)

A1 : access T'Class := new T;
A2 : T_Class;
begin
  A2 := T_Class (A1);
end Show_Accessibility_Check;

Code block metadata

  Accessibility_Check
MD5: 7120d908b55ef576db93e9a15db257f2

Build output

show_accessibility_check.adb:9:19: warning: accessibility check fails [enabled by default]
show_accessibility_check.adb:9:19: warning: Program_Error will be raised at runtime [enabled by default]

Runtime output

raised PROGRAM_ERROR : show_accessibility_check.adb:9 accessibility check failed

The anonymous type (access T'Class), which is used in the declaration of A1, doesn't have the same accessibility level as the T_Class type. Therefore, the accessibility check fails during the T_Class (A1) conversion.

We can see the accessibility check failing in this example as well:

Listing 150: show_accessibility_check.adb

with P; use P;
procedure Show_Accessibility_Check is
  A : access T'Class := new T;
  procedure P (A : T_Class) is null;
begin
  P (T_Class (A));
end Show_Accessibility_Check;

Code block metadata

  Accessibility_Check
MD5: 97db824a0dd345249d0e7a97118b7ef

Build output

show_accessibility_check.adb:10:16: warning: accessibility check fails [enabled by default]
show_accessibility_check.adb:10:16: warning: Program_Error will be raised at runtime [enabled by default]

Runtime output
Learning Ada

raised PROGRAM_ERROR : show_accessibility_check.adb:10 accessibility check failed

Again, the check fails in the T_Class (A) conversion and raises a Program_Error exception.

**Allocation Check**

The allocation check ensures, when a task is about to be created, that its master has not been completed or the finalization has not been started.

This is an example adapted from AI-00280:\textsuperscript{151}:

```ada
with Ada.Finalization;
with Ada.Unchecked_Deallocation;

package P is
  type T1 is new
    Ada.Finalization.Controlled with null record;
  procedure Finalize (X : in out T1);

  type T2 is new
    Ada.Finalization.Controlled with null record;
  procedure Finalize (X : in out T2);

  X1 : T1;

  type T2_Ref is access T2;
  procedure Free is new
    Ada.Unchecked_Deallocation (T2, T2_Ref);
end P;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body P is

  procedure Finalize (X : in out T1) is
    X2 : T2_Ref := new T2;
  begin
    Put_Line ("Finalizing T1...");
    Free (X2);
  end Finalize;

  procedure Finalize (X : in out T2) is
  begin
    Put_Line ("Finalizing T2...");
  end Finalize;
end P;
```

```ada
with P; use P;

procedure Show_Allocation_Check is

(continues on next page)
```

\textsuperscript{151} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00280.txt?rev=1.12&raw=N

26.4. Exceptions
Learning Ada

(continued from previous page)

4 X2 : T2_Ref := new T2;
5 begin
6 Free (X2);
7 end Show_Allocation_Check;

Code block metadata

MD5: 915e8ab21e550c981503c014bcceadel

Runtime output

Finalizing T2...
raised PROGRAM_ERROR : finalize/adjust raised exception

Here, in the finalization of the X1 object of T1 type, we're trying to create an object of T2 type. This is forbidden and, therefore, the allocation check raises a Program_Error exception.

Elaboration Check

The elaboration check verifies that subprograms — or protected entries, or task activations — have been elaborated before being called.

This is an example adapted from Al-00064:

Listing 154: p.ads

function P return Integer;

Listing 155: p.adb

function P return Integer is
begin
  return 1;
end P;

Listing 156: show_elaboration_check.adb

with P;

procedure Show_Elaboration_Check is
  function F return Integer;
  type Pointer_To_Func is
    access function return Integer;
  X : constant Pointer_To_Func := P'Access;
  Y : constant Integer := F;
  Z : constant Pointer_To_Func := X;
  -- Renaming-as-body
  function F return Integer renames Z.all;

152 http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/al-00064.txt?rev=1.12&raw=N

672 Chapter 26. Control Flow
begin
  null;
end Show_Elaboration_Check;

Code block metadata
Elaboration Check
MD5: 80a39df912aae8788296f81ee9d4a79e

Build output
show_elaboration_check.adb:12:28: warning: cannot call "F" before body seen,
   [enabled by default]
show_elaboration_check.adb:12:28: warning: Program_Error will be raised at run,
   time [enabled by default]

Runtime output
raised PROGRAM_ERROR : show_elaboration_check.adb:12 access before elaboration

This is a curious example: first, we declare a function F and assign the value returned by this
function to constant Y in its declaration. Then, we declare F as a renamed function, thereby
providing a body to F — this is called renaming-as-body. Consequently, the compiler doesn't
complain that a body is missing for function F. (If you comment out the function renaming,
you'll see that the compiler can then detect the missing body.) Therefore, at runtime, the
elaboration check fails because the body of the first declaration of the F function is actually
missing.

Storage Check

The storage check ensures that the storage pool has enough space when allocating memory.
Let's revisit an example that we discussed earlier (page 355):

Listing 157: custom_types.ads
package Custom_Types is
  type UInt_7 is range 0 .. 127;
  type UInt_7_Resumed_Access is access UInt_7
   with Storage_Size => 8;
end Custom_Types;

Listing 158: show_storage_check.adb
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Types; use Custom_Types;
procedure Show_Storage_Check is
  RAV1, RAV2 : UInt_7_Resumed_Access;
begin
  Put_Line ("Allocating RAV1...");
  (continues on next page)
RAV1 := \texttt{new} UInt_7;

Put_Line ("Allocating RAV2...");
RAV2 := \texttt{new} UInt_7;

New_Line;
end Show_Storage_Check;

Code block metadata

-Storage_Check
MD5: 4e4bd284adb1c1d97f8f7563068c18de

Runtime output

Allocating RAV1...
Allocating RAV2...
raised STORAGE_ERROR : s-poosiz.adb:108 explicit raise

On each allocation (\texttt{new} UInt_7), a storage check is performed. Because there isn't enough reserved storage space before the second allocation, the checks fail and raises a Storage_Error exception.

In the Ada Reference Manual

- 11.5 Suppressing Checks\textsuperscript{153}

26.4.4 Ada.Exceptions package

\textbf{Note:} Parts of this section were originally published as Gem \#142: Exception-ally\textsuperscript{154}

The standard Ada run-time library provides the package Ada.Exceptions. This package provides a number of services to help analyze exceptions.
Each exception is associated with a (short) message that can be set by the code that raises the exception, as in the following code:

\texttt{raise Constraint_Error with "some message";}

Historically

Since Ada 2005, we can use the \texttt{raise Constraint_Error with "some message"} syntax. In Ada 95, you had to call the Raise_Exception procedure:

\texttt{Ada.Exceptions.Raise_Exception (Constraint_Error'Identity, "some message");}

In Ada 83, there was no way to do it at all.
The new syntax is now very convenient, and developers should be encouraged to provide as much information as possible along with the exception.

\textsuperscript{153} \url{http://www.ada-auth.org/standards/22rm/html/RM-11-5.html}
\textsuperscript{154} \url{https://www.adacore.com/gems/gem-142-exceptions}
In the GNAT toolchain

The length of the message is limited to 200 characters by default in GNAT, and messages longer than that will be truncated.

In the Ada Reference Manual

- 11.4.1 The Package Exceptions

Retrieving exception information

Exceptions also embed information set by the run-time itself that can be retrieved by calling the `Exception_Information` function. The function `Exception_Information` also displays the `Exception_Message`.

For example:

```ada
exception
  when E : others =>
    Put_Line
    (Ada.Exceptions.Exception_Information (E));
```

In the GNAT toolchain

In the case of GNAT, the information provided by an exception might include the source location where the exception was raised and a nonsymbolic traceback.

You can also retrieve this information individually. Here, you can use:

- the `Exception_Name` functions — and its derivatives `Wide_Exception_Name` and `Wide_Wide_Exception_Name` — to retrieve the name of an exception.
- the `Exception_Message` function to retrieve the message associated with an exception.

Let's see a complete example:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Show_Exception_Info is
  Custom_Exception : exception;

  procedure Nested is
  begin
    raise Custom_Exception
      with Custom_Exception
      with "We got a problem";
  end Nested;

begin
  Nested;
```

Listing 159: show_exception_info.adb

exception
when E : others =>
  Put_Line ("Exception info: 
    & Exception_Information (E));
  Put_Line ("Exception name: 
    & Exception_Name (E));
  Put_Line ("Exception msg: 
    & Exception_Message (E));
end Show_Exception_Info;

Collecting exceptions

Save_Occurrence

You can save an exception occurrence using the Save_Occurrence procedure. (Note that a
Save_Occurrence function exists as well.)

For example, the following application collects exceptions into a list and displays them after
running the Test_Exceptions procedure:

Listing 160: exception_tests.ads

with Ada.Exceptions; use Ada.Exceptions;

package Exception_Tests is

  Custom_Exception : exception;

  type All_Exception_Occur is
    array (Positive range <>) of
      Exception_Occurrence;

  procedure Test_Exceptions
    (All_Occur : in out All_Exception_Occur;
     Last_Occur : out Integer);

end Exception_Tests;

Listing 161: exception_tests.adb

package body Exception_Tests is

  procedure Save_To_List
    (E : Exception_Occurrence;
     All_Occur : in out All_Exception_Occur;
     Last_Occur : in out Integer)
  is
    L : Integer renames Last_Occur;
    O : All_Exception_Occur renames All_Occur;
    begin
      L := L + 1;
      if L > O'Last then
        raise Constraint_Error
        with "Cannot save occurrence";
      end if;
      Save_Occurrence (Target => O (L),
                      Source => E);

(continues on next page)
end Save_To_List;

procedure Test_Exceptions
(All_Occur : in out All_Exception_Occur;
Last_Occur : out Integer)
is

procedure Nested_1 is
begin
raise Custom_Exception
with "We got a problem";
exception
when E : others =>
Save_To_List (E,
All_Occur,
Last_Occur);
end Nested_1;

procedure Nested_2 is
begin
raise Constraint_Error
with "Constraint is not correct";
exception
when E : others =>
Save_To_List (E,
All_Occur,
Last_Occur);
end Nested_2;

begin
Last_Occur := 0;
Nested_1;
Nested_2;
end Test_Exceptions;
end Exception_Tests;

Listing 162: show_exception_info.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Exception_Tests; use Exception_Tests;

procedure Show_Exception_Info is
L : Integer;
O : All_Exception_Occur (1 .. 10);
begin
Test_Exceptions (O, L);
for I in O 'First .. L loop
Put_Line (Exception_Information (O (I)));
end loop;
end Show_Exception_Info;

Code block metadata
MD5: da0cc5db7039e1458dbcf8be49db969d

26.4. Exceptions
Learning Ada

Runtime output

raised EXCEPTION_TESTS.CUSTOM_EXCEPTION : We got a problem
raised CONSTRAINT_ERROR : Constraint is not correct

In the Save_To_List procedure of the Exception_Tests package, we call the Save_Occurrence procedure to store the exception occurrence to the All_Occur array. In the Show_Exception_Info, we display all the exception occurrences that we collected.

Read and Write attributes

Similarly, we can use files to read and write exception occurrences. To do that, we can simply use the Read and Write attributes.

Listing 163: exception_occurrence_stream.adb

```ada
with Ada.Text_IO;
with Ada.Streams.Stream_IO;
use Ada.Streams.Stream_IO;
with Ada.Exceptions;
use Ada.Exceptions;

procedure Exception_Occurrence_Stream is
  Custom_Exception : exception;
  S : Stream_Access;

  procedure Nested_1 is
  begin
    raise Custom_Exception
    with "We got a problem";
  exception
    when E : others =>
      Exception_Occurrence'Write (S, E);
  end Nested_1;

  procedure Nested_2 is
  begin
    raise Constraint_Error
    with "Constraint is not correct";
  exception
    when E : others =>
      Exception_Occurrence'Write (S, E);
  end Nested_2;

  F : File_Type;
  File_Name : constant String :=
    "exceptions_file.bin";

begin
  Create (F, Out_File, File_Name);
  S := Stream (F);
  Nested_1;
  Nested_2;
end
```

(continues on next page)
Close (F);

**Read Exceptions** : declare
  E : Exception_Occurrence;
begin
  Open (F, In_File, File_Name);
  S := Stream (F);
  while not End_Of_File (F) loop
    Exception_Occurrence'Read (S, E);
    Ada.Text_IO.Put_Line
      (Exception_Information (E));
  end loop;
  Close (F);
end Read_Exceptions;
end Exception_Occurrence_Stream;

**Code block metadata**

MD5: 3d9f2bd9480aa6dcc250b249b9ef4870

**Runtime output**

raised EXCEPTION_OCCURRENCE_STREAM.CUSTOM_EXCEPTION : We got a problem
raised CONSTRAINT_ERROR : Constraint is not correct

In this example, we store the exceptions raised in the application in the exceptions_file.bin file. In the exception part of procedures Nested_1 and Nested_2, we call Exception_Occurrence'Write to store an exception occurrence in the file. In the Read_Exceptions block, we read the exceptions from the file by calling Exception_Occurrence'Read.

**Debugging exceptions in the GNAT toolchain**

Here is a typical exception handler that catches all unexpected exceptions in the application:

```
with Ada.Exceptions;
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
  procedure Nested is
  begin
    raise Constraint_Error
      with "some message";
  end Nested;
begin
  Nested;
```

(continues on next page)
exception
  when E =>
  Put_Line
  (Ada.Exceptions.Exception.Information (E));
end Main;

Code block metadata

MD5: f95068ca90d79b92a7c2031322349153

Runtime output

raised CONSTRAINT_ERROR : some message

The output we get when running the application is not very informative. To get more information, we need to rerun the program in the debugger. To make the session more interesting though, we should add debug information in the executable, which means using the -g switch in the gnatmake command.

The session would look like the following (omitting some of the output from the debugger):

> rm *.o  # Cleanup previous compilation
> gnatmake -g main.adb
> gdb ./main
(gdb) catch exception
(gdb) run

Catchpoint 1, CONSTRAINT_ERROR at 0x0000000000402860 in main.nested () at main. 
     .adb:8
     8 raise Constraint_Error with "some message";

(gdb) bt
#0 <__gnat_debug_raise_exception> (e=0x62ec40 <constraint_error>) at s-excdeb. 
     .adb:43
#1 0x000000000040426f in ada.exceptions.complete_occurrence (x=x@entry=0x637050) at a-except.adb:934
#2 0x000000000040427b in ada.exceptions.complete_and_propagate_occurrence ( 
     x=x@entry=0x637050) at a-except.adb:943
#3 0x00000000004042d0 in __gnat_raise_exception (e=0x62ec40 <constraint_error>, 
     message=...) at a-except.adb:982
#4 0x0000000000402860 in main.nested ()
#5 0x000000000040287c in main ()

And we now know exactly where the exception was raised. But in fact, we could have this information directly when running the application. For this, we need to bind the application with the switch -E, which tells the binder to store exception tracebacks in exception occurrences. Let's recompile and rerun the application.

> rm *.o   # Cleanup previous compilation
> gnatmake -g main.adb -bargs -E
> ./main

Exception name: CONSTRAINT_ERROR
Message: some message
Call stack traceback locations:
0x10b7e24d1 0x10b7e24ee 0x10b7e2472

The traceback, as is, is not very useful. We now need to use another tool that is bundled with GNAT, called addr2line. Here is an example of its use:

680 Chapter 26. Control Flow
This time we do have a symbolic backtrace, which shows information similar to what we got in the debugger.

For users on OSX machines, `addr2line` does not exist. On these machines, however, an equivalent solution exists. You need to link your application with an additional switch, and then use the tool `atos`, as in:

```
> rm *.o
> gnatmake -g main.adb -bargs -E -largs -Wl,-no_pie
> ./main
```

We will now discuss a relatively new switch of the compiler, namely `-gnateE`. When used, this switch will generate extra information in exception messages.

Let’s amend our test program to:

```
with Ada.Exceptions;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    procedure Nested (Index : Integer) is
        type T_Array is array (1 .. 2) of Integer;
        T : constant T_Array := (10, 20);
        begin
            Put_Line (T (Index)'Img);
        end Nested;
    begin
        Nested (3);
    exception
        when E : others =>
            Put_Line
            (Ada.Exceptions.Exception_Information (E));
    end Main;
```

**Code block metadata**

**Project:** Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Exception_Information

**MD5:** 3590f2bf48f6ed1cf7745d576924cad4

**Runtime output**

---

26.4. Exceptions 681
Learning Ada

raised CONSTRAINT_ERROR : main.adb:10:17 index check failed
index 3 not in 1..2

When running the application, we see that the exception information (traceback) is the same as before, but this time the exception message is set automatically by the compiler. So we know we got a Constraint_Error because an incorrect index was used at the named source location (main.adb, line 10). But the significant addition is the second line of the message, which indicates exactly the cause of the error. Here, we wanted to get the element at index 3, in an array whose range of valid indexes is from 1 to 2. (No need for a debugger in this case.)

The column information on the first line of the exception message is also very useful when dealing with null pointers. For instance, a line such as:

```ada
```

where each of the Rec is itself a pointer, might raise Constraint_Error with a message "access check failed". This indicates for sure that one of the pointers is null, and by using the column information it is generally easy to find out which one it is.

### 26.4.5 Exception renaming

We can rename exceptions by using the an exception renaming declaration in this form:

```ada
Renamed_Exception : exception renames Existing_Exception;
```

For example:

```ada
Listing 166: show_exception_renaming.adb

procedure Show_Exception_Renaming is
begin
  raise CE;
end Show_Exception_Renaming;
```

**Code block metadata**


MD5: ff20825162ee9ef6ac8ed329da2a80f

**Runtime output**

raised CONSTRAINT_ERROR : show_exception_renaming.adb:4

Exception renaming creates a new view of the original exception. If we rename an exception from package A in package B, that exception will become visible in package B. For example:

```ada
Listing 167: internal_exceptions.ads

package Internal_Exceptions is

Int_E : exception;
end Internal_Exceptions;
```
Listing 168: test_constraints.ads

```ada
with Internal_Exceptions;

package Test_Constraints is
  Ext_E : exception renames
    Internal_Exceptions.Int_E;
end Test_Constraints;
```

Listing 169: show_exception_renaming_view.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Test_Constraints; use Test_Constraints;

procedure Show_Exception_Renaming_View is
begin
  raise Ext_E;
  exception
    when E : others =>
      Put_Line
        (Ada.Exceptions.Exception_Information (E));
end Show_Exception_Renaming_View;
```

Code block metadata

MD5: a44e2698170c6fab79241d0f33ef8c2e

Runtime output

raised INTERNAL_EXCEPTIONS.INT_E : show_exception_renaming_view.adb:8

Here, we're renaming the Int_E exception in the Test_Constraints package. The Int_E exception isn't directly visible in the Show_Exception_Renaming procedure because we're not `with`ing the Internal_Exceptions package. However, it is indirectly visible in that procedure via the renaming (Ext_E) in the Test_Constraints package.

In the Ada Reference Manual

- 8.5.2 Exception Renaming Declarations\(^\text{156}\)

\(^{156}\) http://www.ada-auth.org/standards/22rm/html/RM-8-5-2.html
26.4.6 Out and Uninitialized

Note: This section was originally written by Robert Dewar and published as Gem #150: Out and Uninitialized\(^{157}\)

Perhaps surprisingly, the Ada standard indicates cases where objects passed to \texttt{out} and \texttt{in out} parameters might not be updated when a procedure terminates due to an exception. Let's take an example:

Listing 170: show_out_uninitialized.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Out_Uninitialized is

  procedure Local (A : in out Integer;
                   Error : Boolean) is
  begin
    A := 1;
    if Error then
      raise Program_Error;
    end if;
  end Local;

  B : Integer := 0;

begin
  Local (B, Error => True);
exception
  when Program_Error =>
    Put_Line ('Value for B is' & Integer'Image (B)); -- "0"
end Show_Out_Uninitialized;
```

This program outputs a value of 0 for B, whereas the code indicates that A is assigned before raising the exception, and so the reader might expect B to also be updated.

The catch, though, is that a compiler must by default pass objects of elementary types (scalars and access types) by copy and might choose to do so for other types (records, for example), including when passing \texttt{out} and \texttt{in out} parameters. So what happens is that while the formal parameter A is properly initialized, the exception is raised before the new value of A has been copied back into B (the copy will only happen on a normal return).

In the GNAT toolchain

In general, any code that reads the actual object passed to an \texttt{out} or \texttt{in out} parameter after an exception is suspect and should be avoided. GNAT has useful warnings here, so that if we simplify the above code to:

\(^{157}\) https://www.adacore.com/gems/gem-150out-and-uninitialized
Listing 171: show_out_uninitialized_warnings.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Out_Uninitialized_Warnings is
  procedure Local (A : in out Integer) is
    begin
      A := 1;
      raise Program_Error;
    end Local;

    B : Integer := 0;
  begin
    Local (B);
  exception
    when others =>
      Put_Line ("Value for B is" & Integer'Image (B));
  end Show_Out_Uninitialized_Warnings;
```

Code block metadata

MD5: 5b6960974c729ea37a70fb313d6e5084

Build output

show_out_uninitialized_warnings.adb:7:10: warning: assignment to pass-by-copy formal may have no effect [enabled by default]
show_out_uninitialized_warnings.adb:7:10: warning: "raise" statement may result in abnormal return (RM 6.4.1(17)) [enabled by default]

Runtime output

Value for B is 0

We now get a compilation warning that the pass-by-copy formal may have no effect. Of course, GNAT is not able to point out all such errors (see first example above), which in general would require full flow analysis.

The behavior is different when using parameter types that the standard mandates be passed by reference, such as tagged types for instance. So the following code will work as expected, updating the actual parameter despite the exception:

Listing 172: show_out_initialized_rec.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Out_Initiaed_Rec is
  type Rec is tagged record
    Field : Integer;
  end record;

  procedure Local (A : in out Rec) is
    begin
      A.Field := 1;
  end Local;
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
raise Program_Error;
end Local;

V : Rec;

begin
V.Field := 0;
Local (V);
exception
when others =>
  Put_Line ("Value of Field is" 
& V.Field'Img); -- "I"
end Show_Out_Initialized_Rec;
```

Code block metadata

MD5: 370031a404657ea18ffabf3c1d507cd4

Runtime output

Value of Field is 1

In the GNAT toolchain

It's worth mentioning that GNAT provides a pragma called Export_Procedure that forces reference semantics on `out` parameters. Use of this pragma would ensure updates of the actual parameter prior to abnormal completion of the procedure. However, this pragma only applies to library-level procedures, so the examples above have to be rewritten to avoid the use of a nested procedure, and really this pragma is intended mainly for use in interfacing with foreign code. The code below shows an example that ensures that `B` is set to 1 after the call to `Local`:

Listing 173: exported_procedures.ads

```ada
package Exported_Procedures is

  procedure Local (A : in out Integer;
                  Error : Boolean);

  pragma Export_Procedure
  (Local,
   Mechanism => (A => Reference));

end Exported_Procedures;
```

Listing 174: exported_procedures.adb

```ada
package body Exported_Procedures is

  procedure Local (A : in out Integer;
                  Error : Boolean) is

  begin A := 1;
    if Error then
      raise Program_Error;
    end if;
  end Local;

end Exported_Procedures;
```
Listing 175: show_out_reference.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Exported_Procedures; use Exported_Procedures;

procedure Show_Out_Reference is
  B : Integer := 0;
begin
  Local (B, Error => True);
exception
  when Program_Error =>
    Put_Line ("Value for B is" & Integer'Image (B)); -- "1"
end Show_Out_Reference;
```

In the case of direct assignments to global variables, the behavior in the presence of exceptions is somewhat different. For predefined exceptions, most notably Constraint_Error, the optimization permissions allow some flexibility in whether a global variable is or is not updated when an exception occurs (see Ada RM 11.6\textsuperscript{158}). For instance, the following code makes an incorrect assumption:

\begin{verbatim}
X := 0; -- about to try addition
Y := Y + 1; -- see if addition raises exception
X := 1 -- addition succeeded
\end{verbatim}

A program is not justified in assuming that $X = 0$ if the addition raises an exception (assuming $X$ is a global here). So any such assumptions in a program are incorrect code which should be fixed.

In the Ada Reference Manual

- 11.6 Exceptions and Optimization\textsuperscript{159}

\textsuperscript{158} http://www.ada-auth.org/standards/22rm/html/RM-11-6.html
\textsuperscript{159} http://www.ada-auth.org/standards/22rm/html/RM-11-6.html
26.4.7 Suppressing checks

Pragma Suppress

Note: This section was originally written by Gary Dismukes and published as Gem #63: The Effect ofPragma Suppress.

One of Ada's key strengths has always been its strong typing. The language imposes stringent checking of type and subtype properties to help prevent accidental violations of the type system that are a common source of program bugs in other less-strict languages such as C. This is done using a combination of compile-time restrictions (legality rules), that prohibit mixing values of different types, together with run-time checks to catch violations of various dynamic properties. Examples are checking values against subtype constraints and preventing dereferences of null access values.

At the same time, Ada does provide certain "loophole" features, such as Unchecked_Conversion, that allow selective bypassing of the normal safety features, which is sometimes necessary when interfacing with hardware or code written in other languages.

Ada also permits explicit suppression of the run-time checks that are there to ensure that various properties of objects are not violated. This suppression can be done using pragma Suppress, as well as by using a compile-time switch on most implementations — in the case of GNAT, with the -gnatp switch.

In addition to allowing all checks to be suppressed, pragma Suppress supports suppression of specific forms of check, such as Index_Check for array indexing, Range_Check for scalar bounds checking, and Access_Check for dereferencing of access values. (See section 11.5 of the Ada Reference Manual for further details.)

Here's a simple example of suppressing index checks within a specific subprogram:

```
procedure Main is
  procedure Sort_Array (A : in out Some_Array) is
    pragma Suppress (Index_Check);
    -- ^^^^^^^^^^^^^^^^^^^^^
    -- eliminate check overhead
  begin
    ... end Sort_Array;
end Main;
```

Unlike a feature such as Unchecked_Conversion, however, the purpose of check suppression is not to enable programs to subvert the type system, though many programmers seem to have that misconception.

What's important to understand about pragma Suppress is that it only gives permission to the implementation to remove checks, but doesn't require such elimination. The intention of Suppress is not to allow bypassing of Ada semantics, but rather to improve efficiency, and the Ada Reference Manual has a clear statement to that effect in the note in RM-11.5, paragraph 29:

> There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress should be used only for efficiency reasons.

There is associated Implementation Advice that recommends that implementations should minimize the code executed for checks that have been suppressed, but it's still the responsibility of the programmer to ensure that the correct functioning of the program doesn't depend on checks not being performed.

---

160 https://www.adacore.com/gems/gem-63
There are various reasons why a compiler might choose not to remove a check. On some hardware, certain checks may be essentially free, such as null pointer checks or arithmetic overflow, and it might be impractical or add extra cost to suppress the check. Another example where it wouldn't make sense to remove checks is for an operation implemented by a call to a run-time routine, where the check might be only a small part of a more expensive operation done out of line.

Furthermore, in many cases GNAT can determine at compile time that a given run-time check is guaranteed to be violated. In such situations, it gives a warning that an exception will be raised, and generates code specifically to raise the exception. Here's an example:

```ada
X : Integer range 1..10 := ...;
...
if A > B then
  X := X + 1;
...
end if;
```

For the assignment incrementing X, the compiler will normally generate machine code equivalent to:

```ada
Temp := X + 1;
if Temp > 10 then
  raise Constraint_Error;
end if;
X := Temp;
```

If range checks are suppressed, then the compiler can just generate the increment and assignment. However, if the compiler is able to somehow prove that \( X = 10 \) at this point, it will issue a warning, and replace the entire assignment with simply:

```ada
raise Constraint_Error;
```

even though checks are suppressed. This is appropriate, because

1. we don't care about the efficiency of buggy code, and
2. there is no "extra" cost to the check, because if we reach that point, the code will unconditionally fail.

One other important thing to note about checks and **pragma Suppress** is this statement in the Ada RM (RM-11.5, paragraph 26):

> If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous.

In Ada, erroneous execution is a bad situation to be in, because it means that the execution of your program could have arbitrary nasty effects, such as unintended overwriting of memory. Note also that a program whose "correct" execution somehow depends on a given check being suppressed might work as the programmer expects, but could still fail when compiled with a different compiler, or for a different target, or even with a newer version of the same compiler. Other changes such as switching on optimization or making a change to a totally unrelated part of the code could also cause the code to start failing.

So it's definitely not wise to write code that relies on checks being removed. In fact, it really only makes sense to suppress checks once there's good reason to believe that the checks can't fail, as a result of testing or other analysis. Otherwise, you're removing an important safety feature of Ada that's intended to help catch bugs.
pragma Unsuppress

We can use `pragma Unsuppress` to reverse the effect of a `pragma Suppress`. While `pragma Suppress` gives permission to the compiler to remove a specific check, `pragma Unsuppress` revokes that permission.

Let’s see an example:

**Listing 176: show_index_check.adb**

```ada
procedure Show_Index_Check is

  type Integer_Array is
    array (Positive range <>) of Integer;

  pragma Suppress (Index_Check);
  -- from now on, the compiler may
  -- eliminate index checks...

  function Unchecked_Value_Of (A : Integer_Array; I : Integer)
    return Integer is
    type Half_Integer_Array is new Integer_Array (A'First ..
      A'First + A'Length / 2);
    begin
      A_2 : Half_Integer_Array := (others => 0);
      return A_2 (I);
    end Unchecked_Value_Of;

  pragma Unsuppress (Index_Check);
  -- from now on, index checks are
  -- typically performed...

  function Value_Of (A : Integer_Array; I : Integer)
    return Integer is
    type Half_Integer_Array is new Integer_Array (A'First ..
      A'First + A'Length / 2);
    begin
      A_2 : Half_Integer_Array := (others => 0);
      return A_2 (I);
    end Value_Of;

  Arr_1 : Integer_Array (1 .. 10) :=
    (others => 1);

  begin
    Arr_1 (10) := Unchecked_Value_Of (Arr_1, 10);
    Arr_1 (10) := Value_Of (Arr_1, 10);
  end Show_Index_Check;
```

Code block metadata
In this example, we first use a `pragma Suppress` (Index_Check), so the compiler is allowed to remove the index check from the Unchecked_Value_Of function. (Therefore, depending on the compiler, the call to the Unchecked_Value_Of function may complete without raising an exception.) Of course, in this specific example, suppressing the index check masks a severe issue.

In contrast, an index check is performed in the Value_Of function because of the `pragma Unsuppress`. As a result, the index checks fails in the call to this function, which raises a Constraint_Error exception.

-In the Ada Reference Manual
27.1 Packages

27.1.1 Package renaming

We’ve seen in the Introduction to Ada course that we can rename packages (page 46).

In the Ada Reference Manual

• 10.1.1 Compilation Units - Library Units

Grouping packages

A use-case that we haven’t mentioned in that course is that we can apply package renaming to group individual packages into a common hierarchy. For example:

Listing 1: driver_m1.ads

```ada
package Driver_M1 is
end Driver_M1;
```

Listing 2: driver_m2.ads

```ada
package Driver_M2 is
end Driver_M2;
```

Listing 3: drivers.ads

```ada
package Drivers
    with Pure is
end Drivers;
```

Listing 4: drivers-m1.ads

```ada
with Driver_M1;
package Drivers.M1 renames Driver_M1;
```

\[162\] http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html
Listing 5: drivers-m2.ads

```ada
with Driver_M2;
package Drivers.M2 renames Driver_M2;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Modular_ProgPackages.Package_Renaming.Package_Renaming_1
MD5: 8d6a6bec32f7ec4397de1fafa9f0b44d9

Here, we're renaming the Driver_M1 and Driver_M2 packages as child packages of the Drivers package, which is a pure package.

**Important**

Note that a package that is renamed as a child package cannot refer to information from its (non-renamed) parent. In other words, Driver_M1 (renamed as Drivers.M1) cannot refer to information from the Drivers package. For example:

Listing 6: driver_m1.ads

```ada
package Driver_M1 is
  Counter_2 : Integer := Drivers.Counter;
end Driver_M1;
```

Listing 7: drivers.ads

```ada
package Drivers is
  Counter : Integer := 0;
end Drivers;
```

Listing 8: drivers-m1.ads

```ada
with Driver_M1;
package Drivers.M1 renames Driver_M1;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Modular_ProgPackages.Package_Renaming.Package_Renaming_1_Refer_To_Parent
MD5: d174746d8151d9a22cd048ad44e853850

**Build output**

```
driver_m1.ads:3:27: error: "Drivers" is undefined
gprbuild: *** compilation phase failed
```

As expected, compilation fails here because Drivers.Counter isn't visible in Driver_M1, even though the renaming (Drivers.M1) creates a virtual hierarchy.
Child of renamed package

Note that we cannot create a child package using a parent package name that was introduced by a renaming. For example, let’s say we want to create a child package Ext for the Drivers.M1 package we’ve seen earlier. We cannot just declare a Drivers.M1.Ext package like this:

```ada
package Drivers.M1.Ext is
end Drivers.M1.Ext;
```

because the parent unit cannot be a renaming. The solution is to actually extend the original (non-renamed) package:

```ada
package Driver_M1.Ext is
end Driver_M1.Ext;
```

Listing 9: driver_m1-ext.ads

Listing 10: dummy.adb

```ada
-- A package called Drivers.M1.Ext is automatically available!
with Drivers.M1.Ext;

procedure Dummy is
begin
    null;
end Dummy;
```

This works fine because any child package of a package P is also a child package of a renamed version of P. (Therefore, because Ext is a child package of Driver_M1, it is also a child package of the renamed Drivers.M1 package.)

Backwards-compatibility via renaming

We can also use renaming to ensure backwards-compatibility when changing the package hierarchy. For example, we could adapt the previous source-code by:

- converting Driver_M1 and Driver_M2 to child packages of Drivers, and
- using package renaming to mimic the original names (Driver_M1 and Driver_M2).

This is the adapted code:

```ada
package Drivers
    with Pure is
end Drivers;
```

Listing 11: drivers.ads
Now, M1 and M2 are actual child packages of Drivers, but their original names are still available. By doing so, we ensure that existing software that makes use of the original packages doesn't break.

### 27.1.2 Private packages

In this section, we discuss the concept of private packages. However, before we proceed with the discussion, let's recapitulate some important ideas that we've seen earlier.

In the Introduction to Ada course (page 113), we've seen that encapsulation plays an important role in modular programming. By using the private part of a package specification, we can disclose some information, but, at the same time, prevent that this information gets accessed where it shouldn't be used directly. Similarly, we've seen that we can use the private part of a package to distinguish between the partial and full view (page 307) of a data type.

The main application of private packages is to create private child packages, whose purpose
is to serve as internal implementation packages within a package hierarchy. By doing so, we can expose the internals to other public child packages, but prevent that external clients can directly access them.

As we’ll see next, there are many rules that ensure that internal visibility is enforced for those private child packages. At the same time, the same rules ensure that private packages aren't visible outside of the package hierarchy.

### Declaration and usage

We declare private packages by using the `private` keyword. For example, let's say we have a package named `Data_Processing`:

```ada
package Data_Processing is
  -- ...
end Data_Processing;
```

Let's see a complete example:

```ada
package Data_Processing is
  type Data is private;
  procedure Process (D : in out Data);
private
  type Data is null record;
end Data_Processing;
```

---

**27.1. Packages**

---
In this example, we refer to the private child package Calculations in the body of the Data_Processing package — by simply writing `with Data_Processing.Calculations;`. After that, we can call the Calculate procedure normally in the Process procedure.
Private sibling packages

We can introduce another private package Advanced_Calculations as a child of Data_Processing and refer to the Calculations package in its specification:

Listing 23: data_processing.ads

```ada
package Data_Processing is
    type Data is private;
    procedure Process (D : in out Data);
private
    type Data is null record;
end Data_Processing;
```

Listing 24: data_processing-calculations.ads

```ada
private package Data_Processing.Calculations is
    procedure Calculate (D : in out Data);
end Data_Processing.Calculations;
```

Listing 25: data_processing-advanced_calculations.ads

```ada
with Data_Processing.Calculations;
use Data_Processing.Calculations;
private
package Data_Processing.Advanced_Calculations is
    procedure Advanced_Calculate (D : in out Data) renames Calculate;
end Data_Processing.Advanced_Calculations;
```

Listing 26: data_processing.adb

```ada
with Data_Processing.Advanced_Calculations;
use Data_Processing.Advanced_Calculations;
package body Data_Processing is
    procedure Process (D : in out Data) is
        begin
            Advanced_Calculate (D);
        end Process;
end Data_Processing;
```

Listing 27: data_processing-calculations.adb

```ada
package body Data_Processing.Calculations is
    procedure Calculate (D : in out Data) is
        begin
            -- Dummy implementation...
```

(continues on next page)
null;
end Calculate;
end Data Processing.Calculations;

Listing 28: test_data_processing.adb

with Data Processing; use Data Processing;

procedure Test_Data_Processing is
  D : Data;
begin
  Process (D);
end Test_Data_Processing;

Code block metadata

MD5: 32fc76ae13fleeccdd854a02979303d8

Note that, in the body of the Data_Processing package, we're now referring to the new Advanced_Calculations package instead of the Calculations package.

Referring to a private child package in the specification of another private child package is OK, but we cannot do the same in the specification of a non-private package. For example, let's change the specification of the Advanced_Calculations and make it non-private:

Listing 29: data_processing-advanced_calculations.ads

with Data Processing.Calculations;
use Data Processing.Calculations;

package Data Processing.Advanced_Calculations is
  procedure Advanced_Calculate (D : in out Data) renames Calculate;
end Data Processing.Advanced_Calculations;

Code block metadata

MD5: 27fd3b863a11ed7797c4e4fae8349

Build output

data_processing-advanced_calculations.ads:1:06: error: current unit must also be a private descendant of "Data_Processing"
gprbuild: *** compilation phase failed

Now, the compilation doesn't work anymore. However, we could still refer to Calculations packages in the body of the Advanced_Calculations package:

Listing 30: data_processing-advanced_calculations.ads

package Data Processing.Advanced_Calculations is
  procedure Advanced_Calculate (D : in out Data);
end Data Processing.Advanced_Calculations;
Listing 31: data_processing-advanced_calculations.adb

```ada
with Data_Processing.Calculations;
use Data_Processing.Calculations;

package body Data_Processing.Advanced_Calculations
is
  procedure Advanced_Calculate (D : in out Data)
  is
  begin
    Calculate (D);
  end Advanced_Calculate;

end Data_Processing.Advanced_Calculations;
```

This works fine as expected: we can refer to private child packages in the body of another package — as long as both packages belong to the same package tree.

### Outside the package tree

While we can use a with-clause of a private child package in the body of the Data_Processing package, we cannot do the same outside the package tree. For example, we cannot refer to it in the Test_Data_Processing procedure:

Listing 32: test_data_processing.adb

```ada
with Data_Processing; use Data_Processing;

with Data_Processing.Calculations;
use Data_Processing.Calculations;

procedure Test_Data_Processing is
  D : Data;
begin
  Calculate (D);
end Test_Data_Processing;
```

Build output

test_data_processing.adb:3:06: error: unit in with clause is private child unit
test_data_processing.adb:3:06: error: current unit must also have parent "Data_Processing"
gprbuild: *** compilation phase failed
As expected, we get a compilation error because Calculations is only accessible within the Data_Processing, but not in the Test_Data_Processing procedure.

The same restrictions apply to child packages of private packages. For example, if we implement a child package of the Calculations package — let's name it Calculations.Child —, we cannot refer to it in the Test_Data_Processing procedure:

```ada
package Data_Processing.Calculations.Child is
   procedure Process (D : in out Data);
end Data_Processing.Calculations.Child;
```

```ada
package body Data_Processing.Calculations.Child is
   procedure Process (D : in out Data) is
      begin
         Calculate(D);
      end Process;
end Data_Processing.Calculations.Child;
```

```ada
with Data_Processing; use Data_Processing;
procedure Test_Data_Processing is
   D : Data;
   begin
      Calculate(D);
   end Test_Data_Processing;
```

Again, as expected, we get an error because Calculations.Child — being a child of a private package — has the same restricted view as its parent package. Therefore, it cannot be visible in the Test_Data_Processing procedure as well. We'll discuss more about visibility later (page 712).

Note that subprograms can also be declared private. We'll see this in another section (page 729).
Important
We've discussed package renaming *in a previous section* (page 693). We can rename a package as a private package, too. For example:

Listing 36: driver_m1.ads
```ada
package Driver_M1 is
end Driver_M1;
```

Listing 37: drivers.ads
```ada
package Drivers
with Pure is
end Drivers;
```

Listing 38: drivers-m1.ads
```ada
with Driver_M1;
private package Drivers.M1 renames Driver_M1;
```

Obviously, Drivers.M1 has the same restrictions as any private package:

Listing 39: test_driver.adb
```ada
with Driver_M1;
with Drivers.M1;
procedure Test_Driver is
begin
  null;
end Test_Driver;
```

As expected, although we can have the Driver_M1 package in a with clause of the Test_Driver procedure, we cannot do the same in the case of the Drivers.M1 package because it is private.

In the Ada Reference Manual

27.1. Packages 703
27.1.3 Private with clauses

Definition and usage

A private with clause allows us to refer to a package in the private part of another package. For example, if we want to refer to package P in the private part of Data, we can write `private with P`:

```
Listing 40: p.ads
package P is
  type T is null record;
end P;
```

```
Listing 41: data.ads
private with P;
package Data is
type T2 is private;
private
  -- Information from P is
  -- visible here
type T2 is new P.T;
end Data;
```

```
Listing 42: main.adb
with Data; use Data;
procedure Main is
  A : T2;
begin
  null;
end Main;
```

As you can see in the example, as the information from P is available in the private part of Data, we can derive a new type T2 based on T from P. However, we cannot do the same in the visible part of Data:

---

Learning Ada

Listing 43: data.ads

```ada
private with P;

package Data is
  -- ERROR: information from P
  -- isn't visible here
  type T2 is new P.T;
end Data;
```

**Code block metadata**

MD5: b454e875f73432f5632a20ab40ae7da6

**Build output**

data.ads:8:19: error: "P" is not visible
data.ads:8:19: error: non-visible declaration at p.ads:1
gprbuild: *** compilation phase failed

Also, the information from P is available in the package body. For example, let’s declare a Process procedure in the P package and use it in the body of the Data package:

Listing 44: p.ads

```ada
package P is
  type T is null record;
  procedure Process (A : T) is null;
end P;
```

Listing 45: data.ads

```ada
private with P;

package Data is
  type T2 is private;
  procedure Process (A : T2);

private
  -- Information from P is
  -- visible here
  type T2 is new P.T;
end Data;
```

Listing 46: data.adb

```ada
package body Data is
  procedure Process (A : T2) is
```

(continues on next page)
begin
P.Process (P.T (A));
end Process;
end Data;

Listing 47: main.adb

with Data; use Data;
procedure Main is
A : T2;
begin
null;
end Main;

In the body of the Data, we can access information from the P package — as we do in the P.Process (P.T (A)) statement of the Process procedure.

Referring to private child package

There's one case where using a private with clause is the only way to refer to a package: when we want to refer to a private child package in another child package. For example, here we have a package P and its two child packages: Private_Child and Public_Child:

Listing 48: p.ads
package P is
end P;

Listing 49: p-private_child.ads
private package P.Private_Child is
type T is null record;
end P.Private_Child;

Listing 50: p-public_child.ads
private with P.Private_Child;
package P.Public_Child is
type T2 is private;
private
type T2 is new P.Private_Child.T;
end P.Public_Child;
Learning Ada

Listing 51: test_parent_child.adb

```ada
with P.Public_Child; use P.Public_Child;

procedure Test_Parent_Child is
  A : T2;
begin
  null;
end Test_Parent_Child;
```

**Code block metadata**

MD5: a6028416a957184be55a54f96a319e61

In this example, we're referring to the `P.Private_Child` package in the `P.Public_Child` package. As expected, this works fine. However, using a *normal* with clause doesn't work in this case:

Listing 52: p-public_child.ads

```ada
with P.Private_Child;

package P.Public_Child is
  type T2 is private;
private
  type T2 is new P.Private_Child.T;
end P.Public_Child;
```

**Code block metadata**

MD5: 2f32f29ecb4ae13bb4487c94d3bf18d9

**Build output**

p-public_child.ads:1:06: error: current unit must also be private descendant of "P"
gprbuild: *** compilation phase failed

This gives an error because the information from the `P.Private_Child`, being a private child package, cannot be accessed in the public part of another child package. In summary, unless both packages are private packages, it's only possible to access the information from a private package in the private part of a non-private child package.

**In the Ada Reference Manual**

- 10.1.2 Context Clauses - With Clauses

---

27.1.4 Limited Visibility

Sometimes, we might face the situation where two packages depend on information from each other. Let’s consider a package A that depends on a package B, and vice-versa:

```
Listing 53: a.ads
with B; use B;
package A is
  type T1 is record
    Value : T2;
  end record;
end A;
```

```
Listing 54: b.ads
with A; use A;
package B is
  type T2 is record
    Value : T1;
  end record;
end B;
```

Here, we have two mutually dependent types (page 418) T1 and T2, which are declared in two packages A and B that refer to each other. These with clauses constitute a circular dependency, so the compiler cannot compile either of those packages.

One way to solve this problem is by transforming this circular dependency into a partial dependency. We do this by limiting the visibility — using a limited with clause. To use a limited with clause for a package P, we simply write `limited with P`.

If a package A has limited visibility to a package B, then all types from package B are visible as if they had been declared as incomplete types (page 305). For the specific case of the previous source-code example, this would be the limited visibility to package B from package A’s perspective:

```
package B is
  -- Incomplete type
  type T2;
end B;
```
As we’ve seen previously,

- we cannot declare objects of incomplete types, but we can declare access types and anonymous access objects of incomplete types. Also,
- we can use anonymous access types to declare *mutually dependent types* (page 418).

Keeping this information in mind, we can now correct the previous code by using limited with clauses for package A and declaring the component of the T1 record using an anonymous access type:

### Listing 55: a.ads

```ada
limited with B;

package A is
  type T1 is record
    Ref : access B.T2;
  end record;
end A;
```

### Listing 56: b.ads

```ada
with A; use A;

package B is
  type T2 is record
    Value : T1;
  end record;
end B;
```

**Code block metadata**


MD5: 48591850665085a6fbb184f51b658a1b

As expected, we can now compile the code without issues.

Note that we can also use limited with clauses for both packages. If we do that, we must declare all components using anonymous access types:

### Listing 57: a.ads

```ada
limited with B;

package A is
  type T1 is record
    Ref : access B.T2;
  end record;
end A;
```

### Listing 58: b.ads

```ada
limited with A;

package B is
```

(continues on next page)
Now, both packages A and B have limited visibility to each other.

**In the Ada Reference Manual**

- 10.1.2 Context Clauses - With Clauses

**Limited visibility and private with clauses**

We can limit the visibility and use `private with clauses` (page 704) at the same time. For a package P, we do this by simply writing `limited private with P`.

Let's reuse the previous source-code example and convert types T1 and T2 to private types:

**Listing 59: a.ads**

```ada
limited private with B;

package A is
    type T1 is private;
    private
        -- Here, we have limited visibility
        -- of package B
        type T1 is record
            Ref : access B.T2;
        end record;
    end private;
end A;
```

**Listing 60: b.ads**

```ada
private with A;

package B is
    type T2 is private;
    private
        use A;
    end private;
end B;
```

---

-- Here, we have full visibility
-- of package A

type T2 is record
  Value : T1;
end record;
end B;

Limited visibility and other elements

It’s important to mention that the limited visibility we’ve been discussing so far is restricted
to type declarations — which are seen as incomplete types. In fact, when we use a limited
with clause, all other declarations have no visibility at all! For example, let’s say we have
a package Info that declares a constant Zero_Const and a function Zero_Func:

Listing 61: info.ads

package Info is
  function Zero_Func return Integer is (0);
  Zero_Const : constant := 0;
end Info;

Also, let’s say we want to use the information (from package Info) in package A. If we have
limited visibility to package Info, however, this information won’t be visible. For example:

Listing 62: a.ads

limited private with Info;

package A is
  type T1 is private;
  private
    type T1 is record
      V : Integer := Info.Zero_Const;
    end record;
  end private;
end A;
As expected, compilation fails because of the limited visibility — as Zero_Const and Zero_Func from the Info package are not visible in the private part of A. (Of course, if we revert to full visibility by simply removing the limited keyword from the example, the code compiles just fine.)

27.1.5 Visibility

In the previous sections, we already discussed visibility from various angles. However, it can be interesting to recapitulate this information with the help of diagrams that illustrate the different parts of a package and its relation with other units.

Automatic visibility

First, let's consider we have a package A, its children (A.G and A.H), and the grandchild A.G.T. As we've seen before, information of a parent package is automatically visible in its children. The following diagrams illustrates this:
Because of this automatic visibility, many with clauses would be redundant in child packages. For example, we don't have to write `with A; package A.G is`, since the specification of package A is already visible in its child packages.

If we focus on package A.G (highlighted in the figure above), we see that it only has automatic visibility to its parent A, but not its child A.G.T. Also, it doesn't have visibility to its sibling A.H.

**With clauses and visibility**

In the rest of this section, we discuss all the situations where using with clauses is necessary to access the information of a package. Let's consider this example where we refer to a package B in the specification of a package A (using `with B`):
As we already know, the information from the public part of package B is visible in the public part of package A. In addition to that, it's also visible in the private part and in the body of package A. This is indicated by the dotted green arrows in the figure above.

Now, let's see the case where we refer to package B in the private part of package A (using `private with B`):
Here, the information is visible in the private part of package A, as well as in its body. Finally, let's see the case where we refer to package B in the body of package A:
Here, the information is only visible in the body of package A.

**Circular dependency**

Let's return to package A and its descendants. As we've seen in previous sections, we cannot refer to a child package in the specification of its parent package because that would constitute circular dependency. (For example, we cannot write `with A.G; package A is`.) This situation — which causes a compilation error — is indicated by the red arrows in the figure below:
Note that referring to the child package A.G in the body of its parent is perfectly fine.
Private packages

The previous examples of this section only showed public packages. As we've seen before, we cannot refer to private packages outside of a package hierarchy, as we can see in the following example where we try to refer to package A and its descendants in the Test procedure:
As indicated by the red arrows, we cannot refer to the private child packages of A in the Test procedure, only the public child packages. Within the package hierarchy itself, we
cannot refer to the private package A.G in public sibling packages. For example:

Here, we cannot refer to the private package A.G in the public package A.H — as indicated by the red arrow. However, we can refer to the private package A.G in other private packages, such as A.I — as indicated by the green arrows.

### 27.1.6 Use type clause

Back in the *Introduction to Ada course* (page 37), we saw that use clauses provide direct visibility — in the scope where they're used — to the content of a package's visible part. For example, consider this simple procedure:

```
Listing 63: display_message.adb

with Ada.Text_IO;
procedure Display_Message is
begin
  Ada.Text_IO.Put_Line ("Hello World!");
end Display_Message;
```

**Code block metadata**
Hello World!

By adding use Ada.Text_IO to this code, we make the visible part of the Ada.Text_IO package directly visible in the scope of the Display_Message procedure, so we can now just write Put_Line instead of Ada.Text_IO.Put_Line:

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Display_Message is
begin
   Put_Line ("Hello World!");
end Display_Message;
```

In this section, we discuss another example of use clauses. In addition, we introduce two specific forms of use clauses: use type and use all type.

In the Ada Reference Manual

- 8.4 Use Clauses

Another use clause example

Let's now consider a simple package called Points, which contains the declaration of the Point type and two primitive: an Init function and an addition operator.

```ada
package Points is
   type Point is private;
   function Init return Point;
   function "+" (P : Point; I : Integer) return Point;
private
   type Point is record
      X, Y : Integer;
   end record;
```

(continues on next page)
function Init return Point is (0, 0);

function "+" (P : Point;
        I : Integer) return Point is
        (P.X + I, P.Y + I);

end Points;

Code block metadata

Project: Courses.Advanced_Ada.Modular_ProgPackages.Use_Type_Clause.Use_Type_Clause
MD5: 1a43740d7231a3cc497e778866a12c55

We can implement a simple procedure that makes use of this package:

Listing 66: show_point.adb

with Points; use Points;

procedure Show_Point is
    P : Point;
begin
    P := Init;
    P := P + 1;
end Show_Point;

Code block metadata

Project: Courses.Advanced_Ada.Modular_ProgPackages.Use_Type_Clause.Use_Type_Clause
MD5: f5d44dd1fee8cf4d1a7e730f9a7c64cc

Here, we have a use clause, so we have direct visibility to the content of Points's visible part.

Visibility and Readability

In certain situations, however, we might want to avoid the use clause. If that's the case, we can rewrite the previous implementation by removing the use clause and specifying the Points package in the prefixed form:

Listing 67: show_point.adb

with Points;

procedure Show_Point is
    P : Points.Point;
begin
    P := Points.Init;
    P := Points. "+" (P, 1);
end Show_Point;

Code block metadata

Project: Courses.Advanced_Ada.Modular_ProgPackages.Use_Type_Clause.Use_Type_Clause
MD5: ca896b456a90c19b29ec4f262144c131

Although this code is correct, it might be difficult to read, as we have to specify the package whenever we're referring to a type or a subprogram from that package. Even worse: we now have to write operators in the prefixed form — such as Points."+" (P, 1).
use type

As a compromise, we can have direct visibility to the operators of a certain type. We do this by using a use clause in the form `use` type. This allows us to simplify the previous example:

```
Listing 68: show_point.adb

with Points;

procedure Show_Point is
  use type Points.Point;
  P : Points.Point;
begin
  P := Points.Init;
  P := P + 1;
end Show_Point;
```

Note that `use` type just gives us direct visibility to the operators of a certain type, but not other primitives. For this reason, we still have to write `Points.Init` in the code example.

use all type

If we want to have direct visibility to all primitives of a certain type (and not just its operators), we need to write a use clause in the form `use all type`. This allows us to simplify the previous example even further:
Now, we've removed the prefix from all operations on the P variable.

### 27.1.7 Use clauses and naming conflicts

Visibility issues may arise when we have multiple use clauses. For instance, we might have types with the same name declared in multiple packages. This constitutes a naming conflict; in this case, the types become hidden — so they're not directly visible anymore, even if we have a use clause.

**In the Ada Reference Manual**

- 8.4 Use Clauses\(^{167}\)

**Code example**

Let's start with a code example. First, we declare and implement a generic procedure that shows the value of a Complex object:

Listing 70: show_any_complex.ads

```ada
with Ada.Numerics.Generic_Complex_Types;

generic
package Complex_Types is new
Ada.Numerics.Generic_Complex_Types (<>);
procedure Show_Any_Complex
(Msg : String;
Val : Complex_Types.Complex);
```

Listing 71: show_any_complex.adb

```ada
with Ada.Text_IO;
with Ada.Text_IO.Complex_IO;

procedure Show_Any_Complex
(Msg : String;
(continues on next page)
```

Val : Complex_Types.Complex

is

package Complex_Float_Types_Io is new
Ada.Text_IO.Complex_IO (Complex_Types);
use Complex_Float_Types_Io;

use Ada.Text_IO;

begin
Put (Msg & " ");
Put (Val);
New_Line;
end Show_Any_Complex;

Code block metadata

Use_Type_Clause_Complex_Types
MD5: 2527291906d3a600eecd6d36e4359c1a

Then, we implement a test procedure where we declare the Complex_Float_Types package as an instance of the Generic_Complex_Types package:

Listing 72: show_use.adb

with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Generic_Complex_Types;
with Show_Any_Complex;

procedure Show_Use is
package Complex_Float_Types is new
Ada.Numerics.Generic_Complex_Types
(Real => Float);
use Complex_Float_Types;

procedure Show_Complex_Float is new
Show_Any_Complex (Complex_Float_Types);
C, D, X : Complex;
begi
C := Compose_From_Polar (3.0, Pi / 2.0);
D := Compose_From_Polar (5.0, Pi / 2.0);
X := C + D;

Show_Complex_Float ("C:" , C);
Show_Complex_Float ("D:" , D);
Show_Complex_Float ("X:" , X);
end Show_Use;

Code block metadata

Use_Type_Clause_Complex_Types
MD5: cc2a612c9b884539f33154680854a4c82

Runtime output
C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)
Learning Ada

In this example, we declare variables of the Complex type, initialize them and use them in operations. Note that we have direct visibility to the package instance because we've added a simple use clause after the package instantiation — see use Complex_Float_Types in the example.

Naming conflict

Now, let's add the declaration of the Complex_Long_Float_Types package — a second instantiation of the Generic_Complex_Types package — to the code example:

Listing 73: show_use.adb

```ada
with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Generic_Complex_Types;
with Show_Any_Complex;

procedure Show_Use is
  package Complex_Float_Types is new
    Ada.Numerics.Generic_Complex_Types
    (Real => Float);
  use Complex_Float_Types;

  package Complex_Long_Float_Types is new
    Ada.Numerics.Generic_Complex_Types
    (Real => Long_Float);
  use Complex_Long_Float_Types;

  procedure Show_Complex_Float is new
    Show_Any_Complex (Complex_Float_Types);

  C, D, X : Complex;
  -- ^ ERROR: type is hidden!
begin
  C := Compose_From_Polar (3.0, Pi / 2.0);
  D := Compose_From_Polar (5.0, Pi / 2.0);
  X := C + D;
  Show_Complex_Float ("C:", C);
  Show_Complex_Float ("D:", D);
  Show_Complex_Float ("X:", X);
end Show_Use;
```

Code block metadata

Use_Type_Clause_Complex_Types
MD5: 30b562e2f81ae62912ec4e067150d5cd

Build output

```
gprbuild: *** compilation phase failed
```

This example doesn't compile because we have direct visibility to both Com-
plex_Float_Types and Complex_Long_Float_Types packages, and both of them declare the Complex type. In this case, the type declaration becomes hidden, as the compiler cannot decide which declaration of Complex it should take.

**Circumventing naming conflicts**

As we know, a simple fix for this compilation error is to add the package prefix in the variable declaration:

Listing 74: show_use.adb

```ada
with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Generic_Complex_Types;
with Show_Any_Complex;

procedure Show_Use is
  package Complex_Float_Types is new Ada.Numerics.Generic_Complex_Types (Real => Float);
  use Complex_Float_Types;

  package Complex_Long_Float_Types is new Ada.Numerics.Generic_Complex_Types (Real => Long_Float);
  use Complex_Long_Float_Types;

  procedure Show_Complex_Float is new Show_Any_Complex (Complex_Float_Types);

  C, D, X : Complex_Float_Types.Complex;
  begin
    C := Compose_From_Polar (3.0, Pi / 2.0);
    D := Compose_From_Polar (5.0, Pi / 2.0);
    X := C + D;
    Show_Complex_Float ("C:", C);
    Show_Complex_Float ("D:", D);
    Show_Complex_Float ("X:", X);
  end Show_Use;
```

**Code block metadata**

Use Type Clause_Complex_Types
MD5: 0b3285364ea0188a678db2fc406741b8

**Runtime output**

C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)

Another possibility is to write a use clause in the form **use all type**:

Listing 75: show_use.adb

```ada
with Ada.Numerics; use Ada.Numerics;
(continues on next page)
```
with Ada.Numerics.Generic_Complex_Types;
with Show_Any_Complex;

procedure Show_Use is
  package Complex_Float_Types is new 
    Ada.Numerics.Generic_Complex_Types 
    (Real => Float);
  use all type Complex_Float_Types.Complex;

  package Complex_Long_Float_Types is new 
    Ada.Numerics.Generic_Complex_Types 
    (Real => Long_Float);
  use all type Complex_Long_Float_Types.Complex;

  procedure Show_Complex_Float is new 
    Show_Any_Complex (Complex_Float_Types);

  C, D, X : Complex_Float_Types.Complex;
begin
  C := Compose_From_Polar (3.0, Pi / 2.0);
  D := Compose_From_Polar (5.0, Pi / 2.0);
  X := C + D;
  Show_Complex_Float ("C:", C);
  Show_Complex_Float ("D:", D);
  Show_Complex_Float ("X:", X);
end Show_Use;

For the sake of completeness, let's declare and use variables of both Complex types:

Listing 76: show_use.adb
procedure Show_Complex_Float is new
    Show_Any_Complex (Complex_Float_Types);

procedure Show_Complex_Long_Float is new
    Show_Any_Complex (Complex_Long_Float_Types);

C, D, X : Complex_Float_Types.Complex;
E, F, Y : Complex_Long_Float_Types.Complex;
begin
    C := Compose_From_Polar (3.0, Pi / 2.0);
    D := Compose_From_Polar (5.0, Pi / 2.0);
    X := C + D;
    Show_Complex_Float ("C:", C);
    Show_Complex_Float ("D:", D);
    Show_Complex_Float ("X:", X);

    E := Compose_From_Polar (3.0, Pi / 2.0);
    F := Compose_From_Polar (5.0, Pi / 2.0);
    Y := E + F;
    Show_Complex_Long_Float ("E:", E);
    Show_Complex_Long_Float ("F:", F);
    Show_Complex_Long_Float ("Y:", Y);
end Show_Use;

Code block metadata

Use_Type_Clause_Complex_Types
MD5: 48f31250116f107d3143703debb3107d

Runtime output

C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)
E: ( 1.83697019872103E-16, 3.00000000000000E+00)
F: ( 3.0616169786838E-16, 5.00000000000000E+00)
Y: ( 4.89858719658941E-16, 8.00000000000000E+00)

As expected, the code compiles correctly.

27.2 Subprograms and Modularity

27.2.1 Private subprograms

We've seen previously (page 696) that we can declare private packages. Because packages
and subprograms can both be library units, we can declare private subprograms as well.
We do this by using the private keyword. For example:

Listing 77: test.ads

private procedure Test;
Listing 78: test.adb

procedure Test is
begin
  null;
end Test;

Code block metadata
MD5: 2ea1770a5fd5dee40f015b9d33d2f309

Such a subprogram as the one above isn't really useful. For example, we cannot write a with clause that refers to the Test procedure, as it's not visible anywhere:

Listing 79: show_test.adb

with Test;

procedure Show_Test is
begin
  Test;
end Show_Test;

Code block metadata
MD5: 0702378a034f65a69a4c5b5258f7b32e

Build output

show_test.adb:1:06: error: current unit must also be private descendant of "Standard"
gprbuild: *** compilation phase failed

As expected, since Test is private, we get a compilation error because this procedure cannot be referenced in the Show_Test procedure.

In the Ada Reference Manual

- 10.1.1 Compilation Units - Library Units
- 10.1.2 Context Clauses - With Clauses

Private subprograms of a package

A more useful example is to declare private subprograms of a package. For example:

Listing 80: data_processing.ads

```ada
package Data_Processing is
  type Data is private;
  procedure Process (D : in out Data);
private
  type Data is record
    F : Float;
  end record;
end Data_Processing;
```

Listing 81: data_processing.adb

```ada
with Data_Processing.Calculate;
package body Data_Processing is
  procedure Process (D : in out Data) is
    begin
      Calculate (D);
    end Process;
end Data_Processing;
```

Listing 82: data_processing-calculate.ads

```ada
private
procedure Data_Processing.Calculate (D : in out Data);
```

Listing 83: data_processing-calculate.adb

```ada
procedure Data_Processing.Calculate (D : in out Data) is
  begin
    -- Dummy implementation...
    D.F := 0.0;
  end Data_Processing.Calculate;
```
In this example, we declare Calculate as a private procedure of the Data_Processing package. Therefore, it's visible in that package (but not in the Test_Data_Processing procedure). Also, in the Calculate procedure, we're able to initialize the private component F of the D object because the child subprogram has access to the private part of its parent package.

### Private subprograms and private packages

We can also use private subprograms to test private packages. As we know, in most cases, we cannot access private packages in external clients — such as external subprograms. However, by declaring a subprogram private, we're allowed to access private packages. This can be very useful to create applications that we can use to test private packages. (Note that these applications must be library-level parameterless subprograms, because only those can be main programs.)

Let's see an example:

---

### Listing 85: private_data_processing.ads

```ada
private package Private_Data_Processing is

  type Data is private;

  procedure Process (D : in out Data);

private

  type Data is record
    F : Float;
  end record;

end Private_Data_Processing;
```

---

### Listing 86: private_data_processing.adb

```ada
package body Private_Data_Processing is

  procedure Process (D : in out Data) is begin
    D.F := 0.0;
  end Process;

end Private_Data_Processing;
```
private procedure Test_Private_Data_Processing;

with Private_Data_Processing;
use Private_Data_Processing;

procedure Test_Private_Data_Processing is
  D : Data;
begin
  Process (D);
end Test_Private_Data_Processing;

In this code example, we have the private Private_Data_Processing package. In order to test it, we implement the private procedure Test_Private_Data_Processing. The fact that this procedure is private allows us to use the Private_Data_Processing package as if it was a non-private package. We then use the private Test_Private_Data_Processing procedure as our main application, so we can run it to test application the private package.

**Child subprograms of private packages**

We could also implement the Test subprogram that we use to test a private package P as a child subprogram of that package. In other words, we could write a procedure P.Test and use it as our main application. The advantage here is that this allows us to access the private part of the parent package P in the test procedure.

Let's rewrite the Test_Private_Data_Processing procedure from the previous example as the child procedure Private_Data_Processing.Test:

```ada
private package Private_Data_Processing is
  type Data is private;
  procedure Process (D : in out Data);
private
  type Data is record
    F : Float;
  end record;
end Private_Data_Processing;
```

```ada
package body Private_Data_Processing is
  procedure Process (D : in out Data) is
begin
  -- (continues on next page)
```
In this code example, we now implement the Test procedure as a child of the `Private_Data_Processing` package. In this procedure, we're able to initialize the private component \( F \) of the \( D \) object. As we know, this initialization of a private component wouldn't be possible if Test wasn't a child procedure. (For instance, writing such an initialization in the `Test_Private_Data_Processing` procedure from the previous code example would trigger a compilation error.)
28.1 Access Types

We discussed access types back in the *Introduction to Ada course* (page 95). In this chapter, we discuss further details about access types and techniques when using them. Before we dig into details, however, we’re going to make sure we understand the terminology.

28.1.1 Access types: Terminology

In this section, we discuss some of the terminology associated with access types. Usually, the terms used in Ada when discussing references and dynamic memory allocation are different than the ones you might encounter in other languages, so it’s necessary you understand what each term means.

**Access type, designated subtype and profile**

The first term we encounter is (obviously) *access type*, which is a type that provides us access to an object or a subprogram. We declare access types by using the `access` keyword:

```ada
package Show_Access_Type_Declaration is
  -- Declaring access types:
  -- Access-to-object type
  type Integer_Access is access Integer;
  -- Access-to-subprogram type
  type Init_Integer_Access is access
    function return Integer;
end Show_Access_Type_Declaration;
```

Here, we’re declaring two access types: the access-to-object type `Integer_Access` and the access-to-subprogram type `Init_Integer_Access`. (We discuss access-to-subprogram types *later on* (page 820)).
Learning Ada

In the declaration of an access type, we always specify — after the access keyword — the kind of thing we want to designate. In the case of an access-to-object type declaration, we declare a subtype we want to access, which is known as the designated subtype of an access type. In the case of an access-to-subprogram type declaration, the subprogram prototype is known as the designated profile.

In our previous code example, Integer is the designated subtype of the Integer_Access type, and function return Integer is the designated profile of the Init_Integer_Access type.

Important

In contrast to other programming languages, an access type is not a pointer, and it doesn't just indicate an address in memory. We discuss more about addresses (page 849) later on.

Access object and designated object

We use an access-to-object type by first declaring a variable (or constant) of an access type and then allocating an object. (This is actually just one way of using access types; we discuss other methods later in this chapter.) The actual variable or constant of an access type is called access object, while the object we allocate (via new) is the designated object.

For example:

```
procedure Show_Simple_Allocation is

  -- Access-to-object type
  type Integer_Access is access Integer;

  -- Access object
  I1 : Integer_Access;

begin
  I1 := new Integer;
  -- ^^^^^^^^^^^ allocating an object,
  --             which becomes the designated
  --             object for I1

end Show_Simple_Allocation;
```

In this example, I1 is an access object and the object allocated via new Integer is its designated object.
Access value and designated value

An access object and a designated (allocated) object, both store values. The value of an access object is the access value and the value of a designated object is the designated value. For example:

Listing 3: show_values.adb

```ada
procedure Show_Values is
  -- Access-to-object type
  type Integer_Access is access Integer;
  I1, I2, I3 : Integer_Access;
begin
  I1 := new Integer;
  I3 := new Integer;
  -- Copying the access value of I1 to I2
  I2 := I1;
  -- Copying the designated value of I1
  I3.all := I1.all;
end Show_Values;
```

Code block metadata

MD5: a152ee813b8ed9fad985cf4e2c25d847

In this example, the assignment `I2 := I1` copies the access value of `I1` to `I2`. The assignment `I3.all := I1.all` copies `I1`'s designated value to `I3`'s designated object. (As we already know, `.all` is used to dereference an access object. We discuss this topic again later in this chapter (page 766).)

In the Ada Reference Manual

- 3.10 Access Types

28.1.2 Access types: Allocation

Ada makes the distinction between pool-specific and general access types, as we'll discuss in this section. Before doing so, however, let's talk about memory allocation.

In general terms, memory can be allocated dynamically on the heap or statically on the stack. (Strictly speaking, both are dynamic allocations, in that they occur at run-time with amounts not previously specified.) For example:

Listing 4: show_simple_allocation.adb

```ada
procedure Show_Simple_Allocation is
  -- Declaring access type:
  type Integer_Access is access Integer;
```

Code block metadata

(continues on next page)

170 http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

28.1. Access Types
When we allocate an object on the heap via `new`, the allocation happens in a memory pool that is associated with the access type. In our code example, there's a memory pool associated with the `Integer_Access` type, and each `new Integer` allocates a new integer object in that pool. Therefore, access types of this kind are called pool-specific access types. (We discuss more about these types (page 740) later.)

It is also possible to access objects that were allocated on the stack. To do that, however, we cannot use pool-specific access types because — as the name suggests — they're only allowed to access objects that were allocated in the specific pool associated with the type. Instead, we have to use general access types in this case:

```
procedure Show_General_Access_Type is
  -- Declaring general access type:
type Integer_Access is access all Integer;
  -- Declaring access object:
  A1 : Integer_Access;
  -- Allocating an Integer object on the stack:
  I : aliased Integer;
begin
  -- Getting access to an Integer object that
  -- was allocated on the stack
  A1 := I'Access;
end Show_General_Access_Type;
```
In this example, we declare the general access type `Integer_Access` and the access object `A1`. To initialize `A1`, we write `I'Access` to get access to an integer object `I` that was allocated on the stack. (For the moment, don't worry much about these details: we'll talk about general access types again when we introduce the topic of *aliased objects* (page 778) later on.)

**For further reading...**

Note that it is possible to use general access types to allocate objects on the heap:

Listing 6: show_simple_allocation.adb

```ada
procedure Show_Simple_Allocation is
  -- Declaring general access type:
  type Integer_Access is access all Integer;

  -- Declaring access object:
  A1 : Integer_Access;

begin
  -- Allocating an Integer object on the heap
  -- and initializing an access object of
  -- the general access type Integer_Access.
  A1 := new Integer;
end Show_Simple_Allocation;
```

**Important**

In many code examples, we have used the `Integer` type as the designated subtype of the access types — by writing `access Integer`. Although we have used this specific scalar type, we aren't really limited to those types. In fact, we can use *any type* as the designated subtype, including user-defined types, composite types, task types and protected types.

**In the Ada Reference Manual**

- 3.10 Access Types\(^1\)

\(^1\) [http://www.ada-auth.org/standards/22rm/html/RM-3-10.html](http://www.ada-auth.org/standards/22rm/html/RM-3-10.html)
Pool-specific access types

We've already discussed many aspects about pool-specific access types. In this section, we recapitulate some of those aspects, and discuss some new details that haven't seen yet.

As we know, we cannot directly assign an object Distance_Miles of type Miles to an object Distance_Meters of type Meters, even if both share a common Float type ancestor. The assignment is only possible if we perform a type conversion from Miles to Meters, or vice-versa — e.g.: Distance_Meters := Meters (Distance_Miles) * Miles_To_Meters_Factor.

Similarly, in the case of pool-specific access types, a direct assignment between objects of different access types isn't possible. However, even if both access types have the same designated subtype (let's say, they are both declared using is access Integer), it's still not possible to perform a type conversion between those access types. The only situation when an access type conversion is allowed is when both types have a common ancestor.

Let's see an example:

Listing 7: show_simple_allocation.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Allocation is
  -- Declaring pool-specific access type:
  type Integer_Access_1 is access Integer;
  type Integer_Access_2 is access Integer;
  type Integer_Access_2B is new Integer_Access_2;

  -- Declaring access object:
  A1 : Integer_Access_1;
  A2 : Integer_Access_2;
  A2B : Integer_Access_2B;

begin
  A1 := new Integer;
  Put_Line ("A1 : " & A1'Image);
  Put_Line ("Pool: " & A1'Storage_Pool'Image);

  A2 := new Integer;
  Put_Line ("A2: " & A2'Image);
  Put_Line ("Pool: " & A2'Storage_Pool'Image);

  -- ERROR: Cannot directly assign access values
  -- for objects of unrelated access
  -- types; also, cannot convert between
  -- these types.

  A1 := A2;
  A1 := Integer_Access_1 (A2);

  A2B := Integer_Access_2B (A2);
  Put_Line ("A2B: " & A2B'Image);
  Put_Line ("Pool: " & A2B'Storage_Pool'Image);
end Show_Simple_Allocation;
```

Code block metadata
In this example, we declare three access types: `Integer_Access_1`, `Integer_Access_2` and `Integer_Access_2B`. Also, the `Integer_Access_2B` type is derived from the `Integer_Access_2` type. Therefore, we can convert an object of `Integer_Access_2` type to the `Integer_Access_2B` type — we do this in the `A2B := Integer_Access_2B (A2)` assignment. However, we cannot directly assign to or convert between unrelated types such as `Integer_Access_1` and `Integer_Access_2`. (We would get a compilation error if we included the `A1 := A2` or the `A1 := Integer_Access_1 (A2)` assignment.)

### Important

Remember that:

- As mentioned in the *Introduction to Ada course* (page 97):
  - an access type can be unconstrained, but the actual object allocation must be constrained;
  - we can use a *qualified expression* (page 332) to allocate an object.

- We can use the `Storage_Size` attribute to limit the size of the memory pool associated with an access type, as discussed previously in the *section about storage size* (page 353).

- When running out of memory while allocating via `new`, we get a `Storage_Error` exception because of the *storage check* (page 673).

For example:

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Array_Allocation is
  -- Unconstrained array type:
  type Integer_Array is
    array (Positive range <>) of Integer;

  -- Access type with unconstrained designated subtype and limited storage size.
  type Integer_Array_Access is
    access Integer_Array
      with Storage_Size => 128;
```

(continues on next page)
-- An access object:
A1 : Integer_Array_Access;

procedure Show_Info
(IAA : Integer_Array_Access)
is
begin
  Put_Line ("Allocated: " & IAA'Image);
  Put_Line ("Length: " & IAA.all'Length'image);
  Put_Line ("Values: " & IAA.all'image);
end Show_Info;

begin
  -- Allocating an integer array with
  -- constrained range on the heap:
  A1 := new Integer_Array (1 .. 3);
  A1.all := [others => 42];
  Show_Info (A1);
  -- Allocating an integer array on the
  -- heap using a qualified expression:
  A1 := new Integer_Array'(5, 10);
  Show_Info (A1);
  -- A third allocation fails at run time
  -- because of the constrained storage
  -- size:
  A1 := new Integer_Array (1 .. 100);
  Show_Info (A1);
exception
  when Storage_Error =>
    Put_Line ("Out of memory!");
end Show_Array_Allocation;

Multiple allocation

Up to now, we have seen examples of allocating a single object on the heap. It’s possible to allocate multiple objects at once as well — i.e. syntactic sugar is available to simplify the code that performs this allocation. For example:

Listing 9: show_access_array_allocation.adb

pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Access_Array_Allocation is
  type Integer_Access is access Integer;
  type Integer_Access_Array is
    array (Positive range <>) of Integer_Access;
  -- An array of access objects:
Arr : Integer_Access_Array (1 .. 10); (continues on next page)
begin
   -- Allocating 10 access objects and
   -- initializing the corresponding designated
   -- object with zero:
   --
   Arr := (others => new Integer'(0));
   -- Same as:
   for I in Arr'Range loop
      Arr (I) := new Integer'(0);
   end loop;
   Put_Line ("Arr: " & Arr'Image);
   Put_Line ("Arr (designated values): ");
   for E of Arr loop
      Put (E.all'Image);
   end loop;
end Show_Access_Array_Allocation;

In this example, we have the access type Integer_Access and an array type of this access
type (Integer_Access_Array). We also declare an array Arr of Integer_Access_Array
type. This means that each component of Arr is an access object. We allocate all ten com-
ponents of the Arr array by simply writing Arr := (others => new Integer). This array aggregate
(page 450) is syntactic sugar for a loop over Arr that allocates each component.
(Not that, by writing Arr := (others => new Integer'(0)), we’re also initializing the
designated objects with zero.)

Let’s see another code example, this time with task types:

Listing 10: workers.ads

package Workers is
   task type Worker is
      entry Start (Id : Positive);
      entry Stop;
   end Worker;

   type Worker_Access is access Worker;

   type Worker_Array is
      array (Positive range <>) of Worker_Access;
end Workers;
end Workers;

Listing 11: workers.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Workers is

  task body Worker is
    Id: Positive;
  begin
    accept Start (Id: Positive) do
      Worker.Id := Id;
      Put_Line ("Started Worker #" & Id'Image);
    end Start;

    accept Stop;

    Put_Line ("Stopped Worker #" & Id'Image);
  end Worker;

end Workers;

Listing 12: show_workers.adb

with Ada.Text_IO; use Ada.Text_IO;

with Workers; use Workers;

procedure Show_Workers is
  Worker_Arr: Worker_Array (1 .. 20);
  begin
    -- Allocating 20 workers at once:
    Worker_Arr := (others => new Worker);
    for I in Worker_Arr'Range loop
      Worker_Arr (I).Start (I);
    end loop;

    Put_Line ("Some processing..."');
    delay 1.0;

    for W of Worker_Arr loop
      W.Stop;
    end loop;
  end Show_Workers;

Code block metadata
MD5: d29e3d56585f8d9a63b805c680e5dc54
Runtime output
In this example, we declare the task type Worker, the access type Worker_Access and an array of access to tasks Worker_Array. Using this approach, a task is only created when we allocate an individual component of an array of Worker_Array type. Thus, when we declare the Worker_Arr array in this example, we're only preparing a container of 20 workers, but we don't have any actual tasks yet. We bring the 20 tasks into existence by writing Worker_Arr := (others => new Worker).

### 28.1.3 Discriminants as Access Values

We can use access types when declaring discriminants. Let's see an example:

```ada
package Custom_Recs is

  -- Declaring an access type:
type Integer_Access is access Integer;

(continues on next page)
```

28.1. Access Types
--- Declaring a discriminant with this
--- access type:

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Custom_Recs is

    procedure Show (R : Rec) is
    begin
        Put_Line ("R.IA = " & Integer'Image (R.IA.all));
        Put_Line ("R.I = " & Integer'Image (R.I));
    end Show;

end Custom_Recs;
```

Listing 14: custom_recs.adb

```ada
procedure Show (R : Rec);
end Custom_Recs;
```

Listing 15: show_discriminants_as_access_values.adb

```ada
with Custom_Recs; use Custom_Recs;

procedure Show_Discriminants_As_Access_Values is
    IA : constant Integer_Access :=
        new Integer'(10);
    R : Rec (IA);

begin
    Show (R);
    IA.all := 20;
    R.I := 30;
    Show (R);
    -- As expected, we cannot change the
    -- discriminant. The following line is
    -- triggers a compilation error:
    --
    -- R.IA := new Integer;

end Show_Discriminants_As_Access_Values;
```

Code block metadata


MD5: c7850acefd8e5227f4be654faed13055
In the `Custom_Recs` package from this example, we declare the access type `Integer_Access`. We then use this type to declare the discriminant (IA) of the `Rec` type. In the `Show_Discriminants_As_Access_Values` procedure, we see that (as expected) we cannot change the discriminant of an object of `Rec` type: an assignment such as `R.IA := new Integer` would trigger a compilation error.

Note that we can use a default for the discriminant:

```
package Custom_Recs is
  type Integer_Access is access Integer;
  type Rec (IA : Integer_Access := new Integer'(0)) is
    record
      I : Integer := IA.all;
    end record;
  procedure Show (R : Rec);
end Custom_Recs;
```

```
with Custom_Recs; use Custom_Recs;
procedure Show_Discriminants_As_Access_Values is
  R1 : Rec; -- ^^ -- no discriminant: use default
  R2 : Rec (new Integer'(20)); -- allocating an unnamed integer object
begin
  Show (R1);
  Show (R2);
end Show_Discriminants_As_Access_Values;
```
Here, we've changed the declaration of the Rec type to allocate an integer object if the type's discriminant isn't provided — we can see this in the declaration of the R1 object in the Show_Discriminants_As_Access_Values procedure. Also, in this procedure, we're allocating an unnamed integer object in the declaration of R2.

### In the Ada Reference Manual

- [3.10 Access Types](http://www.ada-auth.org/standards/22rm/html/RM-3-10.html)
- [3.7.1 Discriminant Constraints](http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html)

### Unconstrained type as designated subtype

Notice that we were using a scalar type as the designated subtype of the Integer_Access type. We could have used an unconstrained type as well. In fact, this is often used for the sake of having the effect of an unconstrained discriminant type.

Let's see an example:

#### Listing 18: persons.ads

```ada
package Persons is
  -- Declaring an access type whose
  -- designated subtype is unconstrained:
  type String_Access is access String;

  -- Declaring a discriminant with this
  -- access type:
  type Person (Name : String_Access) is record
    Age : Integer;
  end record;

  procedure Show (P : Person);
end Persons;
```

#### Listing 19: persons.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Persons is

  procedure Show (P : Person) is
    begin
      Put_Line ("Name = ", P.Name.all);
      Put_Line ("Age = ", Integer"Image (P.Age));
    end Show;
end Persons;
```

---

173 http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html
Listing 20: show_person.adb

```ada
with Persons; use Persons;

procedure Show_Person is
    P : Person (new String("John"));
begin
    P.Age := 30;
    Show (P);
end Show_Person;
```

**Code block metadata**

MD5: 9b1109d076b6f06632c8685a41616210

**Runtime output**

Name = John
Age = 30

In this example, the discriminant of the Person type has an unconstrained designated type. In the Show_Person procedure, we declare the P object and specify the constraints of the allocated string object — in this case, a four-character string initialized with the name "John".

**For further reading...**

In the previous code example, we used an array — actually, a string — to demonstrate the advantage of using discriminants as access values, for we can use an unconstrained type as the designated subtype. In fact, as we discussed earlier in another chapter (page 296), we can only use discrete types (or access types) as discriminants. Therefore, you wouldn’t be able to use a string, for example, directly as a discriminant without using access types:

Listing 21: persons.ads

```ada
package Persons is

    -- ERROR: Declaring a discriminant with an unconstrained type:
    type Person (Name : String) is record
        Age : Integer;
    end record;

end Persons;
```

**Code block metadata**

MD5: 4144852aaf95da62bc4781b1e8dc2717

**Build output**

persons.ads:5:24: error: discriminants must have a discrete or access type
gprbuild: *** compilation phase failed

As expected, compilation fails for this code because the discriminant of the Person type is indefinite.

However, the advantage of discriminants as access values isn't restricted to being able to

---

28.1. Access Types 749
use unconstrained types such as arrays: we could really use any type as the designated subtype! In fact, we can generalized this to:

Listing 22: gen_custom_recs.ads

```ada
generic
  type T (<>);  -- any type
  type T_Access is access T;
package Gen_Custom_Recs is
  -- Declare a type whose discriminant D can
  -- access any type:
  type T_Rec (D : T_Access) is null record;
end Gen_Custom_Recs;
```

Listing 23: custom_recs.ads

```ada
with Gen_Custom_Recs;

package Custom_Recs is
  type Incomp;  -- Incomplete type declaration!
  type Incomp_Access is access Incomp;
  -- Instantiating package using
  -- incomplete type Incomp:
  package Inst is new
    Gen_Custom_Recs
      (T => Incomp,
       T_Access => Incomp_Access);
  subtype Rec is Inst.T_Rec;
  -- At this point, Rec (Inst.T_Rec) uses
  -- an incomplete type as the designated
  -- subtype of its discriminant type
  procedure Show (R : Rec) is null;
  -- Now, we complete the Incomp type:
  type Incomp (B : Boolean := True) is private;
private
  -- Finally, we have the full view of the
  -- Incomp type:
  type Incomp (B : Boolean := True) is
    null record;
end Custom_Recs;
```
Listing 24: show_rec.adb

```ada
with Custom_Recs; use Custom_Recs;

procedure Show_Rec is
  R : Rec (new Incomp);
begin
  Show (R);
end Show_Rec;
```

In the Gen_Custom_Recs package, we're using type \( T \) (<>) — which can be any type — for the designated subtype of the access type \( T \_Access \), which is the type of \( T \_Rec \)'s discriminant. In the Custom_Recs package, we use the incomplete type Incomp to instantiate the generic package. Only after the instantiation, we declare the complete type.

Later on, we'll discuss discriminants again when we look into anonymous access discriminants (page 868), which provide some advantages in terms of accessibility rules (page 788).

Whole object assignments

As expected, we cannot change the discriminant value in whole object assignments. If we do that, the Constraint_Error exception is raised at runtime:

Listing 25: show_person.adb

```ada
with Persons; use Persons;

procedure Show_Person is
  S1 : String_Access := new String'("John");
  S2 : String_Access := new String'("Mark");
  P : Person := (Name => S1, Age => 30);
begin
  P := (Name => S1, Age => 31);
  -- ^^ OK: we didn't change the discriminant.
  Show (P);
  -- We can just repeat the discriminant:
  P := (Name => P.Name, Age => 32);
  -- ^^^^^^ OK: we didn't change the discriminant.
  Show (P);
  -- Of course, we can change the string itself:
  S1.all := "Mark";
  Show (P);
  P := (Name => S2, Age => 40);
  -- ^^ ERROR: we changed the discriminant!
  Show (P);
end Show_Person;
```
Learning Ada

Code block metadata

MD5: 96f4742365eb6a07c377a5dec28b5767

Runtime output

Name = John
Age = 31
Name = John
Age = 32
Name = Mark
Age = 32

raised CONSTRAINT_ERROR : show_person.adb:24 discriminant check failed

The first and the second assignments to P are OK because we didn’t change the discriminant. However, the last assignment raises the Constraint_Error exception at runtime because we’re changing the discriminant.

28.1.4 Parameters as Access Values

In addition to using discriminants as access values (page 745), we can use access types for subprogram formal parameters. For example, the N parameter of the Show procedure below has an access type:

Listing 26: names.ads

```ada
package Names is
  type Name is access String;
  procedure Show (N : Name);
end Names;
```

This is the complete code example:

Listing 27: names.ads

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Names is
```

Listing 28: names.adb

```ada
(continues on next page)
```
procedure Show (N : Name) is
begin
  Put_Line ("Name: " & N.all);
end Show;
end Names;

Listing 29: show_names.adb

with Names; use Names;

procedure Show_Names is
  N : Name := new String'("John");
begin
  Show (N);
end Show_Names;

Code block metadata
MD5: 526baf1996b4a2970c3fa2e3485dcbad

Runtime output
Name: John

Note that in this example, the Show procedure is basically just displaying the string. Since
the procedure isn't doing anything that justifies the need for an access type, we could have
implemented it with a simpler type:

package Names is
  type Name is access String;
  procedure Show (N : String);
end Names;

Listing 30: names.ads

with Ada.Text_IO; use Ada.Text_IO;

package body Names is
  procedure Show (N : String) is
  begin
    Put_Line ("Name: " & N);
  end Show;
end Names;

Listing 31: names.adb

with Names; use Names;

procedure Show_Names is
(continues on next page)

28.1. Access Types
4 N : Name := new String'("John");
5 begin
6    Show (N.all);
7 end Show_Names;

Code block metadata
MD5: 097ec1ff781fda9deed1de23cae39ae5

Runtime output
Name: John

It's important to highlight the difference between passing an access value to a subprogram and passing an object by reference. In both versions of this code example, the compiler will make use of a reference for the actual parameter of the N parameter of the Show procedure. However, the difference between these two cases is that:

• N : Name is a reference to an object (because it's an access value) that is passed by value, and
• N : String is an object passed by reference.

Changing the referenced object

Since the Name type gives us access to an object in the Show procedure, we could actually change this object inside the procedure. To illustrate this, let's change the Show procedure to lower each character of the string before displaying it (and rename the procedure to Lower_And_Show):

```
package Names is
  type Name is access String;
  procedure Lower_And_Show (N : Name);
end Names;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Characters.Handling;
use Ada.Characters.Handling;
package body Names is
  procedure Lower_And_Show (N : Name) is
    begin
      for I in N'Range loop
        N (I) := To_Lower (N (I));
      end loop;
      Put_Line ("Name: " & N.all);
    end Lower_And_Show;
end Names;
```
Listing 35: show_changed_names.adb

```ada
with Names; use Names;

procedure Show_Changed_Names is
  N : Name := new String'("John");
begin
  Lower_And_Show (N);
end Show_Changed_Names;
```

**Code block metadata**

MD5: 063a50728f5e7ffa669db2c8fdd3d6f

**Runtime output**

Name: john

Notice that, again, we could have implemented the Lower_And_Show procedure without using an access type:

Listing 36: names.ads

```ada
package Names is
  type Name is access String;
  procedure Lower_And_Show (N : in out String);
end Names;
```

Listing 37: names.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Names is
  procedure Lower_And_Show (N : in out String) is
    begin
      for I in N'Range loop
        N (I) := To_Lower (N (I));
      end loop;
      Put_Line ("Name: " & N);
    end Lower_And_Show;
end Names;
```

Listing 38: show_changed_names.adb

```ada
with Names; use Names;

procedure Show_Changed_Names is
  N : Name := new String'("John");
begin
  Lower_And_Show (N.all);
end Show_Changed_Names;
```

28.1. Access Types
Replace the access value

Instead of changing the object in the Lower_And_Show procedure, we could replace the access value by another one — for example, by allocating a new string inside the procedure. In this case, we have to pass the access value by reference using the in out parameter mode:

```
package Names is
  type Name is access String;
  procedure Lower_And_Show (N : in out Name);
end Names;
```

Listing 39: names.ads

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Characters.Handling;
use Ada.Characters.Handling;

package body Names is
  procedure Lower_And_Show (N : in out Name) is
    begin
      N := new String'(To_Lower (N.all));
      Put_Line ("Name: " & N.all);
    end Lower_And_Show;
end Names;
```

Listing 40: names.adb

```
with Names; use Names;

procedure Show_Changed_Names is
  N : Name := new String'("John");
begin
  Lower_And_Show (N);
end Show_Changed_Names;
```

Listing 41: show_changed_names.adb
Runtime output

Name: john

Now, instead of changing the object referenced by N, we're actually replacing it with a new object that we allocate inside the Lower_And_Show procedure.

As expected, contrary to the previous examples, we cannot implement this code by relying on parameter modes to replace the object. In fact, we have to use access types for this kind of operations.

Note that this implementation creates a memory leak. In a proper implementation, we should make sure to deallocate the object (page 800), as explained later on.

Side-effects on designated objects

In previous code examples from this section, we've seen that passing a parameter by reference using the in or in out parameter modes is an alternative to using access values as parameters. Let's focus on the subprogram declarations of those code examples and their parameter modes:

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Parameter type</th>
<th>Parameter mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show</td>
<td>Name</td>
<td>in</td>
</tr>
<tr>
<td>Show</td>
<td>String</td>
<td>in</td>
</tr>
<tr>
<td>Lower_And_Show</td>
<td>Name</td>
<td>in</td>
</tr>
<tr>
<td>Lower_And_Show</td>
<td>String</td>
<td>in out</td>
</tr>
</tbody>
</table>

When we analyze the information from this table, we see that in the case of using strings with different parameter modes, we have a clear indication whether the subprogram might change the object or not. For example, we know that a call to Show (N : String) won't change the string object that we're passing as the actual parameter.

In the case of passing an access value, we cannot know whether the designated object is going to be altered by a call to the subprogram. In fact, in both Show and Lower_And_Show procedures, the parameter is the same: N : Name — in other words, the parameter mode is in in both cases. Here, there's no clear indication about the effects of a subprogram call on the designated object.

The simplest way to ensure that the object isn't changed in the subprogram is by using access-to-constant types (page 780), which we discuss later on. In this case, we're basically saying that the object we're accessing in Show is constant, so we cannot possibly change it:

```
package Names is

   type Name is access String;
   type Constant_Name is access constant String;

   procedure Show (N : Constant_Name);

end Names;
```

28.1. Access Types
Listing 43: names.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
-- with Ada.Characters.Handling;
-- use Ada.Characters.Handling;

package body Names is

procedure Show (N : Constant_Name) is
begin
-- for I in N'Range loop
-- N (I) := To_Lower (N (I));
-- end loop;
Put_Line ("Name: " & N.all);
end Show;
end Names;
```

Listing 44: show_names.adb

```ada
with Names; use Names;

procedure Show_Names is
N : Name := new String'("John");
begin
Show (Constant_Name (N));
end Show_Names;
```

Code block metadata

MD5: 77526e0a159bf1bcbe58a21be250f3c

Runtime output

Name: John

In this case, the Constant_Name type ensures that the N parameter won't be changed in the Show procedure. Note that we need to convert from Name to Constant_Name to be able to call the Show procedure (in the Show_Names procedure). Although using in String is still a simpler solution, this approach works fine.

(Feel free to uncomment the call to To_Lower in the Show procedure and the corresponding with- and use-clauses to see that the compilation fails when trying to change the constant object.)

We could also mitigate the problem by using contracts. For example:

Listing 45: names.ads

```ada
package Names is

type Name is access String;

procedure Show (N : Name)
with Post => N.all'Old = N.all;
-- ^^^^^^^^^^^^^^^^^
-- we promise that we won't change
-- the object
end Names;
```
Learning Ada

Listing 46: names.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
-- with Ada.Characters.Handling;
-- use Ada.Characters.Handling;

package body Names is

  procedure Show (N : Name) is
    begin
      -- for I in N'Range loop
      --  N (I) := To_Lower (N (I));
      --  end loop;
      Put_Line ("Name: " & N.all);
    end Show;

end Names;
```

Listing 47: show_names.adb

```ada
with Names; use Names;

procedure Show_Names is
  N : Name := new String'("John");
  begin
    Show (N);
  end Show_Names;
```

Code block metadata

MD5: 2a70993232baca9d58d36e537a6fd32b

Runtime output

Name: John

Although a bit more verbose than a simple `in String`, the information in the specification of Show at least gives us an indication that the object won’t be affected by the call to this subprogram. Note that this code actually compiles if we try to modify `N.all` in the Show procedure, but the post-condition fails at runtime when we do that.

(By uncommentating and building the code again, you'll see an exception being raised at runtime when trying to change the object.)

In the postcondition above, we're using `'Old` to refer to the original object before the subprogram call. Unfortunately, we cannot use this attribute when dealing with `limited private types` (page 928) — or limited types in general. For example, let's change the declaration of Name and have it as a limited private type instead:

Listing 48: names.ads

```ada
package Names is

  type Name is limited private;

  function Init (S : String) return Name;
  function Equal (N1, N2 : Name)
```

(continues on next page)
In this case, we have no means to indicate that a call to Show won't change the internal state of the actual parameter.
For further reading...

As an alternative, we could declare a new `Constant_Name` type that is also limited private. If we use this type in `Show` procedure, we're at least indicating (in the type name) that the type is supposed to be constant — even though we're not directly providing means to actually ensure that no modifications occur in a call to the procedure. However, the fact that we declare this type as an access-to-constant (in the private part of the specification) makes it clear that a call to `Show` won't change the designated object.

Let's look at the adapted code:

```
package Names is

  type Name is limited private;
  type Constant_Name is limited private;
  function Init (S : String) return Name;
  function To_Constant_Name (N : Name) return Constant_Name;
  procedure Show (N : Constant_Name);

private

  type Name is access String;
  type Constant_Name is access constant String;

  function Init (S : String) return Name is (new String'(S));
  function To_Constant_Name (N : Name) return Constant_Name is (Constant_Name (N));

end Names;
```

```
with Ada.Text_IO; use Ada.Text_IO;
-- with Ada.Characters.Handling;
-- use Ada.Characters.Handling;

package body Names is

  procedure Show (N : Constant_Name) is
  begin
    -- for I in N'Range loop
    --   N (I) := To_Lower (N (I));
    -- end loop;
    Put_Line ("Name: " & N.all);
  end Show;
```

(continues on next page)
In this version of the source code, the Show procedure doesn't have any side-effects, as we cannot modify \textit{N} inside the procedure.

Having the information about the effects of a subprogram call to an object is very important: we can use this information to set expectations — and avoid unexpected changes to an object. Also, this information can be used to prove that a program works as expected. Therefore, whenever possible, we should avoid access values as parameters. Instead, we can rely on appropriate parameter modes and pass an object by reference.

There are cases, however, where the design of our application doesn't permit replacing the access type with simple parameter modes. Whenever we have an abstract data type encapsulated as a limited private type — such as in the last code example —, we might have no means to avoid access values as parameters. In this case, using the access type is of course justifiable. We'll see such a case in the next section (page 762).

### 28.1.5 Self-reference

As we've discussed in the section about incomplete types \(<\text{Adv}_\text{Ada}_\text{Incomplete}_\text{Types}>\), we can use incomplete types to create a recursive, self-referencing type. Let's revisit a code example from that section:

```
package Linked_List_Example is
  type Integer_List;
  type Next is access Integer_List;
  type Integer_List is record
    I : Integer;
    N : Next;
  end record;
end Linked_List_Example;
```
Here, we're using the incomplete type `Integer_List` in the declaration of the `Next` type, which we then use in the complete declaration of the `Integer_List` type.

Self-references are useful, for example, to create unbounded containers — such as the linked lists mentioned in the example above. Let's extend this code example and partially implement a generic package for linked lists:

**Listing 55: linked_lists.ads**

```ada
generic
  type T is private;
package Linked_Lists is
  type List is limited private;
  procedure Append_Front (L : in out List; E : T);
  procedure Append_Rear (L : in out List; E : T);
  procedure Show (L : List);
private
  -- Incomplete type declaration:
  type Component;
  -- Using incomplete type:
  type List is access Component;
  type Component is record
    Value : T;
    Next : List;
    -- ^^^^ Self-reference via access type
  end record;
end Linked_Lists;
```

**Listing 56: linked_lists.adb**

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;

package body Linked_Lists is
  procedure Append_Front (L : in out List; E : T) is
    New_First : constant List := new Component'(Value => E,
      (continues on next page)
null;  
begin  
if L = null then  
L := New_Last;  
else  
declare  
Last : List := L;  
begin  
while Last.Next /= null loop  
Last := Last.Next;  
end loop;  
Last.Next := New_Last;  
end;  
end if;  
end Append_Rear;  
end Show;  
end Linked_Lists;  
end Test_Linked_List;  
end with;
In this example, we declare an incomplete type Component in the private part of the generic Linked_Lists package. We use this incomplete type to declare the access type List, which is then used as a self-reference in the Next component of the Component type.

Note that we're using the List type as a parameter (page 752) for the Append_Front, Append_Rear and Show procedures.

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations

28.1.6 Mutually dependent types using access types

In the section on mutually dependent types (page 418), we've seen a code example where each type depends on the other one. We could rewrite that code example using access types:

Listing 58: mutually_dependent.ads

```ada
package Mutually_Dependent is
  type T2;
  type T2_Access is access T2;

  type T1 is record
    B : T2_Access;
  end record;

  type T1_Access is access T1;

  type T2 is record
    A : T1_Access;
  end record;
end Mutually_Dependent;
```

Code block metadata

174 http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html
In this example, T1 and T2 are mutually dependent types via the access types T1_Access and T2_Access — we're using those access types in the declaration of the B and A components.

### 28.1.7 Dereferencing

In the *Introduction to Ada course* (page 98), we discussed the `.all` syntax to dereference access values:

Listing 59: show_dereferencing.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Dereferencing is

-- Declaring access type:
type Integer_Access is access Integer;

-- Declaring access object:
A1 : Integer_Access;

begin
  A1 := new Integer;

  -- Dereferencing access value:
  A1.all := 22;

  Put_Line ("A1: " & Integer'image (A1.all));
end Show_Dereferencing;
```

#### Code block metadata


MD5: 65655768c17a02991ffeda9a853b6ff

#### Runtime output

A1: 22

In this example, we declare A1 as an access object, which allows us to access objects of `Integer` type. We dereference A1 by writing A1.all.

Here's another example, this time with an array:

Listing 60: show_dereferencing.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Dereferencing is

  type Integer_Array is array (Positive range <>) of Integer;

  type Integer_Array_Access is access Integer_Array;
```

(continues on next page)
In this example, we dereference the access value by writing Arr.all. We then assign an array aggregate to it — this becomes Arr.all := (1, 2, 3, 5, 8, 13);. Similarly, in the loop, we write Arr.all (I) to access the I component of the array.

In the Ada Reference Manual

- 4.1 Names

Implicit Dereferencing

Implicit dereferencing allows us to omit the .all suffix without getting a compilation error. In this case, the compiler knows that the dereferenced object is implied, not the access value.

Ada supports implicit dereferencing in these use cases:

- when accessing components of a record or an array — including array slices.
- when accessing subprograms that have at least one parameter (we discuss this topic later in this chapter);
- when accessing some attributes — such as some array and task attributes.
Arrays

Let's start by looking into an example of implicit dereferencing of arrays. We can take the previous code example and replace `Arr.all (I)` by `Arr (I)`: 

Listing 61: show_dereferencing.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Dereferencing is

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Integer_Array_Access is
    access Integer_Array;

  Arr : constant Integer_Array_Access :=
    new Integer_Array (1 .. 6);

begin
  Arr.all := (1, 2, 3, 5, 8, 13);
  Arr (1 .. 6) := (1, 2, 3, 5, 8, 13);

  for I in Arr'Range loop
    Put_Line
      ("Arr (: " & Integer'Image (I) & ") : " & Integer'Image (Arr (I));
       ^ .all is implicit.
  end loop;
end Show_Dereferencing;
```

Code block metadata

MD5: ade602a9e6976018e0c80f930a2399f1

Runtime output

Arr (: 1): 1
Arr (: 2): 2
Arr (: 3): 3
Arr (: 4): 5
Arr (: 5): 8
Arr (: 6): 13

Both forms — `Arr.all (I)` and `Arr (I)` — are equivalent. Note, however, that there's no implicit dereferencing when we want to access the whole array. (Therefore, we cannot write `Arr := (1, 2, 3, 5, 8, 13);`) However, as slices are implicitly dereferenced, we can write `Arr (1 .. 6) := (1, 2, 3, 5, 8, 13);` instead of `Arr.all (1 .. 6) := (1, 2, 3, 5, 8, 13);` Alternatively, we can assign to the array components individually and use implicit dereferencing for each component:

Arr (1) := 1;
Arr (2) := 2;
Arr (3) := 3;
Arr (4) := 5;
Arr (5) := 8;
Arr (6) := 13;
Implicit dereferencing isn't available for the whole array because we have to distinguish between assigning to access objects and assigning to actual arrays. For example:

Listing 62: show_array_assignments.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Array_Assignments is

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Integer_Array_Access is
    access Integer_Array;

  procedure Show_Array
    (Name : String;
     Arr : Integer_Array_Access) is
  begin
    Put (Name);
    for E of Arr.all loop
      Put (Integer'Image (E));
    end loop;
    New_Line;
  end Show_Array;

  Arr_1 : constant Integer_Array_Access :=
    new Integer_Array (1 .. 6);
  Arr_2 : Integer_Array_Access :=
    new Integer_Array (1 .. 6);

  begin
    Arr_1.all := (1, 2, 3, 5, 8, 13);
    Arr_2.all := (21, 34, 55, 89, 144, 233);
    -- Array assignment
    Arr_2.all := Arr_1.all;
    Show_Array ("Arr_2", Arr_2);
    -- Access value assignment
    Arr_2 := Arr_1;
    Arr_1.all := (377, 610, 987, 1597, 2584, 4181);
    Show_Array ("Arr_2", Arr_2);
  end Show_Array_Assignments;
```

Code block metadata

- **MD5**: 9b1f99af081000c28a6bf9b033127ea3

Runtime output

```
Arr_2 1 2 3 5 8 13
Arr_2 377 610 987 1597 2584 4181
```

Here, `Arr_2.all := Arr_1.all` is an array assignment, while `Arr_2 := Arr_1` is an access value assignment. By forcing the usage of the `.all` suffix, the distinction is clear. Implicit dereferencing, however, could be confusing here. (For example, the `.all` suffix in `Arr_2 := Arr_1.all` is an oversight by the programmer when the intention actually was to use access values on both sides.) Therefore, implicit dereferencing is only supported in those
Learning Ada

cases where there's no risk of ambiguities or oversights.

Records

Let's see an example of implicit dereferencing of a record:

Listing 63: show_dereferencing.adb

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Dereferencing is
  type Rec is record
    I : Integer;
    F : Float;
  end record;
  type Rec_Access is access Rec;
  R : constant Rec_Access := new Rec;
begin
  R.all := (I => 1, F => 5.0);
  Put_Line ("R.I: " & Integer'Image (R.I));
  Put_Line ("R.F: " & Float'Image (R.F));
end Show_Dereferencing;
```

Code block metadata

MD5: 9af72502d04f128785f77dcc829d5d48

Runtime output

R.I: 1
R.F: 5.00000E+00

Again, we can replace R.all.I by R.I, as record components are implicitly dereferenced. Also, we could use implicit dereference when assigning to record components individually:

R.I := 1;
R.F := 5.0;

However, we have to write R.all when assigning to the whole record R.

Attributes

Finally, let's see an example of implicit dereference when using attributes:

Listing 64: show_dereferencing.adb

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Dereferencing is
  type Integer_Array is
```

(continues on next page)
array (Positive range <>) of Integer;

type Integer_Array_Access is access Integer_Array;

Arr : constant Integer_Array_Access :=
    new Integer_Array (1 .. 6);

begin
    Put_Line ("Arr'First: 
        & Integer'Image (Arr'First));
    Put_Line ("Arr'Last: 
        & Integer'Image (Arr'Last));
    Put_Line ("Arr'Component_Size: 
        & Integer'Image (Arr'Component_Size));
    Put_Line ("Arr.all'Component_Size: 
        & Integer'Image (Arr.all'Component_Size));
    Put_Line ("Arr'Size: 
        & Integer'Image (Arr'Size));
    Put_Line ("Arr.all'Size: 
        & Integer'Image (Arr.all'Size));
end Show_Dereferencing;

Here, we can write Arr'First and Arr'Last instead of Arr.all'First and Arr.
all'Last, respectively, because Arr is implicitly dereferenced. The same applies to
Arr'Component_Size. Note that we can write both Arr'Size and Arr.all'Size, but they
have different meanings:

- Arr'Size is the size of the access object; while
- Arr.all'Size indicates the size of the actual array Arr.

In other words, the Size attribute is not implicitly dereferenced. In fact, any attribute that
could potentially be ambiguous is not implicitly dereferenced. Therefore, in those cases,
we must explicitly indicate (by using .all or not) how we want to use the attribute.
### Summary

The following table summarizes all instances where implicit dereferencing is supported:

<table>
<thead>
<tr>
<th>Entities</th>
<th>Standard Usage</th>
<th>Implicit Dereference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array components</td>
<td>Arr.all (I)</td>
<td>Arr (I)</td>
</tr>
<tr>
<td>Array slices</td>
<td>Arr.all (F .. L)</td>
<td>Arr (F .. L)</td>
</tr>
<tr>
<td>Record components</td>
<td>Rec.all.C</td>
<td>Rec.C</td>
</tr>
<tr>
<td>Array attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arr.all'First</td>
<td>Arr'First</td>
</tr>
<tr>
<td></td>
<td>Arr.all'First (N)</td>
<td>Arr'First (N)</td>
</tr>
<tr>
<td></td>
<td>Arr.all'Last</td>
<td>Arr'Last</td>
</tr>
<tr>
<td></td>
<td>Arr.all'Last (N)</td>
<td>Arr'Last (N)</td>
</tr>
<tr>
<td></td>
<td>Arr.all'Range</td>
<td>Arr'Range</td>
</tr>
<tr>
<td></td>
<td>Arr.all'Range (N)</td>
<td>Arr'Range (N)</td>
</tr>
<tr>
<td></td>
<td>Arr.all'Length</td>
<td>Arr'Length</td>
</tr>
<tr>
<td></td>
<td>Arr.all'Length (N)</td>
<td>Arr'Length (N)</td>
</tr>
<tr>
<td></td>
<td>Arr.all'Component_Size</td>
<td>Arr'Component_Size</td>
</tr>
<tr>
<td>Task attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T.all'Identity</td>
<td>T'Identity</td>
</tr>
<tr>
<td></td>
<td>T.all'Storage_Size</td>
<td>T'Storage_Size</td>
</tr>
<tr>
<td></td>
<td>T.all'Terminated</td>
<td>T'Terminated</td>
</tr>
<tr>
<td></td>
<td>T.all'Callable</td>
<td>T'Callable</td>
</tr>
<tr>
<td>Tagged type attributes</td>
<td>X.all'Tag</td>
<td>X'Tag</td>
</tr>
<tr>
<td>Other attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X.all'Valid</td>
<td>X'Valid</td>
</tr>
<tr>
<td></td>
<td>X.all'Old</td>
<td>X'Old</td>
</tr>
<tr>
<td></td>
<td>A.all'Constrained</td>
<td>A'Constrained</td>
</tr>
</tbody>
</table>

### In the Ada Reference Manual

- 4.1 Names[^176]
- 4.1.1 Indexed Components[^177]
- 4.1.2 Slices[^178]
- 4.1.3 Selected Components[^179]
- 4.1.4 Attributes[^180]

### 28.1.8 Ragged arrays

Ragged arrays — also known as jagged arrays — are non-uniform, multidimensional arrays. They can be useful to implement tables with varying number of coefficients, as we discuss as an example in this section.

[^177]: http://www.ada-auth.org/standards/22rm/html/RM-4-1-1.html
Uniform multidimensional arrays

Consider an algorithm that processes data based on coefficients that depend on a selected quality level:

<table>
<thead>
<tr>
<th>Quality level</th>
<th>Number of coefficients</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplified</td>
<td>1</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better</td>
<td>3</td>
<td>0.02</td>
<td>0.16</td>
<td>0.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td>5</td>
<td>0.01</td>
<td>0.08</td>
<td>0.12</td>
<td>0.20</td>
<td>0.34</td>
</tr>
</tbody>
</table>

(Note that this is just a bogus table with no real purpose, as we’re not trying to implement any actual algorithm.)

We can implement this table as a two-dimensional array (Calc_Table), where each quality level has an associated array:

```
Listing 65: data_processing.ads

package Data_Processing is

  type Quality_Level is
    (Simplified, Better, Best);

private

  Calc_Table : constant array
    (Quality_Level, 1 .. 5) of Float :=
    (Simplified =>
      (0.15, 0.00, 0.00, 0.00, 0.00),
    Better =>
      (0.02, 0.16, 0.27, 0.00, 0.00),
    Best =>
      (0.01, 0.08, 0.12, 0.20, 0.34));

  Last : constant array
    (Quality_Level) of Positive :=
    (Simplified => 1,
    Better => 3,
    Best => 5);

end Data_Processing;
```

Code block metadata

MD5: befa8d2b684ee20495f2dd6907dc44d4

Note that, in this implementation, we have a separate table Last that indicates the actual number of coefficients of each quality level.

Alternatively, we could use a record (Table_Coefficient) that stores the number of coefficients and the actual coefficients:

```
Listing 66: data_processing.ads

package Data_Processing is

  type Quality_Level is
    (Simplified, Better, Best);

  record Table_Coefficient is
    .number_of_coefficients : Positive := 0;
    .coefficients : array (1..5) of Float;
  end record;

private

end Data_Processing;
```
type Data is
     array (Positive range <>) of Float;
private
  type Table_Coefficient is record
     Last : Positive;
     Coef : Data (1 .. 5);
  end record;
Calc Table : constant array
     (Quality_Level) of Table_Coefficient :=
     (Simplified =>
      (1, (0.15, 0.00, 0.00, 0.00, 0.00)),
      Better =>
      (3, (0.02, 0.16, 0.27, 0.00, 0.00)),
      Best =>
      (5, (0.01, 0.08, 0.12, 0.20, 0.34)));
end Data_Processing;

In this case, we have a unidimensional array where each component (of Table_Coefficient type) contains an array (Coef) with the coefficients.
This is an example of a Process procedure that references the Calc_Table:

package Data_Processing.Operations is
  procedure Process (D : in out Data;
                     Q : Quality_Level);
end Data_Processing.Operations;

begin
  for I in D’Range loop
    for J in 1 .. Calc_Table (Q).Last loop
      -- ... * Calc_Table (Q).Coef (J)
      null;
    end loop;
    -- D (I) := ...
    null;
  end loop;
end Process;
end Data_Processing.Operations;
Note that, to loop over the coefficients, we’re using `for J in 1 .. Calc_Table (Q)`.

Last loop instead of `for J in Calc_Table (Q)`’s Range loop. As we’re trying to make a non-uniform array fit in a uniform array, we cannot simply loop over all elements using the Range attribute, but must be careful to use the correct number of elements in the loop instead.

Also, note that Calc_Table has 15 coefficients in total. Out of those coefficients, 6 coefficients (or 40 percent of the table) aren’t being used. Naturally, this is wasted memory space. We can improve this by using ragged arrays.

### Non-uniform multidimensional array

Ragged arrays are declared by using an access type to an array. By doing that, each array can be declared with a different size, thereby creating a non-uniform multidimensional array.

For example, we can declare a constant array Table as a ragged array:

```ada
package Data_Processing is
type Integer_Array is
  array (Positive range <>) of Integer;
private
  type Integer_Array_Access is
    access constant Integer_Array;
Table : constant array (1 .. 3) of
  Integer_Array_Access :=
  (1 => new Integer_Array' (1 => 15),
   2 => new Integer_Array' (1 => 12,
                            2 => 15,
                            3 => 20),
   3 => new Integer_Array' (1 => 12,
                            2 => 15,
                            3 => 20,
                            4 => 20,
                            5 => 25,
                            6 => 30));
end Data_Processing;
```

Here, each component of Table is an access to another array. As each array is allocated via `new`, those arrays may have different sizes.

We can rewrite the example from the previous subsection using a ragged array for the Calc_Table:
package Data_Processing is

   type Quality_Level is
      (Simplified, Better, Best);

   type Data is
      array (Positive range <>) of Float;

private

   type Coefficients is access constant Data;

   Calc_Table : constant array (Quality_Level) of
                  Coefficients :=
                  (Simplified =>
                    new Data'((1 => 0.15),
                               Better =>
                                 new Data'(0.02, 0.16, 0.27),
                               Best =>
                                 new Data'(0.01, 0.08, 0.12,
                                             0.20, 0.34));

end Data_Processing;

Code block metadata

.Ragged_Table
MD5: 0781b27cb2a27dbd1e74da54e425a1f4b

Now, we aren't wasting memory space because each data component has the right size that is required for each quality level. Also, we don't need to store the number of coefficients, as this information is automatically available from the array initialization — via the allocation of the Data array for the Coefficients type.

Note that the Coefficients type is defined as access constant. We discuss access-to-constant types (page 780) in more details later on.

This is the adapted Process procedure:

package Data_Processing.Operations is

   procedure Process (D : in out Data;
                      Q : Quality_Level);

end Data_Processing.Operations;

package body Data_Processing.Operations is

   procedure Process (D : in out Data;
                      Q : Quality_Level) is

      begin
        for I in D'Range loop
          for J in Calc_Table (Q)'Range loop
            null;
          end loop;
        end loop;
      end Process;

(continues on next page)
Now, we can simply loop over the coefficients by writing `for J in Calc_Table (Q)’Range loop`, as each element of Calc_Table automatically has the correct range.

### 28.1.9 Aliasing

The term aliasing\(^ {181}\) refers to objects in memory that we can access using more than a single reference. In Ada, if we allocate an object via `new`, we have a potentially aliased object. We can then have multiple references to this object:

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Show_Aliasing is
  type Integer_Access is access Integer;
  A1, A2 : Integer_Access;
begin
  A1 := new Integer;
  A2 := A1;
  A1.all := 22;
  Put_Line ("A1: " & Integer'Image (A1.all));
  A2.all := 24;
  Put_Line ("A2: " & Integer'Image (A2.all));
end Show_Aliasing;
```

**Code block metadata**

MD5: 2fde6073cec9823a1a9d93aecd82384e1

**Runtime output**

A1: 22
A2: 22
A1: 24
A2: 24

In this example, we access the object allocated via `new` by using either A1 or A2, as both refer to the same *aliased* object. In other words, A1 or A2 allow us to access the same object in memory.

**Important**

Learning Ada

Note that aliasing is unrelated to renaming. For example, we could use renaming to write a program that looks similar to the one above:

Listing 74: show_renaming.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Renaming is
   A1 : Integer;
   A2 : Integer renames A1;
begin
   A1 := 22;
   Put_Line ("A1: " & Integer'Image (A1));
   Put_Line ("A2: " & Integer'Image (A2));
   A2 := 24;
   Put_Line ("A1: " & Integer'Image (A1));
   Put_Line ("A2: " & Integer'Image (A2));
end Show_Renaming;
```

Here, A1 or A2 are two different names for the same object. However, the object itself isn't aliased.

In the Ada Reference Manual

- 3.10 Access Types\(^{182}\)

Aliased objects

As we discussed *previously* (page 737), we use new to create aliased objects on the heap. We can also use general access types to access objects that were created on the stack. By default, objects created on the stack aren't aliased. Therefore, we have to indicate that an object is aliased by using the aliased keyword in the object's declaration: Obj : aliased Integer;

Let's see an example:

Listing 75: show_aliased_obj.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Aliased_Obj is
   type Integer_Access is access all Integer;

   (continues on next page)
```

---

Here, we declare I_Var as an aliased integer variable and get a reference to it, which we assign to A1. Naturally, we could also have two accesses A1 and A2:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Aliased_Obj is
  type Integer_Access is access all Integer;
  I_Var : aliased Integer;
  A1, A2 : Integer_Access;
begin
  A1 := I_Var'Access;
  A2 := A1;
  A1.all := 22;
  Put_Line ("A1: " & Integer'Image (A1.all));
  A2.all := 24;
  Put_Line ("A2: " & Integer'Image (A2.all));
end Show_Aliased_Obj;
```

In this example, both A1 and A2 refer to the I_Var variable. Note that these examples make use of these two features:
Learning Ada

1. The declaration of a general access type (Integer_Access) using access all.
2. The retrieval of a reference to I_Var using the Access attribute.

In the next sections, we discuss these features in more details.

In the Ada Reference Manual

- 3.3.1 Object Declarations
- 3.10 Access Types

General access modifiers

Let’s now discuss how to declare general access types. In addition to the standard (pool-specific) access type declarations, Ada provides two access modifiers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access-to-variable</td>
<td>type T_Acc is access all T</td>
</tr>
<tr>
<td>Access-to-constant</td>
<td>type T_Acc is access constant T</td>
</tr>
</tbody>
</table>

Let’s look at an example:

Listing 77: integer_access_types.ads

```ada
package Integer_Access_Types is

  type Integer_Access is
      access Integer;

  type Integer_Access_All is
      access all Integer;

  type Integer_Access_Const is
      access constant Integer;

end Integer_Access_Types;
```

As we’ve seen previously, we can use a type such as Integer_Access to allocate objects dynamically. However, we cannot use this type to refer to declared objects, for example. In this case, we have to use an access-to-variable type such as Integer_Access_All. Also, if we want to access constants — or access objects that we want to treat as constants — we use a type such as Integer_Access_Const.

183 http://www.ada-auth.org/standards/22rm/html/RM-3-3-1.html
184 http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
**Access attribute**

To get access to a variable or a constant, we make use of the `Access` attribute. For example, `I_Var'Access` gives us access to the `I_Var` object.

Let's look at an example of how to use the integer access types from the previous code snippet:

Listing 78: integer_access_types.ads

```ada
package Integer_Access_Types is

    type Integer_Access is
        access Integer;

    type Integer_Access_All is
        access all Integer;

    type Integer_Access_Const is
        access constant Integer;

    procedure Show;

end Integer_Access_Types;
```

Listing 79: integer_access_types.adb

```ada
with Ada.Text_IO;
use Ada.Text_IO;

package body Integer_Access_Types is

    I_Var : aliased Integer := 0;
    Fact : aliased constant Integer := 42;
    Dyn_Ptr : constant Integer_Access := new Integer'(30);
    I_Var_Ptr : constant Integer_Access_All := I_Var'Access;
    I_Var_C_Ptr : constant Integer_Access_Const := I_Var'Access;
    Fact_Ptr : constant Integer_Access_Const := Fact'Access;

    procedure Show is
    begin
        Put_Line ("Dyn_Ptr: " & Integer'Image (Dyn_Ptr.all));
        Put_Line ("I_Var_Ptr: " & Integer'Image (I_Var_Ptr.all));
        Put_Line ("I_Var_C_Ptr: " & Integer'Image (I_Var_C_Ptr.all));
        Put_Line ("Fact_Ptr: " & Integer'Image (Fact_Ptr.all));
    end Show;

end Integer_Access_Types;
```

Listing 80: show_access_modifiers.adb

```ada
with Integer_Access_Types;
(continues on next page)
procedure Show_Access_Modifiers is
begin
  Integer_Access_Types.Show;
end Show_Access_Modifiers;

In this example, Dyn_Ptr refers to a dynamically allocated object, I_Var_Ptr refers to the I_Var variable, and Fact_Ptr refers to the Fact constant. We get access to the variable and the constant objects by using the Access attribute.

Also, we declare I_Var_C_Ptr as an access-to-constant, but we get access to the I_Var variable. This simply means the object I_Var_C_Ptr refers to is treated as a constant. Therefore, we can write I_Var := 22;, but we cannot write I_Var_C_Ptr.all := 22.;

In the Ada Reference Manual

• 3.10.2 Operations of Access Types

Non-aliased objects

As mentioned earlier, by default, declared objects — which are allocated on the stack — aren’t aliased. Therefore, we cannot get a reference to those objects. For example:

Listing 81: show_access_error.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Access_Error is
  type Integer_Access is access all Integer;
  I_Var : Integer;
  A1 : Integer_Access;
begin
  A1 := I_Var'Access;
  A1.all := 22;
  Put_Line ("A1: " & Integer'Image (A1.all));
end Show_Access_Error;
Build output

```
show_access_error.adb:8:10: error: prefix of "Access" attribute must be aliased

gprbuild: *** compilation phase failed
```

In this example, the compiler complains that we cannot get a reference to I_Var because I_Var is not aliased.

Ragged arrays using aliased objects

We can use aliased objects to declare **ragged arrays** (page 772). For example, we can rewrite a previous program using aliased constant objects:

```
package Data_Processing is

  type Integer_Array is
      array (Positive range <>) of Integer;

private

  type Integer_Array_Access is
      access constant Integer_Array;

  Tab_1 : aliased constant Integer_Array := (1 => 15);
  Tab_2 : aliased constant Integer_Array := (12, 15, 20);
  Tab_3 : aliased constant Integer_Array := (12, 15, 20,
      20, 25, 30);

  Table : constant array (1 .. 3) of
      Integer_Array_Access :=
          (1 => Tab_1'Access,
           2 => Tab_2'Access,
           3 => Tab_3'Access);

end Data_Processing;
```

Here, instead of allocating the constant arrays dynamically via **new**, we declare three aliased arrays (Tab_1, Tab_2 and Tab_3) and get a reference to them in the declaration of Table.
Aliased access objects

It's interesting to mention that access objects can be aliased themselves. Consider this example where we declare the Integer_Access_Access type to refer to an access object:

Listing 83: show.aliased_access_obj.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Aliased_Access_Obj is
  type Integer_Access is access all Integer;
  type Integer_Access_Access is access all Integer_Access;

  I_Var : aliased Integer;
  A : aliased Integer_Access;
  B : Integer_Access_Access;

begin
  A := I_Var'Access;
  B := A'Access;
  B.all.all := 22;
  Put_Line ("A: " & Integer'Image (A.all));
  Put_Line ("B: " & Integer'Image (B.all.all));
end Show_Aliased_Access_Obj;
```

After the assignments in this example, B refers to A, which in turn refers to I_Var. Note that this code only compiles because we declare A as an aliased (access) object.

Aliased components

Components of an array or a record can be aliased. This allows us to get access to those components:

Listing 84: show.aliased_components.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Aliased_Components is
  type Integer_Access is access all Integer;
  type Rec is record
    I_Var_1 : Integer;
    I_Var_2 : aliased Integer;
  end record;
```

After the assignments in this example, B refers to A, which in turn refers to I_Var. Note that this code only compiles because we declare A as an aliased (access) object.
In this example, we get access to the \texttt{I\_Var\_2} component of record \texttt{R}. (Note that trying to access the \texttt{I\_Var\_1} component would give us a compilation error, as this component is not aliased.) Similarly, we get access to the second component of array \texttt{Arr}.

Declaring components with the \texttt{aliased} keyword allows us to specify that those are accessible via other paths besides the component name. Therefore, the compiler won't store them in registers. This can be essential when doing low-level programming — for example, when accessing memory-mapped registers. In this case, we want to ensure that the compiler uses the memory address we're specifying (instead of assigning registers for those components).

In the Ada Reference Manual

\begin{itemize}
\item 3.6 Array Types\footnote{http://www.ada-auth.org/standards/22rm/html/RM-3-6.html}
\end{itemize}

\section*{Aliased parameters}

In addition to aliased objects and components, we can declare \texttt{aliased parameters} (page 625), as we already discussed in an earlier chapter. As we mentioned there, aliased parameters are always passed by reference, independently of the type we're using.

The parameter mode indicates which type we must use for the access type:
Using aliased parameters in a subprogram allows us to get access to those parameters in the body of that subprogram. Let’s see an example:

Listing 85: data_processing.ads

```ada
package Data_Processing is
  procedure Proc (I : aliased in out Integer);
end Data_Processing;
```

Listing 86: data_processing.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Data_Processing is

  procedure Show (I : aliased Integer) is
    -- ^ equivalent to
    -- "aliased in Integer"
    type Integer_Constant_Access is
      access constant Integer;
    A : constant Integer_Constant_Access := I'Access;
    begin
      Put_Line ("Value : I ": Integer'Image (A.all));
    end Show;

  procedure Set_One (I : aliased out Integer) is
    type Integer_Access is access all Integer;
    procedure Local_Set_One (A : Integer_Access) is
      begin
        A.all := 1;
      end Local_Set_One;
    begin
      Local_Set_One (I'Access);
    end Set_One;

  procedure Proc (I : aliased in out Integer) is
    type Integer_Access is access all Integer;
    procedure Add_One (A : Integer_Access) is
      begin
        A.all := A.all + 1;
      end Add_One;

end Data_Processing;
```
begin
  Show (I);
  Add_One (I'Access);
  Show (I);
end Proc;
end Data_Processing;

Listing 87: show_aliased_param.adb

```ada
with Data_Processing; use Data_Processing;

procedure Show_Aliased_Param is
  I : aliased Integer := 22;
begin
  Proc (I);
end Show_Aliased_Param;
```

Here, Proc has an aliased in out parameter. In Proc's body, we declare the Integer_Access type as an access all type. We use the same approach in body of the Set_One procedure, which has an aliased out parameter. Finally, the Show procedure has an aliased in parameter. Therefore, we declare the Integer_Constant_Access as an access constant type.

Note that parameter aliasing has an influence on how arguments are passed to a subprogram when the parameter is of scalar type. When a scalar parameter is declared as aliased, the corresponding argument is passed by reference. For example, if we had declared procedure Show (I : Integer), the argument for I would be passed by value. However, since we're declaring it as aliased Integer, it is passed by reference.

In the Ada Reference Manual

- 6.1 Subprogram Declarations\(^{187}\)
- 6.2 Formal Parameter Modes\(^{188}\)
- 6.4.1 Parameter Associations\(^{189}\)

\(^{187}\) http://www.ada-auth.org/standards/22rm/html/RM-6-1.html
\(^{188}\) http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
\(^{189}\) http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
28.1.10 Accessibility Levels and Rules: An Introduction

This section provides an introduction to accessibility levels and accessibility rules. This topic can be very complicated, and by no means do we intend to cover all the details here. (In fact, discussing all the details about accessibility levels and rules could be a long chapter on its own. If you're interested in them, please refer to the Ada Reference Manual.) In any case, the goal of this section is to present the intention behind the accessibility rules and build intuition on how to best use access types in your code.

In the Ada Reference Manual

- 3.10.2 Operations of Access Types

Lifetime of objects

First, let's talk a bit about lifetime of objects. We assume you understand the concept, so this section is very short.

In very simple terms, the lifetime of an object indicates when an object still has relevant information. For example, if a variable V gets out of scope, we say that its lifetime has ended. From this moment on, V no longer exists.

For example:

Listing 88: show_lifetime.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Lifetime is
  I_Var_1: Integer := 22;
begin
  Inner_Block:
    declare
      I_Var_2: Integer := 42;
    begin
      Put_Line ("I_Var_1: ",
                & Integer'Image (I_Var_1));
      Put_Line ("I_Var_2: ",
                & Integer'Image (I_Var_2));

      -- I_Var_2 will get out of scope
      -- when the block finishes.
    end Inner_Block;

    -- I_Var_2 is now out of scope...
    Put_Line ("I_Var_1: ",
              & Integer'Image (I_Var_1));
    Put_Line ("I_Var_2: ",
              & Integer'Image (I_Var_2));

    -- ERROR: lifetime of I_Var_2 has ended!
end Show_Lifetime;
```

Code block metadata

190 http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html
191 https://en.wikipedia.org/wiki/Variable_(computer_science)#Scope_and_extent
In this example, we declare I_Var_1 in the Show_Lifetime procedure, and I_Var_2 in its Inner_Block.

This example doesn't compile because we're trying to use I_Var_2 after its lifetime has ended. However, if such a code could compile and run, the last call to Put_Line would potentially display garbage to the user. (In fact, the actual behavior would be undefined.)

**Accessibility Levels**

In basic terms, accessibility levels are a mechanism to assess the lifetime of objects (as we've just discussed). The starting point is the library level: this is the base level, and no level can be deeper than that. We start "moving" to deeper levels when we use a library in a subprogram or call other subprograms for example.

Suppose we have a procedure Proc that makes use of a package Pkg, and there's a block in the Proc procedure:

```ada
package Pkg is
   -- Library level
end Pkg;

with Pkg; use Pkg;

procedure Proc is
   -- One level deeper than
   -- library level
begin
   declare
      -- Two levels deeper than
      -- library level
      begin
      null;
   end;
end Proc;
```

For this code, we can say that:

- the specification of Pkg is at library level;
- the declarative part of Proc is one level deeper than the library level; and
- the block is two levels deeper than the library level.

(Note that this is still a very simplified overview of accessibility levels. Things start getting more complicated when we use information from Pkg in Proc. Those details will become more clear in the next sections.)
The levels themselves are not visible to the programmer. For example, there's no Access_Level attribute that returns an integer value indicating the level. Also, you cannot write a user message that displays the level at a certain point. In this sense, accessibility levels are assessed relatively to each other: we can only say that a specific operation is at the same or at a deeper level than another one.

**Accessibility Rules**

The accessibility rules determine whether a specific use of access types or objects is legal (or not). Actually, accessibility rules exist to prevent dangling references (page 795), which we discuss later. Also, they are based on the accessibility levels (page 789) we discussed earlier.

**Code example**

As mentioned earlier, the accessibility level at a specific point isn't visible to the programmer. However, to illustrate which level we have at each point in the following code example, we use a prefix (L0, L1, and L2) to indicate whether we're at the library level (L0) or at a deeper level.

Let's now look at the complete code example:

---

**Listing 89: library_level.ads**

```ada
package Library_Level is

   type L0_Integer_Access is
      access all Integer;

   L0_IA : L0_Integer_Access;
   L0_Var : aliased Integer;

end Library_Level;
```

**Listing 90: show_library_level.adb**

```ada
with Library_Level; use Library_Level;

procedure Show_Library_Level is

   type L1_Integer_Access is
      access all Integer;

   L0_IA_2 : L0_Integer_Access;
   L1_IA : L1_Integer_Access;
   L1_Var : aliased Integer;

   procedure Test is

      type L2_Integer_Access is
         access all Integer;

      L2_IA : L2_Integer_Access;
      L2_Var : aliased Integer;

      begin
         L1_IA := L2_Var'Access;
         -- ^^^^^^^ (continues on next page)
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
-- ILLEGAL: L2 object to L1 access object

L2_IA := L2_Var'Access;
-- ^^^^^^^
-- LEGAL: L2 object to L2 access object

end Test;

begin
L0_IA := new Integer'(22);
-- ^^^^^^^^^^^
-- LEGAL: L0 object to L0 access object

L0_IA_2 := new Integer'(22);
-- ^^^^^^^^^^^
-- LEGAL: L0 object to L0 access object

L0_IA := L1_Var'Access;
-- ^^^^^
-- ILLEGAL: L1 object to L0 access object

L0_IA_2 := L1_Var'Access;
-- ^^^^^
-- ILLEGAL: L1 object to L0 access object

L1_IA := L0_Var'Access;
-- ^^^^^
-- LEGAL: L0 object to L1 access object

L1_IA := L1_Var'Access;
-- ^^^^^
-- LEGAL: L1 object to L1 access object

L0_IA := L1_IA;
-- ^^^^
-- ILLEGAL: type mismatch

L0_IA := L0_Integer_Access (L1_IA);
-- ^^^^^^^^^^^^^^^^^
-- ILLEGAL: cannot convert L1 access object to L0 access object

Test;

end Show_Library_Level;
```

**Code block metadata**


MD5: b3bed7e7eb2a8dfe78a2e7a7d2ce99f736

**Build output**

28.1. Access Types
show_library_level.adb:20:16: error: non-local pointer cannot point to local object
show_library_level.adb:42:13: error: non-local pointer cannot point to local object
show_library_level.adb:47:15: error: non-local pointer cannot point to local object
show_library_level.adb:62:13: error: expected type "L0_Integer_Access" defined at
    library_level.ads:3
show_library_level.adb:62:13: error: found type "L1_Integer_Access" defined at
    line 4
show_library_level.adb:66:32: error: cannot convert local pointer to non-local
    access type
gprbuild: *** compilation phase failed

In this example, we declare

- in the Library_Level package: the L0_Integer_Access type, the L0_IA access object, and the L0_Var aliased variable;
- in the Show_Library_Level procedure: the L1_Integer_Access type, the L0_IA_2 and L1_IA access objects, and the L1_Var aliased variable;
- in the nested Test procedure: the L2_Integer_Access type, the L2_IA, and the L2_Var aliased variable.

As mentioned earlier, the Ln prefix indicates the level of each type or object. Here, the n value is zero at library level. We then increment the n value each time we refer to a deeper level.

For instance:

- when we declare the L1_Integer_Access type in the Show_Library_Level procedure, that declaration is one level deeper than the level of the Library_Level package — so it has the L1 prefix.
- when we declare the L2_Integer_Access type in the Test procedure, that declaration is one level deeper than the level of the Show_Library_Level procedure — so it has the L2 prefix.

**Types and Accessibility Levels**

It's very important to highlight the fact that:

- types themselves also have an associated level, and
- objects have the same accessibility level as their types.

When we declare the L0_IA_2 object in the code example, its accessibility level is at library level because its type (the L0_Integer_Access type) is at library level. Even though this declaration is in the Show_Library_Level procedure — whose declarative part is one level deeper than the library level —, the object itself has the same accessibility level as its type.

Now that we’ve discussed the accessibility levels of this code example, let’s see how the accessibility rules use those levels.
Operations on Access Types

In very simple terms, the accessibility rules say that:

- operations on access types at the same accessibility level are legal;
- assigning or converting to a deeper level is legal;

Otherwise, operations targeting objects at a less-deep level are illegal.

For example, \( L_0 \_IA := \text{new Integer}'(22) \) and \( L_1 \_IA := L_1 \_Var'\text{Access} \) are legal because we're operating at the same accessibility level. Also, \( L_1 \_IA := L_0 \_Var'\text{Access} \) is legal because \( L_1 \_IA \) is at a deeper level than \( L_0 \_Var'\text{Access} \).

However, many operations in the code example are illegal. For instance, \( L_0 \_IA := L_1 \_Var'\text{Access} \) and \( L_0 \_IA_2 := L_1 \_Var'\text{Access} \) are illegal because the target objects in the assignment are less deep.

Note that the \( L_0 \_IA := L_1 \_IA \) assignment is mainly illegal because the access types don't match. (Of course, in addition to that, assigning \( L_1 \_Var'\text{Access} \) to \( L_0 \_IA \) is also illegal in terms of accessibility rules.)

Conversion between Access Types

The same rules apply to the conversion between access types. In the code example, the \( L_0 \_Integer'\text{Access} \) \( \rightarrow \) \( L_1 \_IA \) conversion is illegal because the resulting object is less deep. That being said, conversions on the same level are fine:

```
procedure Show_Same_Level_Conversion is
  type L1_Integer_Access is
    access all Integer;
  type L1_B_Integer_Access is
    access all Integer;

  L1_IA : L1_Integer_Access;
  L1_B_IA : L1_B_Integer_Access;
  L1_Var : aliased Integer;

begin
  L1_IA := L1_Var'Access;
  L1_B_IA := L1_B_Integer_Access (L1_IA);
  -- """"""""""""""""""""""""
  -- LEGAL: conversion from L1 access object to L1 access object
end Show_Same_Level_Conversion;
```

Here, we're converting from the \( L_1 \_Integer'\text{Access} \) type to the \( L_1 \_B'\text{Integer'Access} \), which are both at the same level.
**Accessibility rules on parameters**

Note that the accessibility rules also apply to access values as subprogram parameters. For example, compilation fails for this example:

```
package Names is
  type Name is access all String;
  type Constant_Name is
    access constant String;
  procedure Show (N : Constant_Name);
end Names;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Names is
  procedure Show (N : Constant_Name) is
  begin
    -- for I in N'Range loop
    --   N (I) := To_Lower (N (I));
    -- end loop;
    Put_Line ("Name: " & N.all);
  end Show;
end Names;
```

```
with Names; use Names;
procedure Show_Names is
  S : aliased String := "John";
begin
  Show (S'Access);
end Show_Names;
```

In this case, the `S'Access` cannot be used as the actual parameter for the `N` parameter of the `Show` procedure because it's in a deeper level. If we allocate the string via `new`, however, the code compiles as expected:
Listing 95: show_names.adb

```ada
with Names; use Names;

procedure Show_Names is
  S : Name := new String'("John");
begin
  Show (Constant_Name (S));
end Show_Names;
```

**Code block metadata**


MD5: 30237c83426db758804b802e1953d5d9

**Runtime output**

Name: John

This version of the code works because both object and access object have the same level.

**Dangling References**

An access value that points to a non-existent object is called a dangling reference. Later on, we'll discuss how dangling references may occur using *unchecked deallocation* (page 803).

Dangling references are created when we have an access value pointing to an object whose lifetime has ended, so it becomes a non-existent object. This could occur, for example, when an access value still points to an object X that has gone out of scope.

As mentioned in the previous section, the accessibility rules of the Ada language ensure that such situations never happen! In fact, whenever possible, the compiler applies those rules to detect potential dangling references at compile time. When this detection isn't possible at compile time, the compiler introduces an *accessibility check* (page 669). If this check fails at runtime, it raises a *Program_Error* exception — thereby preventing that a dangling reference gets used.

Let's see an example of how dangling references could occur:

Listing 96: show_dangling_reference.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Dangling_Reference is
  type Integer_Access is
    access all Integer;

  I_Var_1 : aliased Integer := 22;
  A1 : Integer_Access;

begin
  A1 := I_Var_1'Access;
  Put_Line ("A1.all: ", Integer'Image (A1.all));
  Put_Line ("Inner_Block will start now!");
  Inner_Block : declare
```
Here, we declare the access objects A1 and A2 of Integer_Access type, and the I_Var_1 and I_Var_2 objects. Moreover, A1 and I_Var_1 are declared in the scope of the Show_Dangling_Reference procedure, while A2 and I_Var_2 are declared in the Inner_Block.

When we try to compile this code, we get two compilation errors due to violation of accessibility rules. Let's now discuss these accessibility rules in terms of lifetime, and see which problems they are preventing in each case.

1. In the A1 := I_Var_2'Access assignment, the main problem is that A1 has a longer lifetime than I_Var_2. After the Inner_Block finishes — when I_Var_2 gets out of
Learning Ada

scope and its lifetime has ended —, A1 would still be pointing to an object that does not longer exist.

2. In the A2 := I_VAR_2'Access assignment, however, both A2 and I_VAR_2 have the same lifetime. In that sense, the assignment may actually look pretty much OK.

- However, as mentioned in the previous section, Ada also cares about the lifetime of access types. In fact, since the Integer_Access type is declared outside of the Inner_Block, it has a longer lifetime than A2 and I_VAR_2.

- To be more precise, the accessibility rules detect that A2 is an access object of a type that has a longer lifetime than I_VAR_2.

At first glance, this last accessibility rule may seem too strict, as both A2 and I_VAR_2 have the same lifetime — so nothing bad could occur when dereferencing A2. However, consider the following change to the code:

```ada
A2 := I_VAR_2'Access;
A1 := A2;
-- PROBLEM: A1 will still be referring
to I_VAR_2 after the
-- Inner_Block, i.e. when the
-- lifetime of I_VAR_2 has
-- ended!
```

Here, we're introducing the A1 := A2 assignment. The problem with this is that I_VAR_2's lifetime ends when the Inner_Block finishes, but A1 would continue to refer to an I_VAR_2 object that doesn't exist anymore — thereby creating a dangling reference.

Even though we're actually not assigning A2 to A1 in the original code, we could have done it. The accessibility rules ensure that such an error is never introduced into the program.

For further reading...

In the original code, we can consider the A2 := I_VAR_2'Access assignment to be safe, as we're not using the A1 := A2 assignment there. Since we're confident that no error could ever occur in the Inner_Block due to the assignment to A2, we could replace it with A2 := I_VAR_2'Unchecked_Access, so that the compiler accepts it. We discuss more about the unchecked access attribute later in this chapter (page 798).

Alternatively, we could have solved the compilation issue that we see in the A2 := I_VAR_2'Access assignment by declaring another access type locally in the Inner_Block:

```ada
Inner_Block : declare
  type Integer_Local_Access is
    access all Integer;

  I_VAR_2 : aliased Integer := 42;

  A2 : Integer_Local_Access;
begin
  A2 := I_VAR_2'Access;
  -- This assignment is fine because
  -- the Integer_Local_Access type has
  -- the same lifetime as I_VAR_2.
end Inner_Block;
```

With this change, A2 becomes an access object of a type that has the same lifetime as I_VAR_2, so that the assignment doesn't violate the rules anymore.

(Nota note that in the Inner_Block, we could have simply named the local access type Integer_Access instead of Integer_Local_Access, thereby masking the Integer_Access

28.1. Access Types
We discuss the effects of dereferencing dangling references later in this chapter (page 805).

28.1.11 Unchecked Access

In this section, we discuss the Unchecked_Access attribute, which we can use to circumvent accessibility issues for objects in specific cases. (Note that this attribute only exists for objects, not for subprograms.)

We’ve seen previously (page 788) that the accessibility levels verify the lifetime of access types. Let’s see a simplified version of a code example from that section:

Listing 97: integers.ads

```ada
package Integers is
  type Integer_Access is access all Integer;
end Integers;
```

Listing 98: show_access_issue.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Integers; use Integers;
procedure Show_Access_Issue is
  I_Var : aliased Integer := 42;
  A : Integer_Access;
begin
  A := I_Var'Access;
  -- PROBLEM: A has the same lifetime as I_Var, but Integer_Access type has a longer lifetime.
  Put_Line ("A.all: " & Integer'Image (A.all));
end Show_Access_Issue;
```

Here, the compiler complains about the A := I_Var'Access assignment because the Integer_Access type has a longer lifetime than A. However, we know that this assignment to A — and further uses of A in the code — won’t cause dangling references to be created. Therefore, we can assume that assigning the access to I_Var to A is safe.

When we’re sure that an access assignment cannot possibly generate dangling references, we can use the use Unchecked_Access attribute. For instance, we can use this attribute to circumvent the compilation error in the previous code example, since we know that the assignment is actually safe:
Learning Ada

Listing 99: integers.ads

```ada
package Integers is
    type Integer_Access is access all Integer;
end Integers;
```

Listing 100: show_access_issue.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Integers; use Integers;

procedure Show_Access_Issue is
    I_Var : aliased Integer := 42;
    A : Integer_Access;

begin
    A := I_Var'Unchecked_Access;
    -- OK: assignment is now accepted.
    Put_Line ("A.all: " & Integer'Image (A.all));
end Show_Access_Issue;
```

Code block metadata

Dangling_Reference_Rules
MD5: a71b9076d9e2903ff9811183afdf6c1

Runtime output

| A.all: 42 |

When we use the Unchecked_Access attribute, most rules still apply. The only difference to the standard Access attribute is that unchecked access applies the rules as if the object we're getting access to was being declared at library level. (For the code example we've just seen, the check would be performed as if I_Var was declared in the Integers package instead of being declared in the procedure.)

It is strongly recommended to avoid unchecked access in general. You should only use it when you can safely assume that the access object will be discarded before the object we had access to gets out of scope. Therefore, if this situation isn't clear enough, it's best to avoid unchecked access. (Later in this chapter, we'll see some of the nasty issues that arrive from creating dangling references.) Instead, you should work on improving the software design of your application by considering alternatives such as using containers or encapsulating access types in well-designed abstract data types.

In the Ada Reference Manual

- Unchecked Access Value Creation\(^{192}\)

---


---

28.1. Access Types
28.1.12 Unchecked Deallocation

So far, we’ve seen multiple examples of using `new` to allocate objects. In this section, we discuss how to manually deallocate objects.

Our starting point to manually deallocate an object is the generic Ada.Unchecked_Deallocation procedure. We first instantiate this procedure for an access type whose objects we want to be able to deallocate. For example, let’s instantiate it for the Integer_Access type:

```
Listing 101: integer_types.ads

with Ada.Unchecked_Deallocation;

package Integer_Types is

  type Integer_Access is access Integer;

  -- Instantiation of Ada.Unchecked_Deallocation
  -- for the Integer_Access type:
  --

  procedure Free is
    new Ada.Unchecked_Deallocation
      (Object => Integer,
       Name => Integer_Access);

end Integer_Types;
```

Here, we declare the `Free` procedure, which we can then use to deallocate objects that were allocated for the `Integer_Access` type.

Ada.Unchecked_Deallocation is a generic procedure that we can instantiate for access types. When declaring an instance of Ada.Unchecked_Deallocation, we have to specify arguments for:

- the formal `Object` parameter, which indicates the type of actual objects that we want to deallocate; and
- the formal `Name` parameter, which indicates the access type.

In a type declaration such as `type Integer_Access is access Integer, Integer` denotes the `Object`, while `Integer_Access` denotes the `Name`.

Because each instance of Ada.Unchecked_Deallocation is bound to a specific access type, we cannot use it for another access type, even if the type we use for the `Object` parameter is the same:

```
Listing 102: integer_types.ads

with Ada.Unchecked_Deallocation;

package Integer_Types is

  type Integer_Access is access Integer;

  procedure Free is
    new Ada.Unchecked_Deallocation

(continues on next page)
Here, we're declaring two Free procedures: one for the Integer_Access type, another for the Another_Integer_Access type. We cannot use the Free procedure for the Integer_Access type when deallocating objects associated with the Another_Integer_Access type, even though both types are declared as access Integer.

Note that we can use any name when instantiating the Ada.Unchecked_Deallocation procedure. However, naming it Free is very common.

Now, let's see a complete example that includes object allocation and deallocation:

Listing 103: integer_types.ads

```ada
with Ada.Unchecked_Deallocation;

package Integer_Types is
  type Integer_Access is access Integer;
  procedure Free is
    new Ada.Unchecked_Deallocation
      (Object => Integer,
       Name => Integer_Access);
  procedure Show_Is_Null (I : Integer_Access);
end Integer_Types;
```

Listing 104: integer_types.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Integer_Types is
  procedure Show_Is_Null (I : Integer_Access) is
    begin
      if I = null then
        Put_Line ("access value is null.");
      else
        Put_Line ("access value is NOT null.");
      end if;
    end Show_Is_Null;
end Integer_Types;
```
In the `Show_Unchecked_Deallocation` procedure, we first allocate an object for `I` and then call `Free (I)` to deallocate it. Also, we call the `Show_Is_Null` procedure at three different points: before any allocation takes place, after allocating an object for `I`, and after deallocating that object.

When we deallocate an object via a call to `Free`, the corresponding access value — which was previously pointing to an existing object — is set to `null`. Therefore, `I = null` after the call to `Free`, which is exactly what we see when running this example code.

Note that it is OK to call `Free` multiple times for the same access object:
The multiple calls to Free for the same access object don't cause any issues. Because the access value is null after the first call to Free (I), we're actually just passing null as an argument in the second and third calls to Free. However, any attempt to deallocate an access value of null is ignored in the Free procedure, so the second and third calls to Free don't have any effect.

In the Ada Reference Manual

- 4.8 Allocators
- 13.11.2 Unchecked Storage Deallocation

Unchecked Deallocation and Dangling References

We've discussed dangling references (page 795) before. In this section, we discuss how unchecked deallocation can create dangling references and the issues of having them in an application.

Let's reuse the last example and introduce I_2, which will point to the same object as I:

Listing 107: showUncheckedDeallocation.adb

```ada
with Integer_Types; use Integer_Types;
procedure Show_Unchecked_Deallocation is
  I, I_2 : Integer_Access;
begin
  I := new Integer;
  I_2 := I;
  -- NOTE: I_2 points to the same
  -- object as I.
  --
  -- Use I and I_2...
  --
  -- ... then deallocate memory...
  --
  Free (I);
  -- NOTE: at this point, I_2 is a
  -- dangling reference!
  --
  -- Further calls to Free (I)
  -- are OK!
  Free (I);
  Free (I);
```
As we've seen before, we can have multiple calls to `Free (I)`. However, the call to `Free (I_2)` is bad because `I_2` is not null. In fact, it is a dangling reference — i.e. `I_2` points to an object that doesn't exist anymore. Also, the first call to `Free (I)` will reclaim the storage that was allocated for the object that I originally referred to. The call to `Free (I_2)` will then try to reclaim the previously-reclaimed object, but it'll fail in an undefined manner.

Because of these potential errors, you should be very careful when using unchecked deallocation: it is the programmer's responsibility to avoid creating dangling references!

For the example we've just seen, we could avoid creating a dangling reference by explicitly assigning `null` to `I_2` to indicate that it doesn't point to any specific object:

```ada
with Integer_Types; use Integer_Types;

procedure Show_Unchecked_Deallocation is
begin
   I, I_2 : Integer_Access;
   I := new Integer;
   I_2 := I;
   -- NOTE: I_2 points to the same object as I.
   --
   -- Use I and I_2...
   --
   -- ... then deallocate memory...
   --
   I_2 := null;
   -- NOTE: now, I_2 doesn't point to any object, so calling Free (I_2) is OK.
   Free (I);
   Free (I_2);
end Show_Unchecked_Deallocation;
```
Now, calling Free \((I_2)\) doesn't cause any issues because it doesn't point to any object. Note, however, that this code example is just meant to illustrate the issues of dangling pointers and how we could circumvent them. We're not suggesting to use this approach when designing an implementation. In fact, it's not practical for the programmer to make every possible dangling reference become null if the calls to Free are strewn throughout the code.

The suggested design is to not use Free in the client code, but instead hide its use within bigger abstractions. In that way, all the occurrences of the calls to Free are in one package, and the programmer of that package can then prevent dangling references. We'll discuss these design strategies (page 812) later on.

**Dereferencing dangling references**

Of course, you shouldn't try to dereference a dangling reference because your program becomes erroneous, as we discuss in this section. Let's see an example:

```
with Ada.Text_IO;  use Ada.Text_IO;
with Integer_Types; use Integer_Types;

procedure Show_Unchecked_Deallocation is
  I_1, I_2 : Integer_Access;
begin
  I_1 := new Integer'(42);
  I_2 := I_1;
  Put_Line ("I_1.all = " & Integer'Image (I_1.all));
  Put_Line ("I_2.all = " & Integer'Image (I_2.all));
  Put_Line ("Freeing I_1");
  Free (I_1);
  if I_1 /= null then
    Put_Line ("I_1.all = " & Integer'Image (I_1.all));
  end if;
  if I_2 /= null then
    Put_Line ("I_2.all = " & Integer'Image (I_2.all));
  end if;
end Show_Unchecked_Deallocation;
```
In this example, we allocate an object for \( I_1 \) and make \( I_2 \) point to the same object. Then, we call \texttt{Free \ (I)}\textit{,} which has the following consequences:

- The call to \texttt{Free \ (I_1)}\textit{ will try to reclaim the storage for the original object (}\( I_1.all \))\textit{, so it may be reused for other allocations.}
- \( I_1 = \texttt{null} \) after the call to \texttt{Free \ (I_1)}\textit{.}
- \( I_2 \) becomes a dangling reference by the call to \texttt{Free \ (I_1)}\textit{.}
  - In other words, \( I_2 \) is still non-null, and what it points to is now undefined.

In principle, we could check for \texttt{null} before trying to dereference the access value. (Remember that when deallocating an object via a call to \texttt{Free}, the corresponding access value is set to \texttt{null}.) In fact, this strategy works fine for \( I_1 \), but it doesn't work for \( I_2 \) because the access value is not \texttt{null}. As a consequence, the application tries to dereference \( I_2 \).

Dereferencing a dangling reference is erroneous: the behavior is undefined in this case. For the example we've just seen,

- \( I_2.all \) might make the application crash;
- \( I_2.all \) might give us a different value than before;
- \( I_2.all \) might even give us the same value as before (42) if the original object is still available.

Because the effect is unpredictable, it might be really difficult to debug the application and identify the cause.

Having dangling pointers in an application should be avoided at all costs! Again, it is the programmer’s responsibility to be very careful when using unchecked deallocation: avoid creating dangling references!

\textbf{In the Ada Reference Manual}

- 13.9.1 Data Validity\textsuperscript{195}
- 13.11.2 Unchecked Storage Deallocation\textsuperscript{196}

\textbf{Restrictions for Ada.Unchecked_Deallocation}

There are two unsurprising restrictions for \texttt{Ada.Unchecked_Deallocation}:

1. It cannot be instantiated for access-to-constant types; and
2. It cannot be used when the Storage_Size aspect of a type is zero (i.e. when its storage pool is empty).

(Note that this last restriction also applies to the allocation via \texttt{new}.)

Let's see an example of these restrictions:

\textsuperscript{195} http://www.ada-auth.org/standards/22rm/html/RM-13-9-1.html
\textsuperscript{196} http://www.ada-auth.org/standards/22rm/html/RM-13-11-2.html
Listing 110: show_unchecked_deallocation_errors.adb

```ada
with Ada.Unchecked_Deallocation;

procedure Show_Unchecked_Deallocation_Errors is

    type Integer_Access_Zero is access Integer
        with Storage_Size => 0;

    procedure Free is
        new Ada.Unchecked_Deallocation
            (Object => Integer,
             Name   => Integer_Access_Zero);

    type Constant_Integer_Access is
        access constant Integer;

    -- ERROR: Cannot use access-to-constant type
    -- for Name
    procedure Free is
        new Ada.Unchecked_Deallocation
            (Object => Integer,
             Name   => Constant_Integer_Access);

    I : Integer_Access_Zero;

begin
    -- ERROR: Cannot allocate objects from
    -- empty storage pool
    I := new Integer;

    -- ERROR: Cannot deallocate objects from
    -- empty storage pool
    Free (I);
end Show_Unchecked_Deallocation_Errors;
```

**Code block metadata**


MD5: 5032d13b2eb6b7ca1979282dd6f98a

**Build output**

```
show_unchecked_deallocation_errors.adb:21:19: error: actual type must be access-to-variable type
show_unchecked_deallocation_errors.adb:21:19: error: instantiation abandoned
show_unchecked_deallocation_errors.adb:28:09: error: allocation from empty storage pool
show_unchecked_deallocation_errors.adb:32:04: error: deallocation from empty storage pool
gprbuild: *** compilation phase failed
```

Here, we see that trying to instantiate `Ada.Unchecked_Deallocation` for the `Constant_Integer_Access` type is rejected by the compiler. Similarly, we cannot allocate or deallocate an object for the `Integer_Access_Zero` type because its storage pool is empty.

28.1. Access Types
28.1.13 Null & Not Null Access

Note: This section was originally written by Robert A. Duff and published as Gem #23: Null Considered Harmful and Gem #24.

Ada, like many languages, defines a special null value for access types. All values of an access type designate some object of the designated type, except for null, which does not designate any object. The null value can be used as a special flag. For example, a singly-linked list can be null-terminated. A Lookup function can return null to mean "not found", presuming the result is of an access type:

Listing 111: show_null_return.ads

```ada
package Show_Null_Return is

   type Ref_Element is access all Element;

   Not_Found : constant Ref_Element := null;

   function Lookup (T : Table) return Ref_Element;
   -- Returns Not_Found if not found.

end Show_Null_Return;
```

An alternative design for Lookup would be to raise an exception:

Listing 112: show_not_found_exception.ads

```ada
package Show_Not_Found_Exception is

   Not_Found : exception;

   function Lookup (T : Table) return Ref_Element;
   -- Raises Not_Found if not found.
   -- Never returns null.

end Show_Not_Found_Exception;
```

Neither design is better in all situations; it depends in part on whether we consider the "not found" situation to be exceptional.

Clearly, the client calling Lookup needs to know whether it can return null, and if so, what that means. In general, it's a good idea to document whether things can be null or not, especially for formal parameters and function results. Prior to Ada 2005, we would do that with comments. Since Ada 2005, we can use the not null syntax:

197 https://www.adacore.com/gems/ada-gem-23
198 https://www.adacore.com/gems/ada-gem-24
Listing 113: show_not_null_return.ads

```ada
package Show_Not_Null_Return is
type Ref_Element is access all Element;

Not_Found : constant Ref_Element := null;

function Lookup (T : Table) return not null Ref_Element;
-- Possible since Ada 2005.
end Show_Not_Null_Return;
```

This is a complete package for the code snippets above:

Listing 114: example.ads

```ada
package Example is
type Element is limited private;
type Ref_Element is access all Element;
type Table is limited private;

Not_Found : constant Ref_Element := null;
function Lookup (T : Table) return Ref_Element;
-- Returns Not_Found if not found.

Not_Found_2 : exception;
function Lookup_2 (T : Table) return not null Ref_Element;
-- Raises Not_Found_2 if not found.

procedure P (X : not null Ref_Element);
procedure Q (X : not null Ref_Element);

private
type Element is limited
record
  Component : Integer;
end record;
type Table is limited null record;
end Example;
```

Listing 115: example.adb

```ada
package body Example is

An_Element : aliased Element;

function Lookup (T : Table) return Ref_Element is
  pragma Unreferenced (T);
begin
(continues on next page)
```

28.1. Access Types
In general, it's better to use the language proper for documentation, when possible, rather than comments, because compile-time and/or run-time checks can help ensure that the "documentation" is actually true. With comments, there's a greater danger that the comment will become false during maintenance, and false documentation is obviously a menace.

In many, perhaps most cases, null is just a tripping hazard. It's a good idea to put in not null when possible. In fact, a good argument can be made that not null should be the default, with extra syntax required when null is wanted. This is the way Standard ML\(^\text{199}\) works, for example — you don't get any special null-like value unless you ask for it. Of course, because Ada 2005 needs to be compatible with previous versions of the language, not null cannot be the default for Ada.

One word of caution: access objects are default-initialized to null, so if you have a not null object (or component) you had better initialize it explicitly, or you will get Constraint_Error. not null is more often useful on parameters and function results, for this reason.

Another advantage of `not null` over comments is for efficiency. Consider procedures `P` and `Q` in this example:

```
package Example.Processing is
  procedure P (X : not null Ref_Element);
  procedure Q (X : not null Ref_Element);
end Example.Processing;
```

```
package body Example.Processing is
  procedure P (X : not null Ref_Element) is
    begin
      X.all.Component := X.all.Component + 1;
    end P;

  procedure Q (X : not null Ref_Element) is
    begin
      for I in 1 .. 1000 loop
        P (X);
      end loop;
    end Q;
end Example.Processing;
```

Without `not null`, the generated code for `P` will do a check that `X /= null`, which may be costly on some systems. `P` is called in a loop, so this check will likely occur many times. With `not null`, the check is pushed to the call site. Pushing checks to the call site is usually beneficial because

1. the check might be hoisted out of a loop by the optimizer, or
2. the check might be eliminated altogether, as in the example above, where the compiler knows that An_Element'Access cannot be null.

This is analogous to the situation with other run-time checks, such as array bounds checks:

```
package Show_Process_Array is
  type My_Index is range 1 .. 10;
  type My_Array is array (My_Index) of Integer;

  procedure Process_Array
    (X   : in out My_Array;
     Index : My_Index);
end Show_Process_Array;
```
Listing 119: show_process_array.adb

```ada
package body Show_Process_Array is

    procedure Process_Array
    (X : in out My_Array;
     Index : My_Index) is
    begin
    X (Index) := X (Index) + 1;
    end Process_Array;

end Show_Process_Array;
```

28.1.14 Design strategies for access types

Previously, we learned about dangling references (page 795) and discussed the effects of dereferencing them (page 805). Also, we've seen the relationship between unchecked deallocation and dangling references (page 803). Ensuring that all calls to Free for a specific access type will never cause dangling references can become an arduous task — if not impossible — if those calls are located in different parts of the source code.

Although we used access types directly in the main application in many of the previous code examples from this chapter, this approach was in fact selected just for illustration purposes — i.e. to make the code look simpler. In general, however, we should avoid this approach. Instead, our recommendation is to encapsulate the access types in some form of abstraction. In this section, we discuss design strategies for access types that take this recommendation into account.

Abstract data type for access types

The simplest form of abstraction is of course an abstract data type. For example, we could declare a limited private type, which allows us to hide the access type and to avoid copies of references that could potentially become dangling references. (We discuss limited private types later in another chapter (page 928).)

Let's see an example:

Listing 120: access_type_abstraction.ads

```ada
package Access_Type_Abstraction is

type Info is limited private;

    function To_Info (S : String) return Info;

    function To_String (Obj : Info)
        return String;

    function Copy (Obj : Info) return Info;
```

(continues on next page)
procedure Copy (To : in out Info;
               From : Info);

procedure Append (Obj : in out Info;
                  S   : String);

procedure Reset (Obj : in out Info);

procedure Destroy (Obj : in out Info);

private

type Info is access String;

end Access_Type_Abstraction;
with Ada.Text_IO; use Ada.Text_IO;

with Access_Type_Abstraction; use Access_Type_Abstraction;

procedure Main is
    Obj_1 : Info := To_Info ("hello");
    Obj_2 : Info := Copy (Obj_1);
begin
    Put_Line ("TO_INFO / COPY");
    Put_Line ("Obj_1 : " & To_String (Obj_1));
    Put_Line ("Obj_2 : " & To_String (Obj_2));
    Put_Line ("----------");
    Reset (Obj_1);
    Append (Obj_2, " world");
    Put_Line ("RESET / APPEND");
    Put_Line ("Obj_1 : " & To_String (Obj_1));
    Put_Line ("Obj_2 : " & To_String (Obj_2));
    Put_Line ("----------");
    Copy (From => Obj_2,
         To => Obj_1);
    Put_Line ("COPY");
    Put_Line ("Obj_1 : " & To_String (Obj_1));
    Put_Line ("Obj_2 : " & To_String (Obj_2));
    Put_Line ("----------");
    Destroy (Obj_1);
    Destroy (Obj_2);
    Put_Line ("DESTROY");
    Put_Line ("Obj_1 : " & To_String (Obj_1));
    Put_Line ("Obj_2 : " & To_String (Obj_2));
    Put_Line ("----------");
    Append (Obj_1, " hey");
    Put_Line ("APPEND");
    Put_Line ("Obj_1 : " & To_String (Obj_1));
    Put_Line ("----------");
    Append (Obj_1, " there");
In this example, we hide an access type in the Info type — a limited private type. We allocate an object of this type in the To_Info function and deallocate it in the Destroy procedure. Also, we make sure that the reference isn't copied in the Copy function — we only copy the designated value in this function. This strategy eliminates the possibility of dangling references, as each reference is encapsulated in an object of Info type.

**Controlled type for access types**

In the previous code example, the Destroy procedure had to be called to deallocate the hidden access object. We could make sure that this deallocation happens automatically by using a controlled (or limited controlled) type. (We discuss controlled types in another chapter.)

Let's adapt the previous example and declare Info as a limited controlled type:

```ada
with Ada.Finalization;

package Access_Type_Abstraction is

  type Info is limited private;

end Access_Type_Abstraction;
```
function To_Info (S : String) return Info;

function To_String (Obj : Info) return String;

function Copy (Obj : Info) return Info;

procedure Copy (To : in out Info; From : Info);

procedure Append (Obj : in out Info; S : String);

procedure Reset (Obj : in out Info);

private

type String_Access is access String;

type Info is new Ada.Finalization.Limited_Controlled with record
   Str_A : String_Access;
end record;

procedure Initialize (Obj : in out Info);

procedure Finalize (Obj : in out Info);

end Access_Type_Abstraction;

with Ada.Unchecked_Deallocation;

package body Access_Type_Abstraction is

   -- STRING_ACCESS SUBPROGRAMS

   function To_String_Access (S : String) return String_Access is
      (new String'(S));

   function To_String (S : String_Access) return String is
      (if S /= null then S.all else "");

   procedure Free is
      new Ada.Unchecked_Deallocation
         (Object => String, Name => String_Access);

   -- PRIVATE SUBPROGRAMS

   procedure Initialize (Obj : in out Info) is
      begin

Listing 124: access_type_abstraction.adb

(continues on next page)
-- Put_Line ("Initializing Info");
Obj.Str_A := null;
-- ^^^^^^^^^^^^^
-- NOTE: This line has just been added to
-- illustrate the "automatic" call to
-- Initialize. Actually, this
-- assignment isn't needed, as
-- the Str_A component is
-- automatically initialized to null
-- upon object construction.
end Initialize;

procedure Finalize (Obj : in out Info) is
begin
-- Put_Line ("Finalizing Info");
Free (Obj.Str_A);
end Finalize;

-- PUBLIC SUBPROGRAMS
--
function To_Info (S : String) return Info is
(Ada.Finalization.Limited_Controlled
with Str_A => To_String_Access (S));

function To_String (Obj : Info) return String is
(To_String (Obj.Str_A));

function Copy (Obj : Info) return Info is
(To_Info (To_String (Obj.Str_A)));

procedure Copy (To : in out Info;
From : Info) is
begin
Free (To.Str_A);
To.Str_A := To_String_Access
(To_String (From.Str_A));
end Copy;

procedure Append (Obj : in out Info;
S : String) is
New_Str_A : constant String_Access :=
(To_String_Access
(To_String (Obj.Str_A) & S));
begin
Free (Obj.Str_A);
Obj.Str_A := New_Str_A;
end Append;

procedure Reset (Obj : in out Info) is
begin
Free (Obj.Str_A);
end Reset;

end Access_Type_Abstraction;

28.1. Access Types
with Ada.Text_IO; use Ada.Text_IO;

with Access_Type_Abstraction;
use Access_Type_Abstraction;

procedure Main is
  Obj_1 : Info := To_Info ("hello");
  Obj_2 : Info := Copy (Obj_1);
begin
  --
  -- TO_INFO / COPY
  --
  Put_Line ("TO_INFO / COPY");
  Put_Line ("Obj_1 : " & To_String (Obj_1));
  Put_Line ("Obj_2 : " & To_String (Obj_2));
  Put_Line ("--------");
  --
  -- RESET: Obj_1
  -- APPEND: Obj_2
  --
  Put_Line ("RESET / APPEND");
  Reset (Obj_1);
  Append (Obj_2, " world");
  Put_Line ("Obj_1 : " & To_String (Obj_1));
  Put_Line ("Obj_2 : " & To_String (Obj_2));
  Put_Line ("--------");
  --
  -- COPY: Obj_2 => Obj_1
  --
  Put_Line ("COPY");
  Copy (From => Obj_2, To => Obj_1);
  Put_Line ("Obj_1 : " & To_String (Obj_1));
  Put_Line ("Obj_2 : " & To_String (Obj_2));
  Put_Line ("--------");
  --
  -- RESET: Obj_1, Obj_2
  --
  Put_Line ("RESET");
  Reset (Obj_1);
  Reset (Obj_2);
  Put_Line ("Obj_1 : " & To_String (Obj_1));
  (continues on next page)
Put_Line ("Obj_2 : 
  & To_String (Obj_2));
Put_Line ("-------------");

--
-- COPY: Obj_2 => Obj_1
--
Put_Line ("COPY");
Copy (From => Obj_2,
  To => Obj_1);

Put_Line ("Obj_1 : 
  & To_String (Obj_1));
Put_Line ("Obj_2 : 
  & To_String (Obj_2));
Put_Line ("-------------");

--
-- APPEND: Obj_1 with "hey"
--
Put_Line ("APPEND");
Append (Obj_1, "hey");

Put_Line ("Obj_1 : 
  & To_String (Obj_1));
Put_Line ("-------------");

--
-- APPEND: Obj_1 with "there"
--
Put_Line ("APPEND");
Append (Obj_1, " there");

Put_Line ("Obj_1 : 
  & To_String (Obj_1));
end Main;
Of course, because we're using the Limited_Controlled type from the Ada Finalization package, we had to adapt the prototype of the subprograms from the Access_Type_Abstraction. In this version of the code, we only have the allocation taking place in the To_Info procedure, but we don't have a Destroy procedure for deallocation: this call was moved to the Finalize procedure.

Since objects of the Info type — such as Obj_1 in the Show_Access_Type_Abstraction procedure — are now controlled, the Finalize procedure is automatically called when they go out of scope. In this procedure, which we override for the Info type, we perform the deallocation of the internal access object Str_A. (You may uncomment the calls to Put_Line in the body of the Initialize and Finalize subprograms to confirm that these subprograms are called in the background.)

### 28.1.15 Access to subprograms

So far in this chapter, we focused mainly on access-to-objects. However, we can use access types to subprograms. This is the topic of this section.

#### Static vs. dynamic calls

In a typical subprogram call, we indicate the subprogram we want to call statically. For example, let's say we've implemented a procedure Proc that calls a procedure P:

```ada
procedure P (I : in out Integer);
```

```ada
procedure P (I : in out Integer) is
begin
  null;
end P;
```

```ada
with P;
procedure Proc is
  I : Integer := 0;
begin
  P (I);
end Proc;
```

The call to P is statically dispatched: every time Proc runs and calls P, that call is always to the same procedure. In other words, we can determine at compilation time which procedure is called.

In contrast, an access to a subprogram allows us to dynamically indicate which subprogram we want to call. For example, if we change Proc in the code above to receive the access to a subprogram P as a parameter, the actual procedure that would be called when running Proc would be determined at run time, and it might be different for every call to Proc. In this case, we wouldn't be able to determine at compilation time which procedure would be called in every case. (In some cases, however, it could still be possible to determine which procedure is called by analyzing the argument that is passed to Proc.)

### Access to subprogram declaration

We declare an access to a subprogram as a type by writing `access procedure` or `access function` and the corresponding prototype:

```ada
package Access_To_Subprogram_Types is
  type Access_To_Procedure is
    access procedure (I : in out Integer);
  type Access_To_Function is
    access function (I : Integer) return Integer;
end Access_To_Subprogram_Types;
```

In the designated profile of the access type declarations, we list all the parameters that we expect in the subprogram.

We can use those types to declare access to subprograms — as subprogram parameters, for example:

```ada
with Access_To_Subprogram_Types;
use Access_To_Subprogram_Types;
package Access_To_Subprogram_Params is
  procedure Proc (P : Access_To_Procedure);
end Access_To_Subprogram_Params;
```
package body Access_To_Subprogram_Params is

   procedure Proc (P : Access_To_Procedure) is
      I : Integer := 0;
      begin
         P (I);
         -- P.all (I);
      end Proc;

end Access_To_Subprogram_Params;

In the implementation of the Proc procedure of the code example, we call the P procedure by simply passing I as a parameter. In this case, P is automatically dereferenced. We may, however, explicitly dereference P by writing P.all (I).

Before we use this package, let's implement a simple procedure that we'll use later on:

    procedure Add_Ten (I : in out Integer);

    procedure Add_Ten (I : in out Integer) is
      begin
         I := I + 10;
      end Add_Ten;

Now, we can get access to a subprogram by using the Access attribute and pass it as an actual parameter:

    with Access_To_Subprogram_Params;
    use Access_To_Subprogram_Params;

    with Add_Ten;

    procedure Show_Access_To_Subprograms is
      begin
         Proc (Add_Ten'Access);
         -- Getting access to Add_Ten procedure and passing it
         -- to Proc
      end Show_Access_To_Subprograms;

Code block metadata

MD5: 17c1a07f48d9fb0efef37aa4c5ec8a51

In the implementation of the Proc procedure of the code example, we call the P procedure by simply passing I as a parameter. In this case, P is automatically dereferenced. We may, however, explicitly dereference P by writing P.all (I).

Before we use this package, let's implement a simple procedure that we'll use later on:

Listing 132: add_ten.ads

      procedure Add_Ten (I : in out Integer);

Listing 133: add_ten.adb

      procedure Add_Ten (I : in out Integer) is
         begin
            I := I + 10;
         end Add_Ten;

Listing 134: show_access_to_subprograms.adb

      with Access_To_Subprogram_Params;
      use Access_To_Subprogram_Params;

      with Add_Ten;

      procedure Show_Access_To_Subprograms is
         begin
            Proc (Add_Ten'Access);
            -- Getting access to Add_Ten procedure and passing it
            -- to Proc
         end Show_Access_To_Subprograms;

Code block metadata

MD5: 8553ad7329bf1ed727147b47b7355a70

Now, we can get access to a subprogram by using the Access attribute and pass it as an actual parameter:
Here, we get access to the Add_Ten procedure and pass it to the Proc procedure.

In the Ada Reference Manual

- 3.10 Access Types

Objects of access-to-subprogram type

In the previous example, the Proc procedure had a parameter of access-to-subprogram type. In addition to parameters, we can of course declare objects of access-to-subprogram types as well. For example, we can extend our previous test application and declare an object P of access-to-subprogram type. Before we do so, however, let's implement another small procedure that we'll use later on:

Listing 135: add_twenty.ads

```ada
procedure Add_Twenty (I : in out Integer);
```

Listing 136: add_twenty.adb

```ada
procedure Add_Twenty (I : in out Integer) is
begin
  I := I + 20;
end Add_Twenty;
```

Code block metadata

In addition to Add_Ten, we've implemented the Add_Twenty procedure, which we use in our extended test application:

Listing 137: show_access_to_subprograms.adb

```ada
with Access_To_Subprogram_Types;
use Access_To_Subprogram_Types;
with Access_To_Subprogram_Params;
use Access_To_Subprogram_Params;
with Add_Ten;
with Add_Twenty;

procedure Show_Access_To_Subprograms is
  P : Access_To_Procedure;
  Some_Int : Integer := 0;
begin
  P := Add_Ten'Access;
  -- ^ Getting access to Add_Ten
  -- procedure and assigning it
```

(continues on next page)
In the `Show_Access_To_Subprograms` procedure, we see the declaration of our access-to-subprogram object `P` (of `Access_To_Procedure` type). We get access to the `Add_Ten` procedure and assign it to `P`, and we then do the same for the `Add_Twenty` procedure.

We can use an access-to-subprogram object either as the actual parameter of a subprogram call, or in a subprogram call. In the code example, we're passing `P` as the actual parameter of the `Proc` procedure in the `Proc (P)` calls. Also, we're calling the subprogram assigned to (designated by the current value of) `P` in the `P (Some_Int)` calls.

### Components of access-to-subprogram type

In addition to declaring subprogram parameters and objects of access-to-subprogram types, we can declare components of these types. For example:

```ada
package Access_To_Subprogram_Types is

  type Access_To_Procedure is
    access procedure (I : in out Integer);

  type Access_To_Function is
    access function (I : Integer) return Integer;

  type Access_To_Procedure_Array is
    array (Positive range <>) of
      Access_To_Procedure;

  type Access_To_Function_Array is
    array (Positive range <>) of
      Access_To_Function;

  type Rec_Access_To_Procedure is record
    AP : Access_To_Procedure;

end Access_To_Subprogram_Types;
```

(continues on next page)
Here, the access-to-procedure type `Access_To_Procedure` is used as a component of the array type `Access_To_Procedure_Array` and the record type `Rec_Access_To_Procedure`. Similarly, the access-to-function type `Access_To_FUNCTION` type is used as a component of the array type `Access_To_FUNCTION_Array` and the record type `Rec_Access_To_FUNCTION`.

Let's see two test applications using these types. First, let's use the `Access_To_Procedure_Array` array type in a test application:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Access_To_Subprogram_Types;
use Access_To_Subprogram_Types;
with Add_Ten;
with Add_Twenty;

procedure Show_Access_To_Subprograms is
  PA : constant Access_To_Procedure_Array (1 .. 2) :=
    (Add_Ten'Access,
     Add_Twenty'Access);
  Some_Int : Integer := 0;
begin
  Put_Line ("Some_Int: " & Some_Int'Image);
  for I in PA'Range loop
    PA (I) (Some_Int);
    Put_Line ("Some_Int: " & Some_Int'Image);
  end loop;
end Show_Access_To_Subprograms;
```

Here, we declare the PA array and use the access to the Add_Ten and Add_Twenty procedures as its components. We can call any of these procedures by simply specifying the
As you might expect, we can use access-to-subprogram types when declaring discriminants. In fact, when we were talking about discriminants as access values (page 745) earlier on, we used access-to-object types in our code examples, but we could have used access-to-subprogram types as well. For example:

```
package Custom_Processing is

  -- Declaring an access type:
type Integer_Processing is
    access procedure (I : in out Integer);
```

-- Declaring a discriminant with this
-- access type:

type Rec (IP : Integer_Processing) is
  private;

procedure Init (R : in out Rec;
  Value : Integer);

procedure Process (R : in out Rec);

procedure Show (R : Rec);

private

  type Rec (IP : Integer_Processing) is
    record
      I : Integer := 0;
    end record;

end Custom_Processing;

Listing 142: custom_processing.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Custom_Processing is

  procedure Init (R : in out Rec;
    Value : Integer) is
  begin
    R.I := Value;
    end Init;

  procedure Process (R : in out Rec) is
  begin
    R.IP (R.I);
    -- ^^^^^^ 
    -- Calling procedure that we specified as
    -- the record's discriminant
    end Process;

  procedure Show (R : Rec) is
  begin
    Put_Line ("R.I = " & Integer'Image (R.I));
    end Show;

end Custom_Processing;

Code block metadata

Subprograms.Access_To_Subprogram_Types
MD5: 02fc0c5172c321c4ec6115de68d1c06

In this example, we declare the access-to-subprogram type Integer_Processing, which
we use as the IP discriminant of the Rec type. In the Process procedure, we call the IP
procedure that we specified as the record's discriminant (R.IP (R.I)).

Before we look at a test application for this package, let's implement another small proce-
dure:
Now, let’s look at the test application:

```
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Processing; use Custom_Processing;
with Add_Ten;
with Mult_Two;

procedure Show_Access_To_Subprogram_Discriminants is
  R_Add_Ten : Rec (IP => Add_Ten'Access);
  -- Using access-to-subprogram as a discriminant
  R_Mult_Two : Rec (IP => Mult_Two'Access);
  -- Using access-to-subprogram as a discriminant

begin
  Init (R_Add_Ten, 1);
  Init (R_Mult_Two, 2);
  Put_Line ("---- R_Add_Ten ----");
  Show (R_Add_Ten);
  Put_Line ("Calling Process procedure...");
  Process (R_Add_Ten);
  Show (R_Add_Ten);
  Put_Line ("---- R_Mult_Two ----");
  Show (R_Mult_Two);
  Put_Line ("Calling Process procedure...");
  Process (R_Mult_Two);
  Show (R_Mult_Two);
end Show_Access_To_Subprogram_Discriminants;
```
Runtime output

---- R_Add_Ten ----
R.I = 1
Calling Process procedure...
R.I = 11
---- R_Mult_Two ----
R.I = 2
Calling Process procedure...
R.I = 4

In this procedure, we declare the R_Add_Ten and R_Mult_Two of Rec type and specify the access to Add_Ten and Mult_Two, respectively, as the IP discriminant. The procedure we specified here is then called inside a call to the Process procedure.

**Access-to-subprograms as formal parameters**

We can use access-to-subprograms types when declaring formal parameters. For example, let's revisit the Custom_Processing package from the previous section and convert it into a generic package.

### Listing 146: gen_custom_processing.ads

```ada
generic
  type T is private;

  -- Declaring formal access-to-subprogram
  -- type:
  --
  type T_Processing is
    access procedure (Element : in out T);

  -- Declaring formal access-to-subprogram
  -- parameter:
  --
  Proc : T_Processing;

  with function Image_T (Element : T)
  return String;
package Gen_Custom_Processing is
  type Rec is private;

  procedure Init (R : in out Rec;
                  Value : in out T);

  procedure Process (R : in out Rec);

  procedure Show (R : Rec);
private
  type Rec is record
    Comp : T;
```

(continues on next page)
Listing 147: gen_custom_processing.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Gen_Custom_Processing is
  procedure Init (R : in out Rec; Value : T) is
    begin
      R.Comp := Value;
    end Init;

  procedure Process (R : in out Rec) is
    begin
      Proc (R.Comp);
    end Process;

  procedure Show (R : Rec) is
    begin
      Put_Line ("R.Comp = " & Image_T (R.Comp));
    end Show;

end Gen_Custom_Processing;
```

Code block metadata

MD5: 6f06e066bafa5f02abb3ee1b33ea0831

In this version of the procedure, instead of declaring Proc as a discriminant of the Rec record, we're declaring it as a formal parameter of the Gen_Custom_Processing package. Also, we're declaring an access-to-subprogram type (T_Processing) as a formal parameter. (Note that, in contrast to these two parameters that we've just mentioned, Image_T is not a formal access-to-subprogram parameter: it's actually just a formal subprogram.)

We then instantiate the Gen_Custom_Processing package in our test application:

Listing 148: show_access_to_subprogram_as_formal_parameter.adb

```ada
with Gen_Custom_Processing;

with Add_Ten;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Access_To_Subprogram_As_Formal_Parameter is
  type Integer_Processing is
    access procedure (I : in out Integer);

  package Custom_Processing is new
    Gen_Custom_Processing
      (T => Integer,
       T_Processing => Integer_Processing,
       ...)
```
Here, we instantiate the Gen_Custom_Processing package as Custom_Processing and specify the access-to-subprogram type and the access-to-subprogram.

### Selecting subprograms

A practical application of access to subprograms is that it enables us to dynamically select a subprogram and pass it to another subprogram, where it can then be called.

For example, we may have a Process procedure that receives a logging procedure as a parameter (Log_Proc). Also, this parameter may be **null** by default — so that no procedure is called if the parameter isn't specified:

```ada
package Data_Processing is
    type Data_Container is array (Positive range <>) of Float;
    type Log_Procedure is access procedure (D : Data_Container);
    procedure Process (D : in out Data_Container;
                        Log_Proc : Log_Procedure := null);
end Data_Processing;
```

(continues on next page)
Listing 150: data_processing.adb

```ada
package body Data_Processing is

procedure Process 
  (D : in out Data_Container;
   Log_Proc : Log_Procedure := null) is
begin
  -- missing processing part...
  if Log_Proc /= null then
    Log_Proc (D);
  end if;
end Process;
end Data_Processing;
```

### Code block metadata


**MD5**: 59399e0809deb476f608faab7e4398bd

In the implementation of Process, we check whether Log_Proc is null or not. (If it's not null, we call the procedure. Otherwise, we just skip the call.)

Now, let's implement two logging procedures that match the expected form of the Log_Procedure type:

Listing 151: log_element_per_line.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Data_Processing; use Data_Processing;

procedure Log_Element_Per_Line 
  (D : Data_Container) is
begin
  Put_Line ("Elements: ");
  for V of D loop
    Put_Line (V'Image);
  end loop;
  Put_Line ("------");
end Log_Element_Per_Line;
```

Listing 152: log_csv.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Data_Processing; use Data_Processing;

procedure Log_Csv (D : Data_Container) is
begin
  for I in D'First .. D'Last - 1 loop
    Put (D (I)'Image & ", ");
  end loop;
  Put (D (D'Last)'Image);
  New_Line;
end Log_Csv;
```
Finally, we implement a test application that selects each of the logging procedures that we've just implemented:

```
with Ada.Text_IO;    use Ada.Text_IO;
with Data_Processing; use Data_Processing;
with Log_Element_Per_Line;
with Log_Csv;

procedure Show_Access_To_Subprograms is
  D : Data_Container (1 .. 5) := (others => 1.0);
begin
  Put_Line ("==== Log_Element_Per_Line ====");
  Process (D, Log_Element_Per_Line'Access);
  Put_Line ("==== Log_Csv ====");
  Process (D, Log_Csv'Access);
  Put_Line ("==== None ====");
  Process (D);
end Show_Access_To_Subprograms;
```

Here, we use the `Access` attribute to get access to the `Log_Element_Per_Line` and `Log_Csv` procedures. Also, in the third call, we don't pass any access as an argument, which is then `null` by default.
Null exclusion

We can use null exclusion when declaring an access to subprograms. By doing so, we ensure that a subprogram must be specified — either as a parameter or when initializing an access object. Otherwise, an exception is raised. Let's adapt the previous example and introduce the Init_Function type:

Listing 154: data_processing.ads

```ada
package Data_Processing is

  type Data_Container is array (Positive range <>) of Float;

  type Init_Function is not null access function return Float;

  procedure Process (
    D       : in out Data_Container;
    Init_Func : Init_Function);

end Data_Processing;
```

Listing 155: data_processing.adb

```ada
package body Data_Processing is

  procedure Process (
    D       : in out Data_Container;
    Init_Func : Init_Function) is
  begin
    for I in D'Range loop
      D (I) := Init_Func.all;
    end loop;
  end Process;

end Data_Processing;
```

In this case, we specify that Init Function is not null access because we want to always be able to call this function in the Process procedure (i.e. without raising an exception).

When an access to a subprogram doesn't have parameters — which is the case for the subprograms of Init_Function type — we need to explicitly dereference it by writing .all. (In this case, .all isn't optional.) Therefore, we have to write Init_Func.all in the implementation of the Process procedure of the code example.

Now, let's declare two simple functions — Init_Zero and Init_One — that return 0.0 and 1.0, respectively:

Listing 156: init_zero.ads

```ada
function Init_Zero return Float;
```

Listing 157: init_one.ads

```ada
function Init_One return Float;
```

Listing 158: init_zero.adb

```ada
function Init_Zero return Float is
begin
  (continues on next page)
```

834 Chapter 28. Resource Management
Finally, let's see a test application where we select each of the init functions we've just implemented:

Listing 160: log_element_per_line.adb

```
with Ada.Text_IO;    use Ada.Text_IO;
with Data_Processing; use Data_Processing;

procedure Log_Element_Per_Line (D : Data_Container) is
begin
  Put_Line ("Elements: ");
  for V of D loop
    Put_Line (V'Image);
  end loop;
  Put_Line ("------");
end Log_Element_Per_Line;
```

Listing 161: show_access_to_subprograms.adb

```
with Ada.Text_IO;    use Ada.Text_IO;
with Data_Processing; use Data_Processing;
with Init_Zero;
with Init_One;
with Log_Element_Per_Line;

procedure Show_Access_To_Subprograms is
  D : Data_Container (1 .. 5) := (others => 1.0);
begin
  Put_Line ("==== Init_Zero ====");
  Process (D, Init_Zero'Access);
  Log_Element_Per_Line (D);
  Put_Line ("==== Init_One ====");
  Process (D, Init_One'Access);
  Log_Element_Per_Line (D);
  -- Put_Line ("==== None ====");
  -- Process (D, null);
  -- Log_Element_Per_Line (D);
end Show_Access_To_Subprograms;
```

Code block metadata


**MD5**: 444110d50ddb430fd5be31cf1b417fc8
Here, we use the Access attribute to get access to the Init_Zero and Init_One functions. Also, if we uncomment the call to Process with null as an argument for the init function, we see that the Constraint_Error exception is raised at run time — as the argument cannot be null due to the null exclusion.

For further reading...

Note: This example was originally written by Robert A. Duff and was part of the Gem #24[^201].

Here’s another example, first with null:

^201[^201]: This example was originally written by Robert A. Duff and was part of the Gem #24[^201].
and without null:

```ada
package Show_Null_Procedure is
  type Element is limited null record;
  -- Not implemented yet

  type Ref_Element is access all Element;

  type Table is limited null record;
  -- Not implemented yet

  procedure Do_Nothing
    (X : not null Ref_Element) is null;

  type Iterate_Action is
    access procedure
      (X : not null Ref_Element);

  procedure Iterate
    (T : Table;
    Action : not null Iterate_Action
      := Do_Nothing'Access);
end Show_Null_Procedure;
```

The style of the second `Iterate` is clearly better because it makes use of the syntax to indicate that a procedure is expected. This is a complete package that includes both versions of the `Iterate` procedure:

```ada
package Example is
  type Element is limited private;
  type Ref_Element is access all Element;

  type Table is limited private;

  type Iterate_Action is
    access procedure
      (X : not null Ref_Element);

  procedure Iterate
    (T : Table;
    Action : Iterate_Action := null);
  -- If Action is null, do nothing.

  procedure Do_Nothing
    (X : not null Ref_Element) is null;

  procedure Iterate_2
    (continues on next page)
```
Writing not null Iterate_Action might look a bit more complicated, but it's worthwhile, and anyway, as mentioned earlier, the compatibility requirement requires that the not null
be explicit, rather than the other way around.

Access to protected subprograms

Up to this point, we've discussed access to normal Ada subprograms. In some situations, however, we might want to have access to protected subprograms. To do this, we can simply declare a type using `access protected`:

Listing 167: simple_protected_access.ads

```ada
package Simple_Protected_Access is
    type Access_Proc is
        access protected procedure;
    protected Obj is
        procedure Do_Something;
    end Obj;
    Acc : Access_Proc := Obj.Do_Something'Access;
end Simple_Protected_Access;
```

Listing 168: simple_protected_access.adb

```ada
package body Simple_Protected_Access is
    protected body Obj is
        procedure Do_Something is
            begin
                -- Not doing anything
                -- for the moment...
                null;
            end Do_Something;
        end Obj;
end Simple_Protected_Access;
```

Here, we declare the Access_Proc type as an access type to protected procedures. Then, we declare the variable Acc and assign to it the access to the Do_Something procedure (of the protected object Obj).

Now, let's discuss a more useful example: a simple system that allows us to register protected procedures and execute them. This is implemented in Work_Registry package:

Listing 169: work_registry.ads

```ada
package Work_Registry is
```

(continues on next page)
type Work_Id is tagged limited private;

type Work_Handler is
  access protected procedure (T : Work_Id);

subtype Valid_Work_Handler is
  not null Work_Handler;

type Work_Handlers is
  array (Positive range <>) of Work_Handler;

protected type Work_Handler_Registry
  (Last : Positive)
  is
    procedure Register (T : Valid_Work_Handler);
    procedure Reset;
    procedure Process_All;

private
  D : Work_Handlers (1 .. Last);
  Curr : Natural := 0;
end Work_Handler_Registry;

private
  type Work_Id is tagged limited null record;
end Work_Registry;

package body Work_Registry is

protected body Work_Handler_Registry is

  procedure Register (T : Valid_Work_Handler) is
    begin
      if Curr < Last then
        Curr := Curr + 1;
        D (Curr) := T;
      end if;
    end Register;

  procedure Reset is
    begin
      Curr := 0;
    end Reset;

  procedure Process_All is
    Dummy_ID : Work_Id;
    begin
      for I in D'First .. Curr loop
        D (I).all (Dummy_ID);
      end loop;
    end Process_All;

end Work_Handler_Registry;

package body work_registry.adb
Here, we declare the protected Work_Handler_Registry type with the following subprograms:

- Register, which we can use to register a protected procedure;
- Reset, which we can use to reset the system; and
- Process_All, which we can use to call all procedures that were registered in the system.

Work_Handler is our access to protected subprogram type. Also, we declare the Valid_Work_Handler subtype, which excludes null. By doing so, we can ensure that only valid procedures are passed to the Register procedure. In the protected Work_Handler_Registry type, we store the procedures in an array (of Work_Handlers type).

Important

Note that, in the type declaration Work_Handler, we say that the protected procedure must have a parameter of Work_Id type. In this example, this parameter is just used to bind the procedure to the Work_Handler_Registry type. The Work_Id type itself is actually declared as a null record (in the private part of the package), and it isn't really useful on its own.

If we had declared type Work_Handler is access protected procedure; instead, we would be able to register any protected procedure into the system, even the ones that might not be suitable for the system. By using a parameter of Work_Id type, however, we make use of strong typing to ensure that only procedures that were designed for the system can be registered.

In the next part of the code, we declare the Integer_Storage type, which is a simple protected type that we use to store an integer value:

```ada
with Work_Registry;
package Integer_Storage_System is
  protected type Integer_Storage is
    procedure Set (V : Integer);
    procedure Show (T : Work_Registry.Work_Id);
  private
    I : Integer := 0;
end Integer_Storage;
```
For the Integer_Storage type, we declare two procedures:

- Set, which we use to assign a value to the (protected) integer value; and
- Show, which we use to show the integer value that is stored in the protected object.

The Show procedure has a parameter of Work_Id type, which indicates that this procedure was designed to be registered in the system of Work_Handler_Registry type.

Finally, we have a test application in which we declare a registry (WHR) and an array of "protected integer objects" (Int_Stor):

```
with Ada.Text_IO; use Ada.Text_IO;
package body Integer_Storage_System is
  protected body Integer_Storage is
    procedure Set (V : Integer) is
      begin
        I := V;
        end Set;

    procedure Show (T : Work_Registry.Work_Id) is
      pragma Unreferenced (T);
      begin
        Put_Line ("Value: " & Integer'Image (I));
        end Show;
  end Integer_Storage;
end Integer_Storage_System;
```
Learning Ada

WHR : Work_Handler_Registry (5);
Int_Stor : Integer_Storage_Array (1 .. 3);
begin
-- Allocate and initialize integer storage
-- (For the initialization, we're just
-- assigning the index here, but we could
-- really have used any integer value.)
for I in Int_Stor'Range loop
  Int_Stor (I) := new Integer_Storage;
  Int_Stor (I).Set (I);
end loop;
-- Register handlers
for I in Int_Stor'Range loop
  WHR.Register (Int_Stor (I).all.Show'Access);
end loop;
-- Now, use Process_All to call the handlers
-- (in this case, the Show procedure for
-- each protected object from Int_Stor).
WHR.Process_All;
end Show_Access_To_Protected_Subprograms;

The work handler registry (WHR) has a maximum capacity of five procedures, whereas the
Int_Stor array has a capacity of three elements. By calling WHR.Register and passing
Int_Stor (I).all.Show'Access, we register the Show procedure of each protected object from Int_Stor.

Important
Note that the components of the Int_Stor array are of Integer_Storage_Access type,
which is declared as an access to Integer_Storage objects. Therefore, we have to dereference the object (by writing Int_Stor (I).all) before getting access to the Show procedure (by writing .Show'Access).

We have to use an access type here because we cannot pass the access (to the Show
procedure) of a local object in the call to the Register procedure. Therefore, the protected
objects (of Integer_Storage type) cannot be local.

This issue becomes evident if we replace the declaration of Int_Stor with a local array
(and then adapt the remaining code). If we do this, we get a compilation error in the call to
Register:

28.1. Access Types 843
with Work_Registry;
use Work_Registry;

with Integer_Storage_System;
use Integer_Storage_System;

procedure Show_Access_To_Protected_Subprograms
is
  WHR : Work_Handler_Registry (5);
  Int_Stor : array (1 .. 3) of Integer_Storage;

begin
  -- Allocate and initialize integer storage
  -- (For the initialization, we're just
  -- assigning the index here, but we could
  -- really have used any integer value.)
  for I in Int_Stor'Range loop
    Int_Stor (I).Set (I);
  end loop;
  -- Register handlers
  for I in Int_Stor'Range loop
    WHR.Register (Int_Stor (I).Show'Access);
    ^ ERROR!
  end loop;

  -- Now, call the handlers
  -- (i.e. the Show procedure of each
  -- protected object).
  WHR.Process_All;

end Show_Access_To_Protected_Subprograms;

Code block metadata

MD5: 359241c84cd30313fe2d7701b55f303e

Build output

show_access_to_protected_subprograms.adb:28:21: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed

As we've just discussed, this error is due to the fact that Int_Stor is now a "local" protected object, and the accessibility rules don't allow mixing it with non-local accesses in order to prevent the possibility of dangling references.

When we call WHR.Process_All, the registry system calls each procedure that has been registered with the system. When looking at the values displayed by the test application, we may notice that each call to Show is referring to a different protected object. In fact, even though we're passing just the access to a protected procedure in the call to Register,
that access is also associated to a specific protected object. (This is different from access to non-protected subprograms we’ve discussed previously: in that case, there’s no object associated.) If we replace the argument to Register by \texttt{Int\_Stor (2).all.Show’Access}, for example, the three Show procedures registered in the system will now refer to the same protected object (stored at \texttt{Int\_Stor (2)}).

Also, even though we have registered the same procedure (Show) of the same type (\texttt{Integer\_Storage}) in all calls to Register, we could have used a different protected procedure — and of a different protected type. As an exercise, we could, for example, create a new type called \texttt{Float\_Storage} (based on the code that we used for the \texttt{Integer\_Storage} type) and register some objects of \texttt{Float\_Storage} type into the system (with a couple of additional calls to Register). If we then call \texttt{WHR.Process\_All}, we’d see that the system is able to cope with objects of both \texttt{Integer\_Storage} and \texttt{Float\_Storage} types. In fact, the system implemented with the \texttt{Work\_Handler\_Registry} can be seen as “type agnostic,” as it doesn’t care about which type the protected objects have — as long as the subprograms we want to register are conformant to the \texttt{Valid\_Work\_Handler} type.

### 28.1.16 Accessibility Rules and Access-To-Subprograms

In general, the accessibility rules that we discussed previously for access-to-objects (page 788) also apply to access-to-subprograms. In this section, we discuss minor differences when applying those rules to access-to-subprograms.

In our discussion about accessibility rules, we’ve looked into accessibility levels (page 789) and the accessibility rules (page 790) that are based on those levels. The same accessibility rules apply to access-to-subprograms. As we said previously (page 793), operations targeting objects at a \textit{less-deep} level are illegal, as it’s the case for subprograms as well:

```ada
package Access_To_Subprogram_Types is
  type Access_To_Procedure is
    access procedure (I : in out Integer);

  type Access_To_Function is
    access function (I : Integer) return Integer;
end Access_To_Subprogram_Types;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Access_To_Subprogram_Types; use Access_To_Subprogram_Types;
procedure Show_Access_To_Subprogram_Error is
  Func : Access_To_Function;
begin
  Value : Integer := 0;
  begin
    declare
      function Add_One (I : Integer)
        return Integer is
          (I + 1); begin
        Func := Add_One’Access;
        -- This assignment is illegal because the
```

(continues on next page)
-- Access_To_Function type is less deep
-- than Add_One.
end;

Put_Line ("Value: " & Value'Image);
Value := Func (Value);
Put_Line ("Value: " & Value'Image);
end Show_Access_To_Subprogram_Error;

Obviously, we can correct this error by putting the Add_One function at the same level as the Access_To_Function type, i.e. at library level:

Listing 177: access_to_subprogram_types.ads

package Access_To_Subprogram_Types is

  type Access_To_Procedure is
    access procedure (I : in out Integer);

  type Access_To_Function is
    access function (I : Integer) return Integer;

end Access_To_Subprogram_Types;

Listing 178: add_one.ads

function Add_One (I : Integer) return Integer;

Listing 179: add_one.adb

function Add_One (I : Integer) return Integer is
begin
  return I + 1;
end Add_One;

Listing 180: show_access_to_subprogram_error.adb

with Ada.Text_IO; use Ada.Text_IO;

with Access_To_Subprogram_Types;
use Access_To_Subprogram_Types;

with Add_One;

procedure Show_Access_To_Subprogram_Error is
  Func : Access_To_Function;
begin
  Value : Integer := 0;
end Show_Access_To_Subprogram_Error;

(continues on next page)
begin
  Func := Add_One'Access;
  Put_Line ("Value: " & Value'Image);
  Value := Func (Value);
  Put_Line ("Value: " & Value'Image);
end Show_Access_To_Subprogram_Error;

Code block metadata
MD5: 7f7488c541fb457ced653a2e6cc2fad1

Runtime output
Value:  0
Value:  1

As a recommendation, resolving accessibility issues in the case of access-to-subprograms is best done by refactoring the subprograms of your source code — for example, moving subprograms to a different level.

Unchecked Access

Previously, we discussed about the *Unchecked Access attribute* (page 798), which we can use to circumvent accessibility issues in specific cases for access-to-objects. We also said in that section that this attribute only exists for objects, not for subprograms. We can use the previous example to illustrate this limitation:

Listing 181: access_to_subprogram_types.ads

```ada
package Access_To_Subprogram_Types is
  type Access_To_Procedure is
    access procedure (I : in out Integer);
  type Access_To_Function is
    access function (I : Integer) return Integer;
end Access_To_Subprogram_Types;
```

Listing 182: show_access_to_subprogram_error.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Access_To_Subprogram_Types;
use Access_To_Subprogram_Types;
procedure Show_Access_To_Subprogram_Error is
  Func : Access_To_Function;
  function Add_One (I : Integer) return Integer is
    (I + 1);
  begin
    Value : Integer := 0;
    Func := Add_One'Access;
```

---

28.1. Access Types 847
When we analyze the `Show_Access_To_Subprogram_Error` procedure, we see that the `Func` object and the `Add_One` function have the same lifetime. Therefore, in this very specific case, we could safely assign `Add_One'Access` to `Func` and call `Func` for `Value`. Due to the accessibility rules, however, this assignment is illegal. (Obviously, the accessibility issue here is that the `Access_To_Function` type has a potentially longer lifetime.)

In the case of access-to-objects, we could use `Unchecked_Access` to enforce assignments that we consider safe after careful analysis. However, because this attribute isn’t available for access-to-subprograms, the best solution is to move the subprogram to a level that allows the assignment to be legal, as we said before.

### In the GNAT toolchain

GNAT offers an equivalent for `Unchecked_Access` that can be used for subprograms: the `Unrestricted_Access` attribute. Note, however, that this attribute is not portable.

#### Listing 183: `access_to_subprogram_types.ads`

```ada
package Access_To_Subprogram_Types is
  type Access_To_Procedure is
    access procedure (I : in out Integer);
  type Access_To_Function is
    access function (I : Integer) return Integer;
end Access_To_Subprogram_Types;
```

#### Listing 184: `show_access_to_subprogram_error.adb`

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Access_To_Subprogram_Types;
use Access_To_Subprogram_Types;
procedure Show_Access_To_Subprogram_Error is
  Func : Access_To_Function;
  function Add_One (I : Integer) return Integer is
    (I + 1);
```
Value : Integer := 0;

begin
  Func := Add_One'Unrestricted Access;
  -- Allowing access to local function
  Put_Line ("Value: " & Value'Image);
  Value := Func (Value);
  Put_Line ("Value: " & Value'Image);
end Show_Access_To_Subprogram_Error;

As we can see, the Unrestricted_Access attribute can be safely used in this specific case to circumvent the accessibility rule limitation.

28.1.17 Access and Address

As we know, an access type is not a pointer, and it doesn't just indicate an address in memory. In fact, to represent an address in Ada, we use the Address type (page 401). Also, as we discussed earlier, we can use operators such as <, >, + and - for addresses. In contrast to that, those operators aren't available for access types — except, of course, for = and /=.

In certain situations, however, we might need to convert between access types and addresses. In this section, we discuss how to do so.

Address and access conversion

The generic System.Address_To_Access_Conversions package allows us to convert between access types and addresses. This might be useful for specific low-level operations. Let's see an example:

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes\(^{202}\)
- 13.7 The Package System\(^ {203}\)

---


Learning Ada

Listing 185: show_address_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System.Address_To_Access_Conversions;
with System.Address_Image;

procedure Show_Address_Conversion is

   package Integer_AAC is
      new System.Address_To_Access_Conversions
         (Object => Integer);
      use Integer_AAC;

   subtype Integer_Access is Integer_AAC.Object_Pointer;
      -- This is similar to:
      --
      -- type Integer_Access is access all Integer;

   I : aliased Integer := 5;
   AI : Integer_Access := I'Access;

begin
   Put_Line ("I'Address : 
      & System.Address_Image (I'Address));

   Put_Line ("AI.all'Address : 
      & System.Address_Image
      (AI.all'Address));

   Put_Line ("To_Address (AI) : 
      & System.Address_Image
      (To_Address (AI)));
end Show_Address_Conversion;
```

In this example, we instantiate the generic `System.Address_To_Access_Conversions` package using `Integer` as our target object type. This new package (Integer_AAC) has an `Object_Pointer` type, which is equivalent to a declaration such as `type Integer_Access is access all Integer`. (In this example, we declare `Integer_Access` as a subtype of `Integer_AAC.Object_Pointer` to illustrate that.)

The `Integer_AAC` package also includes the `To_Address` function, which converts an access object to an address. If the actual parameter is not null, `To_Address` returns the same information as if we were using the `Address` attribute for the designated object. In other words, `To_Address (AI) = AI.all'Address` when `AI /= null`.

If the access value is null, `To_Address` returns `Null_Address`, while `.all'Address` makes the `access check` (page 662) fail because we have to dereference the access object (via `.all`) before retrieving its address (via the `Address` attribute).

In addition to the `To_Address` function, the `To_Pointer` function is available to convert
from an address to an object of access type. For example:

Listing 186: show_address_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System; use System;
with System.Address_To_Access_Conversions;
with System.Address_Image;

procedure Show_Address_Conversion is
  package Integer_AAC is
    new System.Address_To_Access_Conversions
      (Object => Integer);
    use Integer_AAC;
  
  subtype Integer_Access is
    Integer_AAC.Object_Pointer;

  I    : aliased Integer := 5;
  AI_1, AI_2 : Integer_Access;
  A    : Address;

begin
  AI_1 := I'Access;
  A    := To_Address (AI_1);
  AI_2 := To_Pointer (A);

  Put_Line ("AI_1.all'Address : 
            & System.Address_Image
            (AI_1.all'Address));
  Put_Line ("AI_2.all'Address : 
            & System.Address_Image
            (AI_2.all'Address));

  if AI_1 = AI_2 then
    Put_Line ("AI_1 = AI_2");
  else
    Put_Line ("AI_1 /= AI_2");
  end if;
end Show_Address_Conversion;
```

Code block metadata

   Address_Conversion
MD5: 5c6fc19calaa227feba97ea610dd9218

Runtime output

AI_1.all'Address : 00007FFF08F33EEC
AI_2.all'Address : 00007FFF08F33EEC
AI_1 = AI_2

Here, we convert the A address back to an access value by calling To_Pointer (A). (When running this object, we see that AI_1 and AI_2 have the same access value.)
Conversion of unbounded designated types

Note that the conversions might not work in all cases. For instance, when the designated type — indicated by the formal Object parameter of the generic Address_To_Access_Conversions package — is unbounded, the result of a call to To_Pointer may not have bounds.

Let’s adapt the previous code example and replace the Integer type by the (unbounded) String type:

Listing 187: show_address_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System; use System;
with System.Address_To_Access_Conversions;
with System.Address_Image;

procedure Show_Address_Conversion is

package String_AAC is
  new System.Address_To_Access_Conversions
    (Object => String);
  use String_AAC;

  subtype Integer_Access is
    String_AAC.Object_Pointer;

  S : aliased String := "Hello";
  AI_1, AI_2 : Integer_Access;
  A : Address;

begin
  AI_1 := S'Access;
  A := To_Address (AI_1);
  AI_2 := To_Pointer (A);
  -- ^^^^^^^^^^^^^^  
  -- WARNING: Result might not have bounds
  Put_Line ("AI_1.all'Address : 
    & System.Address_Image
    (AI_1.all'Address));
  Put_Line ("AI_2.all'Address : 
    & System.Address_Image
    (AI_2.all'Address));

  if AI_1 = AI_2 then
    Put_Line ("AI_1 = AI_2");
  else
    Put_Line ("AI_1 /= AI_2");
  end if;
  Put_Line ("AI_1: " & AI_1.all);
  Put_Line ("AI_2: " & AI_2.all);
  -- ^^^^^^^^^^^
  -- WARNING: As AI_2 might not have bounds
  -- due to the call to To_Pointer
  -- the behavior of this call to
  -- the "&" operator is
  -- unpredictable.
end Show_Address_Conversion;
```

Code block metadata
In this case, the call to To_Pointer (A) might not have bounds, so any operation on AI_2 might lead to unpredictable results.

In the Ada Reference Manual

- 13.7.2 The Package System.Address_To_Access_Conversions

28.2 Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---

28.2. Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```ada
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```ada
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration A : Integer_Access becomes A : access Integer. Here, access Integer is the anonymous access type declaration, and A is an access object of this anonymous type.

To be more precise, A : access Integer is an access parameter (page 877) and it's specifying an anonymous access-to-object type (page 858). Another flavor of anonymous access types are anonymous access-to-subprograms (page 901). We discuss all these topics in more details later.

---
Let's see a complete example:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Anonymous_Access_Types is
  I_Var : aliased Integer;
  A : access Integer;
begin
  A := I_Var'Access;
  A.all := 22;
  Put_Line ("A.all: " & Integer'Image (A.all));
end Show_Anonymous_Access_Types;
```

Here, A is an access object whose value is initialized with the access to I_Var. Because the declaration of A includes the declaration of an anonymous access type, we don't declare an extra Integer_Access type, as we did in previous code examples.

In the Ada Reference Manual

- 3.10 Access Types

Relation to named types

Anonymous access types were not part of the first version of the Ada standard, which only had support for named access types. They were introduced later to cover some use-cases that were difficult — or even impossible — with access types.

In this sense, anonymous access types aren't just access types without names. Certain accessibility rules for anonymous access types are a bit less strict. In those cases, it might be interesting to consider using them instead of named access types.

In general, however, we should only use anonymous access types in those specific cases where using named access types becomes too cumbersome. As a general recommendation, we should give preference to named access types whenever possible. (Anonymous access-to-object types have drawbacks that we discuss later (page 860).)

Benefits of anonymous access types

One of the main benefits of anonymous access types is their flexibility: since there isn't an explicit access type declaration associated with them, we only have to worry about the subtype $S$ we intend to access.

Also, as long as the subtype $S$ in a declaration `access S` is always the same, no conversion is needed between two access objects of that anonymous type, and the $S'$ attribute always works.

Let's see an example:

Listing 189: show.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show (Name : String; V : access Integer) is
begin
  Put_Line (Name & ".all: 
    & Integer'Image (V.all));
end Show;
```

Listing 190: show_anonymous_access_types.adb

```ada
with Show;

procedure Show_Anonymous_Access_Types is
  I Var : aliased Integer;
  A : access Integer;
  B : access Integer;
begin
  A := I Var'Access;
  B := A;
  A.all := 22;
  Show ("A", A);
  Show ("B", B);
end Show_Anonymous_Access_Types;
```

In this example, we have two access objects $A$ and $B$. Since they're objects of anonymous access types that refer to the same subtype `Integer`, we can assign $A$ to $B$ without a type conversion, and pass those access objects as an argument to the `Show` procedure.

(Note that the use of an access parameter in the `Show` procedure is for demonstration purpose only: a simply `Integer` as the type of this input parameter would have been more than sufficient to implement the procedure. Actually, in this case, avoiding the access parameter would be the recommended approach in terms of clean Ada software design.)

In contrast, if we had used named type declarations, the code would be more complicated and more limited:

28.2. Anonymous Access Types
Learning Ada

Listing 191: aux.ads

```ada
package Aux is

    type Integer_Access is access all Integer;

    procedure Show (Name : String;
                     V : Integer_Access);

end Aux;
```

Listing 192: aux.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Aux is

    procedure Show (Name : String;
                    V : Integer_Access) is

    begin
        Put_Line (Name & ".all: 
                  & Integer'Image (V.all));
    end Show;

end Aux;
```

Listing 193: show_anonymous_access_types.adb

```ada
with Aux; use Aux;

procedure Show_Anonymous_Access_Types is

    -- I_Var : aliased Integer;
    A : Integer_Access;
    B : Integer_Access;

    begin
        -- A := I_Var'Access;
        -- ^ ERROR: non-local pointer cannot
        --     point to local object.

        A := new Integer;
        B := A;

        A.all := 22;
        Show ("A", A);
        Show ("B", B);

end Show_Anonymous_Access_Types;
```

Code block metadata

MD5: 681c2cf7f5e8d520490cc5594484ce69

Runtime output

```
A.all: 22
B.all: 22
```

Here, apart from the access type declaration (Integer_Access), we had to make two adap-
tations to convert the previous code example:

1. We had to move the Show procedure to a package (which we simply called Aux) because of the access type declaration.

2. Also, we had to allocate an object for A instead of retrieving the access attribute of I_Var because we cannot use a pointer to a local object in the assignment to a non-local pointer, as indicate in the comments.

This restriction regarding non-local pointer assignments is an example of the stricter accessibility rules that apply to named access types. As mentioned earlier, the \S\'Access attribute always works when we use anonymous access types — this is not always the case for named access types.

**Important**

As mentioned earlier, if we want to use two access objects in an operation, the rule says that the subtype S of the anonymous type used in their corresponding declaration must match. In the following example, we can see how this rule works:

Listing 194: show_anonymous_access_subtype_error.adb

```ada
procedure Show_Anonymous_Access_Subtype_Error is
  subtype Integer_1_10 is Integer range 1 .. 10;
  I_Var : aliased Integer;
  A : access Integer := I_Var'Access;
  B : access Integer_1_10;
begin
  A := I_Var'Access;
  B := A;  
  -- ^ ERROR: subtype doesn't match!
  B := I_Var'Access;
  -- ^ ERROR: subtype doesn't match!
end Show_Anonymous_Access_Subtype_Error;
```

Even though Integer_1_10 is a subtype of \texttt{Integer}, we cannot assign A to B because the subtype that their access type declarations refer to — \texttt{Integer} and Integer_1_10, respectively — doesn't match. The same issue occurs when retrieving the access attribute of I_Var in the assignment to B.

The later sections on anonymous access-to-object type (page 858) and anonymous access-to-subprograms (page 901) cover more specific details on anonymous access types.
28.2.2 Anonymous Access-To-Object Types

In the previous chapter (page 735), we introduced named access-to-object types and used those types throughout the chapter. Also, in the previous section (page 853), we've seen some simple examples of anonymous access-to-object types:

```ada
procedure Add_One (A : access Integer);
-- ^ Anonymous access type
A : access Integer;
-- ^ Anonymous access type
```

In addition to parameters and objects, we can use anonymous access types in discriminants, components of array and record types, renamings and function return types. (We discuss anonymous access discriminants (page 868) and anonymous access parameters (page 877) later on.) Let's see a code example that includes all these cases:

Listing 195: all_anonymous_access_to_object_types.ads

```ada
package All_Anonymous_Access_To_Object_Types is

procedure Add_One (A : access Integer) is null;
-- ^ Anonymous access type

AI : access Integer;
-- ^ Anonymous access type

type Rec (AI : access Integer) is private;
-- ^ Anonymous access type

type Access_Array is
array (Positive range <>) of
access Integer;
-- ^ Anonymous access type

Arr : array (1 .. 5) of access Integer;
-- ^ Anonymous access type

AI_Renaming : access Integer renames AI;
-- ^ Anonymous access type

function Init_Access_Integer
return access Integer is (null);
-- ^ Anonymous access type

private

end All_Anonymous_Access_To_Object_Types;
```

Code block metadata

MD5: 6533b22a4e4526702320cb327bf6f69a
In this example, we see multiple examples of anonymous access-to-object types:

- as the A parameter of the Add_One procedure;
- in the declaration of the AI access object;
- as the AI discriminant of the Rec type;
- as the component type of the Access_Array type;
- as the component type of the Arr array;
- in the AI_Renaming renaming;
- as the return type of the Init_Access_Integer;
- as the Internal_AI of component of the Rec type.

In the Ada Reference Manual

- 3.10 Access Types

Not Null Anonymous Access-To-Object Types

As expected, null is a valid value for an anonymous access type. However, we can forbid null as a valid value by using not null in the anonymous access type declaration. For example:

Listing 196: all_anonymous_access_to_object_types.ads

```
package All_Anonymous_Access_To_Object_Types is

  procedure Add_One (A : not null access Integer)
  is null;  -- ^ Anonymous access type

  I : aliased Integer;

  AI : not null access Integer := I'Access;
  -- ^ Anonymous access type  ^^^^^^^
  -- Initialization required!

  type Rec (AI : not null access Integer) is
  private;
  -- ^ Anonymous access type

type Access_Array is
  array (Positive range <>) of
  not null access Integer;
  -- ^ Anonymous access type

  Arr : array (1 .. 5) of
  not null access Integer :=
  -- ^ Anonymous access type
  (others => I'Access);
  -- ^^^^^^^^^^^^^^^^^^^^^
  -- Initialization required!

  AI_Renaming : not null access Integer

(continues on next page)
```

---

28.2. Anonymous Access Types

---

renames AI; -- Anonymous access type

function Init_Access_Integer return not null access Integer is (I'Access);
-- ^ Anonymous access type
-- Initialization required!

private

type Rec (AI : not null access Integer) is
record
-- ^ Anonymous access type
Internal_AI : not null access Integer;
-- ^ Anonymous access type
end record;

end All_Anonymous_Access_To_Object_Types;

As you might have noticed, we took the previous code example and used not null for each usage instance of the anonymous access type. In this sense, this version of the code example is very similar to the previous one. Note, however, that we now have to explicitly initialize some elements to avoid the Constraint_Error exception being raised at runtime. This is the case for example for the AI access object:

AI : not null access Integer := I'Access;

If we hadn't initialized AI explicitly with I'Access, it would have been set to null, which would fail the not null constraint of the anonymous access type. Similarly, we also have to initialize the Arr array and return a valid access object for the Init_Access_Integer function.

Drawbacks of Anonymous Access-To-Object Types

Anonymous access-to-object types have important drawbacks. For example, some features that are available for named access types aren't available for the anonymous access types. Also, most of the drawbacks are related to how anonymous access-to-object types can potentially make the allocation and deallocation quite complicated or even error-prone.

For starters, some pool-related features aren't available for anonymous access-to-object types. For example, we cannot specify which pool is going to be used in the allocation of an anonymous access-to-object. In fact, the memory pool selection is compiler-dependent, so we cannot rely on an object being allocated from a specific pool when using new with an anonymous access-to-object type. (In contrast, as we know, each named access type has an associated pool, so objects allocated via new will be allocated from that pool.) Also, we cannot identify which pool was selected for the allocation of a specific object, so we don't have any information to use for the deallocation of that object.

Because the pool selection is hidden from us, this makes the memory deallocation more complicated. For example, we cannot instantiate the Ada.Unchecked_Deallocation pro-
procedure for anonymous access types. Also, some of the methods we could use to circumvent this limitation are error-prone, as we discuss in this section.

Also, storage-related features aren’t available: specifying the storage size — especially, specifying that the access type has a storage size of zero — isn’t possible.

**Missing features**

Let’s see a code example that shows some of the features that aren’t available for anonymous access-to-object types:

```
with Ada.Unchecked_Deallocation;

package Missing_Features is

-- We cannot specify which pool will be used
-- in the anonymous access-to-object
-- allocation; the pool is selected by the
-- compiler:
IA : access Integer := new Integer;

-- All the features below aren’t available
-- for an anonymous access-to-object:
--

-- Having a specific storage pool associated
-- with the access type:

type String_Access is
  access String;
-- Automatically creates
-- String_Access’Storage_Pool

type Integer_Access is
  access Integer
  with Storage_Pool =>
  String_Access’Storage_Pool;
-- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
-- Using the pool from another
-- access type.

-- Specifying a deallocation function for the
-- access type:
procedure Free is
  new Ada.Unchecked_Deallocation
    (Object => Integer,
    Name => Integer_Access);

-- Specifying a limited storage size for
-- the access type:

type Integer_Access_Store_128 is
  access Integer
  with Storage_Size => 128;
-- Limiting the storage size for the
-- access type to zero:

type Integer_Access_Store_0 is
  access Integer
  with Storage_Size => 0;
```

(continues on next page)
In the Missing.Features package, we see some of the features that we cannot use for the anonymous access Integer type, but that are available for equivalent named access types:

- There's no specific memory pool associated with the access object IA. In contrast, named types — such as String.Access and Integer.Access — have an associated pool, and we can use the Storage.Pool aspect and the Storage.Pool attribute to customize them.

- We cannot instantiate the Ada.Unchecked_Deallocation procedure for the access Integer type. However, we can instantiate it for named access types such as the Integer.Access type.

- We cannot use the Storage_Size attribute for the access Integer type, but we're allowed to use it with named access types, which we do in the declaration of the Integer.Access_Store_128 and Integer.Access_Store_0 types.

### Dangerous memory deallocation

We might think that we could make up for the absence of the Ada.Unchecked_Deallocation procedure for anonymous access-to-object types by converting those access objects (of anonymous access types) to a named type that has the same designated subtype. For example, if we have an access object IA of an anonymous access Integer type, we can convert it to the named Integer.Access type, provided this named access type is compatible with the anonymous access type, e.g.:

```ada
type Integer_Access is access all Integer
```

Let's see a complete code example:

```
with Ada.Unchecked_Deallocation;

procedure Show_Dangerous_Deallocation is
  type Integer_Access is
    access all Integer;
  procedure Free is
    new Ada.Unchecked_Deallocation
      (Object => Integer,
       Name => Integer_Access);
  IA : access Integer;
begin
  IA := new Integer;
  IA.all := 30;
  -- Potentially erroneous deallocation via type
  -- conversion:
```

(continues on next page)
This example declares the IA access object of the anonymous `access Integer` type. After allocating an object for IA via `new`, we try to deallocate it by first converting it to the `Integer_Access` type, so that we can call the `Free` procedure to actually deallocate the object. Although this code compiles, it’ll only work if both `access Integer` and `Integer_Access` types are using the same memory pool. Since we cannot really determine this, the result is potentially erroneous: it’ll work if the compiler selected the same pool, but it’ll fail otherwise.

**Important**

Because allocating memory for anonymous access types is potentially dangerous, we can use the `No_Anonymous_Allocators` restriction — which is available since Ada 2012 — to prevent this kind of memory allocation being used in the code. For example:

```
pragma Restrictions (No_Anonymous_Allocators);

procedure Show_Dangerous_Allocation is
  IA : access Integer;
begin
  IA := new Integer;
  IA.all := 30;
end Show_Dangerous_Allocation;
```

**Build output**

`show_dangerous_allocation.adb:6:10: error: violation of restriction "No_Anonymous_Allocators" at line 1`
**Possible solution using named access types**

A better solution to avoid issues when allocating and deallocating memory for anonymous access-to-object types is to allocate the object using a known pool. As mentioned before, the memory pool associated with a named access type is well-defined, so we can use this kind of types for memory allocation. In fact, we can use a named memory type to allocate an object via `new`, and then associate this allocated object with the access object of anonymous access type.

Let's see a code example:

```ada
Listing 200: show_successful_deallocation.adb

with Ada.Unchecked_Deallocation;
procedure Show_Successful_Deallocation is
   type Integer_Access is
      access Integer;
   procedure Free is
      new Ada.Unchecked_Deallocation
         (Object => Integer,
          Name => Integer_Access);
   IA    : access Integer;
   Typed_IA : Integer_Access;
begin
   Typed_IA := new Integer;
   IA := Typed_IA;
   IA.all := 30;
   -- Deallocation of the access object that has
   -- an associated type:
   Free (Typed_IA);
end Show_Successful_Deallocation;
```

In this example, all operations related to memory allocation are exclusively making use of the `Integer_Access` type, which is a named access type. In fact, `new Integer` allocates the object from the pool associated with the `Integer_Access` type, and the call to `Free` deallocates this object back into that pool. Therefore, associating this object with the IA access object — in the `IA := Typed_IA` assignment — doesn't create problems afterwards in the object's deallocation. (When calling `Free`, we only refer to the object of named access type, so the object is deallocated from a known pool.)

Of course, a potential issue here is that `IA` becomes a *dangling reference* (page 795) after the call to `Free`. Therefore, we can improve this solution by completely hiding the memory allocation and deallocation for the anonymous access types in subprograms — e.g. as part of a package. By doing so, we don't expose the named access type, thereby reducing the possibility of dangling references.

In fact, we can generalize this approach with the following (generic) package:
In the generic `Hidden_Anonymous_Allocation` package, `New_T` allocates a new object internally and returns an anonymous access to this object. The `Free` procedure deallocates this object.

In the body of the `Hidden_Anonymous_Allocation` package, we use the named access type `T_Access` to handle the actual memory allocation and deallocation. As expected, because those operations happen on the pool associated with the `T_Access` type, we don't have to worry about potential deallocation issues.

Finally, we can instantiate this package for the type we want to have anonymous access types for, say a type named `Rec`. Then, when using the `Rec` type in the main subprogram, we can simply call the corresponding subprograms for memory allocation and deallocation.
Learning Ada

For example:

Listing 203: info.ads

```ada
with Hidden_Anonymous_Allocation;

package Info is
  type Rec is private;
  function New_Rec return not null access Rec;
  procedure Free (Obj : access Rec);

private
  type Rec is record
    I : Integer;
  end record;

  package Rec_Allocation is new
    Hidden_Anonymous_Allocation (T => Rec);
  function New_Rec return not null access Rec
    renames Rec_Allocation.New_T;
  procedure Free (Obj : access Rec)
    renames Rec_Allocation.Free;
end Info;
```

Listing 204: show_info_allocation_deallocation.adb

```ada
with Info; use Info;

procedure Show_Info_Allocation_Deallocation is
  RA : constant not null access Rec := New_Rec;
begin
  Free (RA);
end Show_Info_Allocation_Deallocation;
```

Code block metadata


MD5: d71e8ed70e280c6d5d9fc2d49c1eb6c3

In this example, we instantiate the Hidden_Anonymous_Allocation package in the Info package, which also defines the Rec type. We associate the New_T and Free subprograms with the Rec type by using subprogram renaming. Finally, in the Show_Info_Allocation_Deallocation procedure, we use these subprograms to allocate and deallocate the type.
Possible solution using the stack

Another approach that we could consider to avoid memory deallocation issues for anonymous access-to-object types is by simply using the stack for the object creation. For example:

```
procedure Show_Automatic_Deallocation is
   I : aliased Integer;
   -- ^ Allocating object on the stack
   IA : access Integer;
begin
   IA := I'Access;
   -- Indirect allocation:
   -- object creation on the stack.
   IA.all := 30;
   -- Automatic deallocation at the end of the
   -- procedure because the integer variable is
   -- on the stack.
end Show_Automatic_Deallocation;
```

When to use anonymous access-to-objects types

In summary, anonymous access-to-object types have many drawbacks that often outweigh their benefits (page 855). In fact, allocation for those types can quickly become very complicated. Therefore, in general, they're not a good alternative to named access types. Indeed, the difficulties that we've just seen might make them a much worse option than just using named access types instead.

We might consider using anonymous access-to-objects types only in cases when we reach a point in our implementation work where using named access types becomes impossible — or when using them becomes even more complicated than equivalent solutions using anonymous access types. This scenario, however, is usually the exception rather than the rule. Thus, as a general guideline, we should always aim to use named access types.

That being said, an important exception to this advice is when we're interfacing to other languages (page 880). In this case, as we'll discuss later, using anonymous access-to-objects types can be significantly simpler (compared to named access types) without the drawbacks that we've just discussed.
28.2.3 Access discriminants

Previously, we've discussed *discriminants as access values* (page 745). In that section, we only used named access types. Now, in this section, we see how to use anonymous access types as discriminants. This feature is also known as *access discriminants* and it provides some flexibility that can be interesting in terms of software design, as we'll discuss later. Let's start with an example:

Listing 206: custom_recs.ads

```ada
package Custom_Recs is
  -- Declaring a discriminant with an anonymous
  -- access type:
  type Rec (IA : access Integer) is record
    I : Integer := IA.all;
  end record;
  procedure Show (R : Rec);
end Custom_Recs;
```

Listing 207: custom_recs.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Custom_Recs is
  procedure Show (R : Rec) is
    begin
      Put_Line ("R.IA = ";
        & Integer'Image (R.IA.all));
      Put_Line ("R.I = ";
        & Integer'Image (R.I));
    end Show;
end Custom_Recs;
```

Listing 208: show_access_discriminants.adb

```ada
with Custom_Recs; use Custom_Recs;
procedure Show_Access_Discriminants is
  I : aliased Integer := 10;
  R : Rec (I'Access);
begin
  Show (R);
  I := 20;
  R.I := 30;
  Show (R);
end Show_Access_Discriminants;
```

Code block metadata

MD5: f8e127fda4f7ea0f1593165d6a966df6

Runtime output
In this example, we use an anonymous access type for the discriminant in the declaration of the `Rec` type of the `Custom_Recs` package. In the `Show_Access_Discriminants` procedure, we declare `R` and provide access to the local `I` integer.

Similarly, we can use unconstrained designated subtypes:

```ada
package Persons is
  -- Declaring a discriminant with an anonymous access type whose designated subtype is unconstrained:
  type Person (Name : access String) is record
    Age : Integer;
  end record;
  procedure Show (P : Person);
end Persons;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Persons is
  procedure Show (P : Person) is
    begin
      Put_Line ("Name = " & P.Name.all);
      Put_Line ("Age = " & Integer'image (P.Age));
    end Show;
end Persons;
```

```ada
with Persons; use Persons;
procedure Show_Person is
  S : aliased String := "John";
  P : Person (S'Access);
  begin
    P.Age := 30;
    Show (P);
end Show_Person;
```

Code block metadata

- MD5: f0149d572e0ec192476836bfdf00dd9e

Runtime output
In this example, for the discriminant of the `Person` type, we use an anonymous access type whose designated subtype is unconstrained. In the `Show_Person` procedure, we declare the `P` object and provide access to the `S` string.

---

**In the Ada Reference Manual**

- [3.7 Discriminants](http://www.ada-auth.org/standards/22rm/html/RM-3-7.html)
- [3.10.2 Operations of Access Types](http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html)

---

**Default Value of Access Discriminants**

In contrast to named access types, we cannot use a default value for the access discriminant of a non-limited type:

```ada
package Custom_Recs is
    -- Declaring a discriminant with an anonymous
    -- access type and a default value:
    type Rec (IA : access Integer := new Integer' (0)) is
        record
            I : Integer := IA.all;
        end record;

    procedure Show (R : Rec);
end Custom_Recs;
```

**Code block metadata**


*MD5: 9269cea113f29443a6d7bb719d0616f1*

**Build output**

`custom_recs.ads:6:21: error: (Ada 2005) access discriminants of nonlimited types cannot have defaults`

```
gprbuild: *** compilation phase failed
```

However, if we change the type declaration to be a limited type, having a default value for the access discriminant is OK:

```ada
package Custom_Recs is
    -- Declaring a discriminant with an anonymous
    -- access type and a default value:
    type Rec (IA : access Integer :=
```

---

[207](http://www.ada-auth.org/standards/22rm/html/RM-3-7.html)

[208](http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html)
**Learning Ada**

(continued from previous page)

```ada
new Integer'(0)) is limited

record
  I : Integer := IA.all;
end record;

procedure Show (R : Rec);
end Custom_Recs;
```

**Code block metadata**


MD5: 9e8683c7a27e9097fd2003ad91bac269

**Build output**

```
With Custom_Recs; use Custom_Recs;

procedure Show_Access_Discriminants is
  R : Rec;
  -- ^^^
  -- This triggers "new Integer'(0)", so an
  -- integer object is allocated and stored in
  -- the R.IA discriminant.
begin
  Show (R);
  -- R gets out of scope here, and the object
  -- allocated via new hasn't been deallocated.
  Show_Access_Discriminants;
end Show_Access_Discriminants;
```

**Code block metadata**


MD5: f5d9dee26044ccab2193ab419638de79

**Build output**

```
with Custom_Recs;
with Custom_Recs;
use Custom_Recs;

procedure Show_Access_Discriminants is
  R : Rec;
  -- ^^^
  -- This triggers "new Integer'(0)", so an
  -- integer object is allocated and stored in
  -- the R.IA discriminant.
begin
  Show (R);
  -- R gets out of scope here, and the object
  -- allocated via new hasn't been deallocated.
  Show_Access_Discriminants;
end Show_Access_Discriminants;
```

**Build output**

```
show_access_discriminants.adb:4:04: warning: coextension will not be deallocated when its
  associated owner is deallocated [enabled by default]
custom_recs.ads:6:21: warning: coextension will not be deallocated when its
  associated owner is deallocated [enabled by default]
```

**Runtime output**

```
R.IA = 0
R.I = 0
```

In this case, the allocated object won't be deallocated when R gets out of scope!

28.2. Anonymous Access Types 871
Benefits of Access Discriminants

Access discriminants have the same benefits that we’ve already seen earlier while discussing discriminants as access values (page 745). An additional benefit is its extended flexibility: access discriminants are compatible with any access T’Access, as long as T is of the designated subtype.

Consider the following example using the named access type Access_String:

```ada
package Persons is
  type Access_String is access all String;
  -- Declaring a discriminant with a named access type:
  type Person (Name : Access_String) is record
    Age : Integer;
  end record;

  procedure Show (P : Person);
end Persons;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Persons is
  procedure Show (P : Person) is
    begin
      Put_Line ("Name = ", P.Name.all);
      Put_Line ("Age = ", Integer'Image (P.Age));
    end Show;
end Persons;
```

```ada
with Persons; use Persons;

procedure Show_Person is
  S : aliased String := "John";
  P : Person (S'Access);
  -- ^^^^^^^^^ ERROR: cannot use local object
  -- We can, however, allocate the string via new:
  -- new:
  --
  -- S : Access_String := new String("John");
  -- P : Person (S);
  begin
    P.Age := 30;
    Show (P);
  end Show_Person;
```

Code block metadata
This code doesn't compile because we cannot have a non-local pointer (Access_String) pointing to the local object S. The only way to make this work is by allocating the string via new (i.e.: $S : \text{Access\_String} := \text{new String}$).

However, if we use an access discriminant in the declaration of Person, the code compiles fine:

```ada
package Persons is

  -- Declaring a discriminant with an anonymous access type:
  type Person (Name : access String) is record
    Age : Integer;
  end record;

  procedure Show (P : Person);
end Persons;
```

```ada
with Persons; use Persons;

procedure Show_Person is
  S : aliased String := "John";
  P : Person (S'Access);
  -- ^^^^^^^^ OK

  -- Allocating the string via new and using it in P's declaration is OK as well, but we should manually deallocate it before S gets out of scope:
  --
  -- S : access String := new String'("John");
  -- P : Person (S);
begin
  P.Age := 30;
  Show (P);
end Show_Person;
```

**Code block metadata**


MD5: e918db3790c7ffebeb7c0f54ced9f48b9

**Runtime output**

Name = John
Age = 30
In this case, getting access to the local object S and using it for P's discriminant is perfectly fine.

**Preventing dangling pointers**

Note that the usual rules that prevent dangling pointers still apply here. This ensures that we can safely use access discriminants. For example:

```
with Persons; use Persons;

procedure Show_Person is

  function Local_Init return Person is
    S : aliased String := "John";
  begin
    return (Name => S'Access, Age => 30);
  end

begin
  P : Person := Local_Init;
  Show (P);
end Show_Person;
```

In this example, compilation fails in the Local_Init function when trying to return an object of Person type because S'Access would be a dangling reference.

**28.2.4 Self-reference**

Previously, we've seen that we can declare *self-references* (page 762) using named access types. We can do the same with anonymous access types. Let's revisit the code example that implements linked lists:

```
generic

  type T is private;
package Linked_Lists is

  type List is limited private;

  procedure Append_Front
    (L : in out List;
     E : T);

```

(continues on next page)
procedure Append_Rear
  (L : in out List;
   E : T);

procedure Show (L : List);

private

type Component is record
  Next : access Component;
  -- ^^^^^^^^^^^^^^^^ Self-reference
  -- (Note that we haven't finished the
declaraton of the "Component" type
  -- yet, but we're already referring to
  -- it.)
  Value : T;
end record;

type List is access all Component;

end Linked_Lists;

Listing 222: linked_lists.adb

pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;

package body Linked_Lists is

  procedure Append_Front
  (L : in out List;
   E : T)
  is
    New_First : constant List := new
      Component'(Value => E,
                   Next => L);
begin
  L := New_First;
end Append_Front;

  procedure Append_Rear
  (L : in out List;
   E : T)
  is
    New_Last : constant List := new
      Component'(Value => E,
                 Next => null);
begin
  if L = null then
    L := New_Last;
  else
    declare
      Last : List := L;
    begin
      while Last.Next /= null loop
        Last := List (Last.Next);
      end loop;
    end declare;
  end if;
  L := Last.Next;
end Append_Rear;

end Linked_Lists;

(continues on next page)
34 -- ^^^^  
35 -- type conversion:  
36 -- "access Component" to  
37 -- "List"  
38 end loop;  
39 Last.Next := New_Last;  
40 end;  
41 end Append_Rear;  
42  
43 procedure Show (L : List) is  
44 Curr : List := L;  
45 begin  
46 if L = null then  
47 Put_Line ("[ ]");  
48 else  
49 Put ("[");  
50 loop  
51 Put (Curr.Value'Image);  
52 Put (" ");  
53 exit when Curr.Next = null;  
54 Curr := Curr.Next;  
55 end loop;  
56 Put_Line ("]");  
57 end if;  
58 end Show;  
59 end Linked_Lists;  

Listing 223: test_linked_list.adb

1 with Linked_Lists;  
2 procedure Test_Linked_List is  
3 package Integer_Lists is new  
4 Linked_Lists (T => Integer);  
5 use Integer_Lists;  
6 begin  
7 L : List;  
8 Append_Front (L, 3);  
9 Append_Rear (L, 4);  
10 Append_Rear (L, 5);  
11 Append_Front (L, 2);  
12 Append_Front (L, 1);  
13 Append_Rear (L, 6);  
14 Append_Rear (L, 7);  
15 Show (L);  
16 end Test_Linked_List;  

Code block metadata

MD5: 9e42bf9fa630a0af8d6f7c85a1565ed6  

Runtime output  

[ 1 2 3 4 5 6 7 ]
Here, in the declaration of the Component type (in the private part of the generic Linked_Lists package), we declare Next as an anonymous access type that refers to the Component type. (Note that at this point, we haven't finished the declaration of the Component type yet, but we're already using it as the designated subtype of an anonymous access type.) Then, we declare List as a general access type (with Component as the designated subtype).

It's worth mentioning that the List type and the anonymous access Component type aren't the same type, although they share the same designated subtype. Therefore, in the implementation of the Append_Rear procedure, we have to use type conversion to convert from the anonymous access Component type to the (named) List type.

28.2.5 Mutually dependent types using anonymous access types

In the section on mutually dependent types using access types (page 765), we've seen a code example that was using named access types. We could now rewrite it using anonymous access types:

```
package Mutually_Dependent is
  type T2;
  type T1 is record
    B : access T2;
  end record;
  type T2 is record
    A : access T1;
  end record;
end Mutually_Dependent;
```

In this example, T1 and T2 are mutually dependent types. We're using anonymous access types in the declaration of the B and A components.

28.2.6 Access parameters

In the previous chapter, we talked about parameters as access values (page 752). As you might have expected, we can also use anonymous access types as parameters of a subprogram. However, they're limited to be in parameters of a subprogram or return type of a function (also called the access result type):

```
package Names is
  function Init (S1, S2 : String)
    return access String;
  -- ^^^^^^^^^^^^^^^^^^^^
  -- Anonymous access type as the access
end Names;
```

(continues on next page)
In this example, we have a string as the access result type of the Init function, and another string as the access parameter of the Show procedure.

This is the complete code example:

Listing 226: names.ads

```ada
package Names is
    function Init (S1, S2 : String) return access String;
    procedure Show (N : access constant String);
private
    function Init (S1, S2 : String) return access String is
        (new String'(S1 & "-" & S2));
end Names;
```

Listing 227: names.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Names is
    procedure Show (N : access constant String) is
        begin
            Put_Line ("Name: " & N.all);
            end Show;
end Names;
```

Listing 228: show_names.adb

```ada
with Names; use Names;
procedure Show_Names is
    N : access String := Init ("Lily", "Ann");
    begin
        Show (N);
    end Show_Names;
```

Code block metadata

MD5: 622a76c4b133ed2715f18c175694cbe2

In this example, we have a string as the access result type of the Init function, and another string as the access parameter of the Show procedure.

This is the complete code example:
Runtime output

Name: Lily-Ann

Note that we’re not using the in parameter mode in the Show procedure above. Usually, this parameter mode can be omitted, as it is the default parameter mode — procedure P (I : Integer) is the same as procedure P (I : in Integer). However, in the case of the Show procedure, the in parameter mode isn’t just optionally absent. In fact, for access parameters, the parameter mode is always implied as in, so writing it explicitly is actually forbidden. In other words, we can only write N : access String or N : access constant String, but we cannot write N : in access String or N : in access constant String.

For further reading...

When we discussed parameters as access values (page 752) in the previous chapter, we saw how we can simply use different parameter modes to write a program instead of using access types. Basically, to implement the same functionality, we just replaced the access types by selecting the correct parameter modes instead and used simpler data types.

Let’s do the same exercise again, this time by adapting the previous code example with anonymous access types:

Listing 229: names.ads

```ada
package Names is

  function Init (S1, S2 : String)
    return String;

  procedure Show (N : String);

private

  function Init (S1, S2 : String)
    return String is
    (S1 & "-" & S2);

end Names;
```

Listing 230: names.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Names is

  procedure Show (N : String) is
  begin
    Put_Line ("Name: " & N);
  end Show;

end Names;
```

Listing 231: show_names.adb

```ada
with Names; use Names;
```

(continues on next page)
Learning Ada

(procedure) Show_Names is
  N : String := Init ("Lily", "Ann");
begin
  Show (N);
end Show_Names;

Code block metadata
MD5: 643f193999ef8de9bcefb11d9b2d21d7

Runtime output
Name: Lily-Ann

Although we’re using simple strings instead of access types in this version of the code example, we’re still getting a similar behavior. However, there is a small, yet important difference in the way the string returned by Init is being allocated: while the previous implementation (which was using an access result type) was allocating the string on the heap, we’re now allocating the string on the stack.

Later on, we talk about the accessibility rules in the case of access parameters (page 900). In general, we should avoid access parameters whenever possible and simply use objects and parameter modes directly, as it makes the design simpler and less error-prone. One exception is when we're interfacing to other languages, especially C: this is our next topic (page 880). Another time when access parameters are vital is for inherited primitive operations for tagged types. We discuss this later on (page 884).

In the Ada Reference Manual
• 3.10 Access Types

Interfacing To Other Languages

We can use access parameters to interface to other languages. This can be particularly useful when interfacing to C code that makes use of pointers. For example, let’s assume we want to call the add_one function below in our Ada implementation:

Listing 232: operations_c.h

```ada
void add_one(int *p_i);
```

Listing 233: operations_c.c

```c
void add_one(int *p_i)
{
  *p_i = *p_i + 1;
}
```

Code block metadata

We could map the `int *` parameter of `add_one` to `access Integer` in the Ada specification:

```ada
procedure Add_One (IA : access Integer)
  with Import, Convention => C;
```

This is a complete code example:

```
package Operations is

  procedure Add_One (IA : access Integer)
    with Import, Convention => C;

end Operations;
```

```
with Ada.Text_IO; use Ada.Text_IO;

with Operations; use Operations;

procedure Show_Operations is
  I : aliased Integer := 42;
begin
  Put_Line (I'Image);
  Add_One (I'Access);
  Put_Line (I'Image);
end Show_Operations;
```

Once again, we can replace access parameters with simpler types by using the appropriate parameter mode. In this case, we could replace `access Integer` by `aliased in out Integer`. This is the modified version of the code:

```
package Operations is

  procedure Add_One
    (IA : aliased in out Integer)
    with Import, Convention => C;

end Operations;
```

```
with Ada.Text_IO; use Ada.Text_IO;

with Operations; use Operations;

procedure Show_Operations is
  (continues on next page)
```

---

### 28.2. Anonymous Access Types

---
Learning Ada

(continued from previous page)

```ada
I : aliased Integer := 42;
begin
  Put_Line (I'Image);
  Add_One (I);
  Put_Line (I'Image);
end Show_Operations;
```

Code block metadata

MD5: 2c5a81b8d77f0fff8a73f7912be6b6fe

However, there are situations where aliased objects cannot be used. For example, suppose we want to allocate memory inside a C function. In this case, the pointer to that memory block must be mapped to an access type in Ada.

Let's extend the previous C code example and introduce the `alloc_integer` and `dealloc_integer` functions, which allocate and deallocate an integer value:

Listing 238: operations_c.h

```c
int * alloc_integer();
void dealloc_integer(int *p_i);
void add_one(int *p_i);
```

Listing 239: operations_c.c

```c
#include <stdlib.h>
int * alloc_integer()
{
  return malloc(sizeof(int));
}
void dealloc_integer(int *p_i)
{
  free (p_i);
}
void add_one(int *p_i)
{
  *p_i = *p_i + 1;
}
```

Code block metadata

MD5: ec6dea12d0a948489cce21b0cc01ad2

In this case, we really have to use access types to interface to these C functions. In fact, we need an access result type to interface to the `alloc_integer()` function, and an access parameter in the case of the `dealloc_integer()` function. This is the corresponding specification in Ada:
package Operations is

  function Alloc_Integer return access Integer
    with Import, Convention => C;

  procedure Dealloc_Integer (IA : access Integer)
    with Import, Convention => C;

  procedure Add_One
    (IA : aliased in out Integer)
    with Import, Convention => C;

end Operations;

In this application, we get a C pointer from the alloc_integer function and encapsulate it in an Ada access type, which we then assign to I. In the last line of the procedure, we call Dealloc_Integer and pass I to it, which deallocates the memory block indicated by the C pointer.

In the Ada Reference Manual

- 3.10 Access Types\textsuperscript{210}

\textsuperscript{210} http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
Inherited Primitive Operations For Tagged Types

In order to declare inherited primitive operations for tagged types that use access types, we need to use access parameters. The reason is that, to be a primitive operation for some tagged type — and hence inheritable — the subprogram must reference the tagged type name directly in the parameter profile. This means that a named access type won’t suffice, because only the access type name would appear in the profile. For example:

Listing 242: inherited_primitives.ads

```ada
package Inherited_Primitives is

  type T is tagged private;
  type T_Access is access all T;
  procedure Proc (N : T_Access);
    -- Proc is not a primitive of type T.

  type T_Child is new T with private;
  type T_Child_Access is access all T_Child;

private

  type T is tagged null record;
  type T_Child is new T with null record;

end Inherited_Primitives;
```

Listing 243: inherited_primitives.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Inherited_Primitives is

  procedure Proc (N : T_Access) is null;

end Inherited_Primitives;
```

Listing 244: show_inherited_primitives.adb

```ada
with Inherited_Primitives;
use Inherited_Primitives;

procedure Show_Inherited_Primitives is
  Obj : T_Access := new T;
  Obj_Child : T_Child_Access := new T_Child;
begin
  Proc (Obj);
  Proc (Obj_Child);
    -- ^^^^^^^^^
    -- ERROR: Proc is not inherited!
end Show_Inherited_Primitives;
```

Code block metadata

MD5: 8235b21caa9f1f105f533d74d891adfe

884 Chapter 28. Resource Management
Build output

show_inherited_primitives.adb:9:10: error: expected type "T_Access" defined at _inherited_primitives.ads:5
show_inherited_primitives.adb:9:10: error: found type "T_Child_Access" defined at _inherited_primitives.ads:12
gprbuild: *** compilation phase failed

In this example, Proc is not a primitive of type T because it's referring to type T_Access, not type T. This means that Proc isn't inherited when we derive the T_Child type. Therefore, when we call Proc (Obj_Child), a compilation error occurs because the compiler expects type T_Access — there's no Proc (N : T_Child_Access) that could be used here.

If we replace T_Access in the Proc procedure with an an access parameter (access T), the subprogram becomes a primitive of T:

Listing 245: inherited_primitives.ads

```ada
package Inherited_Primitives is
  type T is tagged private;
  procedure Proc (N : access T);
  -- Proc is a primitive of type T.
  type T_Child is new T with private;

private
  type T is tagged null record;
  type T_Child is new T with null record;
end Inherited_Primitives;
```

Listing 246: inherited_primitives.adb

```ada
package body Inherited_Primitives is
  procedure Proc (N : access T) is null;
end Inherited_Primitives;
```

Listing 247: show_inherited_primitives.adb

```ada
with Inherited_Primitives;
use Inherited_Primitives;

procedure Show_Inherited_Primitives is
  Obj : access T := new T;
  Obj_Child : access T_Child := new T_Child;
begin
  Proc (Obj);
  Proc (Obj_Child);
  -- ^^^^^^^^^
  -- OK: Proc is inherited!
end Show_Inherited_Primitives;
```

28.2. Anonymous Access Types
Now, the child type `T_Child` (derived from the `T`) inherits the primitive operation `Proc`. This inherited operation has an access parameter designating the child type:

```ada
type T_Child is new T with private;
procedure Proc (N : access T_Child);
-- Implicitly inherited primitive operation
```

---

### In the Ada Reference Manual

- 3.9.2 Dispatching Operations of Tagged Types\(^{211}\)

---

#### 28.2.7 User-Defined References

*Implicit dereferencing* (page 767) isn't limited to the contexts that Ada supports by default: we can also add implicit dereferencing to our own types by using the `Implicit_Dereference` aspect.

To do this, we have to declare:

- a reference type, where we use the `Implicit_Dereference` aspect to specify the reference discriminant, which is the record discriminant that will be dereferenced; and
- a reference object, which contains an access value that will be dereferenced.

Also, for the reference type, we have to:

- specify the reference discriminant as an `access discriminant` (page 868); and
- indicate the name of the reference discriminant when specifying the `Implicit_Dereference` aspect.

Let's see a simple example:

Listing 248: show_user_defined_reference.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_User_Defined_Reference is

  type Id_Number is record
    Id : Positive;
  end record;

  -- Reference type:
  -- Reference type:
  type Id_Ref (Ref : access Id_Number) is
    null record
  with Implicit_Dereference => Ref;
  -- ^ reference discriminant
  -- ^ reference discriminant
  -- name of the reference
  -- discriminant

  --
```

Here, we declare a simple record type (Id_Number) and a corresponding reference type (Id_Ref). Note that:

- the reference discriminant Ref has an access to the Id_Number type; and
- we indicate this reference discriminant in the Implicit_Dereference aspect.

Then, we declare an access value (the I constant) and use it for the Ref discriminant in the declaration of the reference object R.

Finally, we implicitly dereference R and access the Id component by simply writing R.Id — instead of the extended forms R.Ref.Id or R.Ref.all.Id.

**Important**

The extended form mentioned in the example that we just saw (R.Ref.all.Id) makes it clear that two steps happen when evaluating R.Id:

- First, R.Ref is implied from R because of the Implicit_Dereference aspect.
- Then, R.Ref is implicitly dereferenced to R.Ref.all.

After these two steps, we can access the actual object. (In our case, we can access the Id component.)

Note that we cannot use access types directly for the reference discriminant. For example, if we made the following change in the previous code example, it wouldn't compile:

```ada
type Id_Number_Access is access Id_Number;
```

---

**28.2. Anonymous Access Types** 887
-- discriminant!
null record
with Implicit_Dereference => Ref;

However, we could use other forms — such as not null access — in the reference discriminant:

-- Reference type:
type Id_Ref (Ref : not null access Id_Number) is
null record
with Implicit_Dereference => Ref;

In the Ada Reference Manual
• 4.1.5 User-Defined References\textsuperscript{212}

Dereferencing of tagged types

Naturally, implicit dereferencing is also possible when calling primitives of a tagged type. For example, let's change the declaration of the Id_Number type from the previous code example and add a Show primitive.

Listing 249: info.ads

package Info is
  type Id_Number (Id : Positive) is
tagged private;
  procedure Show (R : Id_Number);
private
  type Id_Number (Id : Positive) is
tagged null record;
end Info;

Listing 250: info.adb

with Ada.Text_Io; use Ada.Text_Io;

package body Info is
  procedure Show (R : Id_Number) is
  begin
    Put_Line ("ID: " & Positive'Image (R.Id));
  Show;
end Info;

Then, let's declare a reference type and a reference object in the test application:

\textsuperscript{212} \url{http://www.ada-auth.org/standards/22rm/html/RM-4-1-5.html}
with Info; use Info;

procedure Show_User_Defined_Reference is

-- Reference type:
type Id_Ref (Ref : access Id_Number) is
null record
  with Implicit_Dereference => Ref;

-- Access value:
I : constant access Id_Number :=
  new Id_Number (42);

-- Reference object:
R : Id_Ref (I);

begin
  R.Show;
  -- Equivalent to:
  -- R.Ref.all.Show;
end Show_User_Defined_Reference;

Code block metadata
MD5: 9c5dfc4f2b8e085efde9e61689243f70

Runtime output
ID: 42

Here, we can call the Show procedure by simply writing R.Show instead of R.Ref.all.Show.

Simple container

A typical application of user-defined references is to create cursors when iterating over a container. As an example, let's implement the National_Date_Info package to store the national day of a country:

package National_Date_Info is

  subtype Country_Code is String (1 .. 3);

  type Time is record
    Year : Integer;
    Month : Positive range 1 .. 12;
    Day : Positive range 1 .. 31;
  end record;

  type National_Date is tagged record
    Country : Country_Code;
    Date : Time;
  end record;

(continues on next page)
Here, National_Date is a record type that we use to store the national day information. We can call the Show procedure to display this information.

Now, let's implement the National_Date_Containers with a container for national days:

```ada
with National_Date_Info; use National_Date_Info;

-- Reference type:
type National_Date_Reference
   (Ref : access National_Date)
   is
tagged limited null record
   with Implicit_Dereference => Ref;

-- Container (as an array):
type National_Dates is
   array (Positive range <>) of
      National_Date_Access;

-- The Find function scans the container to
-- find a specific country, which is returned
-- as a reference object.
function Find (Nat_Dates : National_Dates;
               Country : Country_Code)
   return National_Date_Reference;
end National_Date_Containers;
```
package body National_Date_Containers is

function Find (Nat_Dates : National_Dates;
   Country   : Country_Code)
   return National_Date_Reference
is
begin
   for I in Nat_Dates'Range loop
      if Nat_Dates (I).Country = Country then
         return National_Date_Reference'(
            Ref => Nat_Dates (I));
      end if;
   end loop;

   return National_Date_Reference'(Ref => null);
end Find;

end National_Date_Containers;

Code block metadata

MD5: ec37ae93a7052c4bc731b2a7be0763ab

Package National_Date_Containers contains the National_Dates type, which is an array type for declaring containers that we use to store the national day information. We can also see the declaration of the National_Date_Reference type, which is the reference type returned by the Find function when looking for a specific country in the container.

Important

We're declaring the container type (National_Dates) as an array type just to simplify the code. In many cases, however, this approach isn't recommended! Instead, we should use a private type in order to encapsulate — and better protect — the information stored in the actual container.

Finally, let's see a test application that stores information for some countries into the Nat_Dates container and displays the information for a specific country:

Listing 256: show_national_dates.adb

(continues on next page)
procedure Show_National_Dates is

    Nat_Dates : constant National_Dates (1 .. 5) :=
        (new National_Date("USA", Time'(1776, 7, 4)),
         new National_Date("FRA", Time'(1789, 7, 14)),
         new National_Date("DEU", Time'(1990, 10, 3)),
         new National_Date("SPA", Time'(1492, 10, 12)),
         new National_Date("BRA", Time'(1822, 9, 7)));

begin
    Find (Nat_Dates, "FRA").Show;
    -- ^ implicit dereference
end Show_National_Dates;

Code block metadata
MD5: 771ecb91e8f890d4bb9b08115ae833f4

Runtime output
Country: FRA
Year: 1789

Here, we call the Find function to retrieve a reference object, whose reference (access value) has the national day information of France. We then implicitly dereference it to get the tagged object (of National_Date type) and display its information by calling the Show procedure.

Relevant topics
The National_Date_Containers package was implemented specifically as an accompanying package for the National_Date_Info package. It is possible, however, to generalize it, so that we can reuse the container for other record types. In fact, this is actually very straightforward:

Listing 257: generic_containers.ads
function Find (Cont : Container; Elem : T_Cmp) return Ref_Type;
end Generic_Containers;

Listing 258: generic_containers.adb

package body Generic_Containers is
function Find (Cont : Container; Elem : T_Cmp)
return Ref_Type is
begin
for I in Cont'Range loop
if Matches (Cont (I), Elem) then
return Ref_Type'(Ref => Cont (I));
end if;
end loop;
return Ref_Type'(Ref => null);
end Find;
end Generic_Containers;

Code block metadata

MD5: 94c23a4b131a47439b5b4e985c3d6c1

When comparing the Generic Containers package to the National_Date_Containers package, we see that the main difference is the addition of the Matches function, which indicates whether the current element we’re evaluating in the for-loop of the Find function is the one we’re looking for.

In the main application, we can implement the Matches function and declare the National_Date_Containers package as an instance of the Generic_Containers package:

Listing 259: show_national_dates.adb

with Generic_Containers;
with National_Date_Info; use National_Date_Info;
procedure Show_National_Dates is
function Matches_Country (E : National_Date_Access; Elem : Country_Code)
return Boolean is
(E.Country = Elem);

package National_Date_Containers is new Generic_Containers
(T => National_Date, T_Access => National_Date_Access, T_Cmp => Country_Code, Matches => Matches_Country);
use National_Date_Containers;

(continues on next page)
Learning Ada

(subtype National_Dates is Container;

Nat_Dates : constant
National_Dates (1 .. 5) :=
(new National_Date("USA", Time'(1776, 7, 4)),
new National_Date("FRA", Time'(1789, 7, 14)),
new National_Date("DEU", Time'(1990, 10, 3)),
new National_Date("SPA", Time'(1492, 10, 12)),
new National_Date("BRA", Time'(1822, 9, 7)));

begin
  Find (Nat_Dates, "FRA").Show;
end Show_National_Dates;

Code block metadata

MD5: f4dac1fed69b9bccee5dccbf17844adc

Runtime output

Country: FRA
Year: 1789

Here, we instantiate the Generic Containers package with the Matches_Country function, which is an expression function that compares the country component of the current National_Date reference with the name of the country we desire to learn about.

This generalized approach is actually used for the standard containers from the Ada. Containers packages. For example, the Ada.Containers.Vectors is specified as follows:

with Ada.Iterator_Interfaces;

generic
  type Index_Type is range <>;
  type Element_Type is private;
  with function "=" (Left, Right : Element_Type)
    return Boolean is <>;

package Ada.Containers.Vectors
  with Preelaborate, Remote_Types,
       Nonblocking,
       Global => in out synchronized is

  -- OMITTED

  type Reference_Type
    (Element : not null access Element_Type) is private
    with Implicit_Dereference => Element,
    Nonblocking,
    Global => in out synchronized,
    Default_Initial_Condition =>
      (raise Program_Error);
function Reference
(Container : aliased in out Vector;
Index : in Index_Type)
return Reference_Type
  with Pre => Index in
      First_Index (Container) ..
    Last_Index (Container)
  or else raise
      Constraint_Error,
Post =>
      Tampering_With.Cursors_Prohibited
      (Container),
    Nonblocking,
    Global => null,
    Use_Formal => null;

function Reference
(Container : aliased in out Vector;
Position : in Cursor)
return Reference_Type
  with Pre => (Position /= No_Element
    or else raise
      Constraint_Error)
    and then
      (Has_Element
        (Container, Position)
    or else raise
      Program_Error),
Post =>
      Tampering_With.Cursors_Prohibited
      (Container),
    Nonblocking,
    Global => null,
    Use_Formal => null;

end Ada.Containers.Vectors;

(Note that most parts of the Vectors package were omitted for clarity. Please refer to the Ada Reference Manual for the complete package specification.)

Here, we see that the Implicit_Dereference aspect is used in the declaration of Reference_Type, which is the reference type returned by the Reference functions for an index or a cursor.

Also, note that the Vectors package has a formal equality function (=) instead of the Matches function we were using in our Generic_Containers package. The purpose of the formal function, however, is basically the same.

In the Ada Reference Manual
• A.18.2 The Generic Package Containers.Vectors\(^{213}\)


28.2. Anonymous Access Types
28.2.8 Anonymous Access Types and Accessibility Rules

In general, the accessibility rules (page 790) we've seen earlier also apply to anonymous access types. However, there are some subtle differences, which we discuss in this section.

Let's adapt the code example from that section (page 790) to make use of anonymous access types:

Listing 260: library_level.ads

```ada
package Library_Level is

   L0_AO : access Integer;
   L0_Var : aliased Integer;

end Library_Level;
```

Listing 261: show_library_level.adb

```ada
with Library_Level; use Library_Level;

procedure Show_Library_Level is
   L1_Var : aliased Integer;

   procedure Test is
      L2_AO : access Integer;
      L2_Var : aliased Integer;

   begin
      L1_AO := L2_Var'Access;
      -- ^^^^^^  -- ILLEGAL: L2 object to L1 access object
      L2_AO := L2_Var'Access;
      -- ^^^^^^  -- LEGAL: L2 object to L2 access object
   end Test;

begin
   L0_AO := new Integer'(22);
   -- ^^^^^^^^^^^
   -- LEGAL: L0 object to L0 access object

   L0_AO := L1_Var'Access;
   -- ^^^^  -- ILLEGAL: L1 object to L0 access object

   L1_AO := L0_Var'Access;
   -- ^^^^^
   -- LEGAL: L0 object to L1 access object

   L1_AO := L1_Var'Access;
   -- ^^^^^
   -- LEGAL: L1 object to L1 access object

   -- ^^^^^^  -- ILLEGAL: L1 object to L1 access object
```

(continues on next page)
As we see in the code, in general, most accessibility rules are the same as the ones we've discussed when using named access types. For example, an assignment such as \texttt{L0\_AO := L1\_Var'Access} is illegal because we're trying to assign to an access object of less deep level.

However, assignment such as \texttt{L0\_AO := L1\_AO} are possible now: we don't get a type mismatch — as we did with named access types — because both objects are of anonymous access types. Note that the accessibility level cannot be determined at compile time: \texttt{L1\_AO} can hold an access value at library level (which would make the assignment legal) or at a deeper level. Therefore, the compiler introduces an accessibility check here.

However, the accessibility check used in \texttt{L0\_AO := L1\_AO} fails at runtime because the corresponding access value (\texttt{L1\_Var'Access}) is of a deeper level than \texttt{L0\_AO}, which is illegal. (If you comment out the \texttt{L1\_AO := L1\_Var'Access} assignment prior to the \texttt{L0\_AO := L1\_AO} assignment, this accessibility check doesn't fail anymore.)

### Conversions between Anonymous and Named Access Types

In the previous sections, we've discussed accessibility rules for named and anonymous access types separately. In this section, we see that the same accessibility rules apply when mixing both flavors together and converting objects of anonymous to named access types.

Let's adapt parts of the previous code example (page 790) and add anonymous access types to it:
package Library_Level is
  type L0_Integer_Access is access all Integer;
  L0_Var : aliased Integer;
  L0_IA : L0_Integer_Access;
  L0_AO : access Integer;
end Library_Level;

with Library_Level; use Library_Level;
procedure Show_Library_Level is
  type L1_Integer_Access is access all Integer;
  L1_IA : L1_Integer_Access;
  L1_AO : access Integer;
  L1_Var : aliased Integer;
begin
  ---------------------------------------
  -- From named type to anonymous type
  ---------------------------------------
  L0_IA := new Integer'(22);
  L1_IA := new Integer'(42);
  L0_AO := L0_IA;
      -- ^^^^^
      -- LEGAL: assignment from
      -- L0 access object (named type)
      -- to
      -- L0 access object
      -- (anonymous type)
  L0_AO := L1_IA;
      -- ^^^^^
      -- ILLEGAL: assignment from
      -- L1 access object (named type)
      -- to
      -- L0 access object
      -- (anonymous type)
  L1_AO := L0_IA;
      -- ^^^^^
      -- LEGAL: assignment from
      -- L0 access object (named type)
      -- to
      -- L1 access object
      -- (anonymous type)
  L1_AO := L1_IA;
      -- ^^^^^
(continues on next page)
-- LEGAL: assignment from
-- L1 access object (named type)
-- to
-- L1 access object
-- (anonymous type)

---------------------------------------
-- From anonymous type to named type
---------------------------------------

L0_AO := L0_Var’Access;
L1_AO := L1_Var’Access;

L0_IA := L0_Integer_Access (L0_AO);
-- ^^^^^^^^^^^^^^^^^
-- LEGAL: conversion / assignment from
-- L0 access object
-- (anonymous type)
-- to
-- L0 access object (named type)

L0_IA := L0_Integer_Access (L1_AO);
-- ^^^^^^^^^^^^^^^^^
-- ILLEGAL: conversion / assignment from
-- L1 access object
-- (anonymous type)
-- to
-- L0 access object (named type)
-- (accessibility check fails)

L1_IA := L1_Integer_Access (L0_AO);
-- ^^^^^^^^^^^^^^^^^
-- LEGAL: conversion / assignment from
-- L0 access object
-- (anonymous type)
-- to
-- L1 access object (named type)

L1_IA := L1_Integer_Access (L1_AO);
-- ^^^^^^^^^^^^^^^^^
-- LEGAL: conversion / assignment from
-- L1 access object
-- (anonymous type)
-- to
-- L1 access object (named type)

end Show_Library_Level;

As we can see in this code example, mixing access objects of named and anonymous access types doesn't change the accessibility rules. Again, the rules are only violated when the
Learning Ada

target object in the assignment is less deep. This is the case in the \[L0\_\text{A0} := L1\_\text{IA}\] and \[L0\_\text{IA} := L0\_\text{Integer}\_\text{Access}\ (L1\_\text{A0})\] assignments. Otherwise, mixing those access objects doesn't impose additional hurdles.

Accessibility rules on access parameters

In the previous chapter, we saw that the accessibility rules also apply to access values as subprogram parameters (page 794). In the case of access parameters, the rules are a bit less strict (as you may generally expect for anonymous access types), and the accessibility rules are checked at runtime. This allows use to use access values that would be illegal in the case of named access types because of their accessibility levels.

Let's adapt a previous code example to make use of access parameters:

Listing 264: names.ads

```adam
package Names is
  procedure Show (N : access constant String);
end Names;
```

Listing 265: names.adb

```adam
with Ada.Text_IO; use Ada.Text_IO;
package body Names is
  procedure Show (N : access constant String) is
  begin
    -- for I in N'Range loop
    -- N (I) := To_Lower (N (I));
    -- end loop;
    Put_Line ("Name: " & N.all);
  end Show;
end Names;
```

Listing 266: show_names.adb

```adam
with Names; use Names;
procedure Show_Names is
  S : aliased String := "John";
begin
  Show (S'Access);
end Show_Names;
```

Code block metadata

MD5: aa930ba9be3264d01eb9115d27b884eb

Runtime output

Name: John
As we've seen in the previous chapter, compilation fails when we use named access types in this code example. In the case of access parameters, using S'Access doesn't make the compilation fail, nor does the accessibility check fail at runtime because S is still in scope when we call the Show procedure.

### 28.2.9 Anonymous Access-To-Subprograms

In the previous chapter, we talked about named access-to-subprogram types (page 820). Now, we'll see that the anonymous version of those types isn’t much different from the named version.

Let’s start our discussion by declaring a subprogram parameter using an anonymous access-to-procedure type:

**Listing 267: anonymous_access_to_subprogram.ads**

```ada
package Anonymous_Access_To_Subprogram

procedure Proc
  (P : access procedure (I : in out Integer));

end Anonymous_Access_To_Subprogram;
```

**Listing 268: anonymous_access_to_subprogram.adb**

```ada
package body Anonymous_Access_To_Subprogram

procedure Proc
  (P : access procedure (I : in out Integer))
is
  I : Integer := 0;
begin
  P (I);
end Proc;

end Anonymous_Access_To_Subprogram;
```

In this example, we use the anonymous access procedure (I : in out Integer) type as a parameter of the Proc procedure. Note that we need an identifier in the declaration: we cannot leave I out and write access procedure (in out Integer).

Before we look at a test application that makes use of the Anonymous_Access_To_Subprogram package, let's implement two simple procedures that we'll use later on:

**Listing 269: add_ten.ads**

```ada
procedure Add_Ten
  (I : in out Integer);
```

**Listing 270: add_ten.adb**

```ada
procedure Add_Ten
  (I : in out Integer) is
begin
  (continues on next page)
Learning Ada

(continued from previous page)

3. I := I + 10;
4. end Add_Ten;

Listing 271: add_twenty.ads

procedure Add_Twenty ( I : in out Integer);

Listing 272: add_twenty.adb

procedure Add_Twenty ( I : in out Integer) is
begin
  I := I + 20;
end Add_Twenty;

Code block metadata

MD5: 50eaeaf27caa961b35ecdf8acc11fe

Finally, this is our test application:

Listing 273: show_anonymous_access_to_subprograms.adb

with Anonymous_Access_To_Subprogram;
use Anonymous_Access_To_Subprogram;
with Add_Ten;
procedure Show_Anonymous_Access_To_Subprograms is
begin
  Proc (Add_Ten'Access);
-- ^ Getting access to Add_Ten
-- procedure and passing it
-- to Proc
end Show_Anonymous_Access_To_Subprograms;

Code block metadata

MD5: 13143ccf9620d26031484ba160a58fe1

Here, we get access to the Add_Ten procedure and pass it to the Proc procedure. Note that this implementation is not different from the example for named access-to-subprogram types (page 822). In fact, in terms of usage, anonymous access-to-subprogram types are very similar to named access-to-subprogram types. The major differences can be found in the corresponding accessibility rules (page 910).

In the Ada Reference Manual

- 3.10 Access Types


902 Chapter 28. Resource Management
Examples of anonymous access-to-subprogram usage

In the section about named access-to-subprogram types (page 820), we've seen a couple of different usages for those types. In all those examples we discussed, we could instead have used anonymous access-to-subprogram types. Let's see a code example that illustrates that:

Listing 274: all_anonymous_access_to_subprogram.ads

```ada
package All_Anonymous_Access_To_Subprogram is

   -- Anonymous access-to-subprogram as
   -- subprogram parameter:
   --
   procedure Proc
     (P : access procedure (I : in out Integer));

   -- Anonymous access-to-subprogram in
   -- array type declaration:
   --
   type Access_To_Procedure_Array is
      array (Positive range <>) of
         access procedure (I : in out Integer);

   protected type Protected_Integer is
      --
      procedure Mult_Ten;
      procedure Mult_Twenty;
      I : Integer := 1;
   end Protected_Integer;

   -- Anonymous access-to-subprogram as
   -- component of a record type.
   --
   type Rec_Access_To_Procedure is record
      AP : access procedure (I : in out Integer);
   end record;

   -- Anonymous access-to-subprogram as
   -- discriminant:
   --
   type Rec_Access_To_Procedure_Discriminant
      (AP : access procedure
       (I : in out Integer)) is
      record
         I : Integer := 0;
      end record;

   procedure Process
     (R : in out
      Rec_Access_To_Procedure_Discriminant);

   generic
      type T is private;
```

(continues on next page)
Anonymous access-to-subprogram as formal parameter:

```ada
Proc_T : access procedure
(Element : in out T);
procedure Gen_Process (Element : in out T);
end All_Anonymous_Access_To_Subprogram;
```

Listing 275: all_anonymous_access_to_subprogram.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body All_Anonymous_Access_To_Subprogram is

procedure Proc
(P : access procedure (I : in out Integer))
is
    I : Integer := 0;
begin
    Put_Line
    ("Calling procedure for Proc...");
    P (I);
    Put_Line ("Finished.");
end Proc;

procedure Process
(R : in out
    Rec_Access_To_Procedure_Discriminant)
is
begin
    Put_Line
    ("Calling procedure for"
    & " Rec_Access_To_Procedure_Discriminant"
    & " type...");
    R.AP (R.I);
    Put_Line ("Finished.");
end Process;

procedure Gen_Process (Element : in out T) is
begin
    Put_Line
    ("Calling procedure for Gen_Process...");
    Proc_T (Element);
    Put_Line ("Finished.");
end Gen_Process;

protected body Protected_Integer is

procedure Mult_Ten
begin
    I := I * 10;
end Mult_Ten;

procedure Mult_Twenty
begin
    I := I * 20;
end Mult_Twenty;
end Protected_Integer;
```

(continues on next page)
In the `All_Anonymous_Access_To_Subprogram` package, we see examples of anonymous access-to-subprogram types:

- as a subprogram parameter;
- in an array type declaration;
- as a component of a record type;
- as a record type discriminant;
- as a formal parameter of a generic procedure.

Let's implement a test application that makes use of this package:

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Add_Ten;
with Add_Twenty;

with All_Anonymous_Access_To_Subprogram;
use All_Anonymous_Access_To_Subprogram;

procedure Show_Anonymous_Access_To_Subprograms is
  -- Anonymous access-to-subprogram as an object:
  P : access procedure (I : in out Integer);

  -- Array of anonymous access-to-subprogram components:
  PA : constant
      Access_To_Procedure_Array (1 .. 2) :=
      (Add_Ten'Access,
       Add_Twenty'Access);

  -- Anonymous array of anonymous access-to-subprogram components:
  PAA : constant
    array (1 .. 2) of access procedure (I : in out Integer) :=
    (Add_Ten'Access,
     Add_Twenty'Access);

  -- Record with anonymous access-to-subprogram components:
  --
end All_Anonymous_Access_To_Subprogram;
```

28.2. Anonymous Access Types
RA : constant Rec_Access_To_Procedure :=
    (AP => Add_Ten'Access);

-- Record with anonymous
-- access-to-subprogram discriminant:
RD : Rec_Access_To_Procedure_Discriminant
    (AP => Add_Twenty'Access) :=
    (AP => Add_Twenty'Access, I => 0);

-- Generic procedure with formal anonymous
-- access-to-subprogram:
procedure Process_Integer is
    new Gen_Process (T => Integer,
                     Proc_T => Add_Twenty'Access);

-- Object (APP) of anonymous
-- access-to-protected-subprogram:
PI : Protected_Integer;
APP : constant access protected procedure :=
    PI.Mult_Ten'Access;

Some_Int : Integer := 0;

begin
    Put_Line ("Some_Int: " & Some_Int'Image);

    P := Add_Ten'Access;
    Proc (P);
    P (Some_Int);

    P := Add_Twenty'Access;
    Proc (P);
    P (Some_Int);

    Put_Line ("Some_Int: " & Some_Int'Image);

    P := Add_Ten'Access;
    Proc (P);
    P (Some_Int);

    Put_Line ("Some_Int: " & Some_Int'Image);

for I in PA'Range loop
    PA (I) (Some_Int);
    Put_Line ("Some_Int: " & Some_Int'Image);
end loop;

Put_Line ("Finished.");
"Calling procedure from PAA array..."

for I in PA’Range loop
    PAA (I) (Some_Int);
    Put_Line ("Some_Int: " & Some_Int'Image);
end loop;

Put_Line ("Finished.");

Put_Line ("Some_Int: " & Some_Int'Image);

-- Using record with component of
-- anonymous access-to-subprogram type:
RA.AP (Some_Int);
Put_Line ("Some_Int: " & Some_Int'Image);

-- Using record with discriminant of
-- anonymous access-to-subprogram type:
Process (RD);
Put_Line ("RD.I: " & RD.I'Image);

-- Using procedure instantiated with
-- formal anonymous access-to-subprogram:
Process_Integer (Some_Int);
Put_Line ("Some_Int: " & Some_Int'Image);

-- Using object of anonymous
-- access-to-protected-subprogram type:
APP.all;
end Show_Anonymous_Access_To_Subprograms;

Code block metadata

MD5: ec770c17e880a98fd2e9ab0110d4a858

Runtime output

Some_Int:  0
Calling procedure for Proc...
Finished.
Calling procedure for Proc...
Finished.
Some_Int:  30
Calling procedure from PA array...
Some_Int:  40
Some_Int:  60
Finished.
Calling procedure from PAA array...
Some_Int:  70
Some_Int:  90
Finished.
In the Show_Anonymous_Access_To_Subprograms procedure, we see examples of anonymous access-to-subprogram types in:

- in objects (P) and (APP);
- in arrays (PA and PAA);
- in records (RA and RD);
- in the binding to a formal parameter (Proc_T) of an instantiated procedure (Process_Integer);
- as a parameter of a procedure (Proc).

Because we already discussed all these usages in the section about named access-to-subprogram types (page 820), we won't repeat this discussion here. If anything in this code example is still unclear to you, make sure to revisit that section from the previous chapter.

**Application of anonymous access-to-subprogram types**

In general, there isn't much that speaks against using anonymous access-to-subprogram types. We can say, for example, that they're much more useful than anonymous access-to-objects types (page 858), which have many drawbacks (page 860) — as we discussed earlier.

There isn't much to be concerned when using anonymous access-to-subprogram types. For example, we cannot allocate or deallocate a subprogram. As a consequence, we won't have storage management issues affecting these types because the access to those subprograms will always be available and no memory leak can occur.

Also, anonymous access-to-subprogram types can be easier to use than named access-to-subprogram types because of their less strict accessibility rules (page 910). Some of the accessibility issues we might encounter when using named access-to-subprogram types can be solved by declaring them as anonymous types. (We discuss the accessibility rules of anonymous access-to-subprogram types in the next section.)

**Readability**

Note that readability suffers if you use a _cascade_ of anonymous access-to-subprograms. For example:

```ada
package Readability_Issue is

function F
    return access
function (A : Integer)
    return access

Listing 277: readability_issue.ads
```

(continues on next page)
function (B : Float)
  return Integer;
end Readability_Issue;

Listing 278: readability_issue-functions.ads

package Readability_Issue.Functions is
  function To_Integer (V : Float)
    return Integer is
      (Integer (V));
  end To_Integer;

  function Select_Conversion
    (A : Integer)
    return access
      function (B : Float)
        return Integer is
          (To_Integer'Access);
    end Select_Conversion;
end Readability_Issue.Functions;

Listing 279: readability_issue.adb

package body Readability_Issue is
  function F
    return access
      function (A : Integer)
        return access
          function (B : Float)
            return Integer is
              (Select_Conversion'Access);
      end F;
end Readability_Issue;

Code block metadata

MD5: 9e2ac58942c97b44c0d847c28e39bd11

In this example, the definition of F might compile fine, but it's simply too long to be readable. Not only that: we need to carry this chain to other functions as well — such as the Select_Conversion function above. Also, using these functions in an application is not straightforward:

Listing 280: show_readability_issue.adb

with Readability_Issue;
use Readability_Issue;

procedure Show_Readability_Issue is
  F1 : access
    function (A : Integer)
      return access
        function (B : Float)
          return Integer is

(continues on next page)
Therefore, our recommendation is to avoid this kind of access cascading by carefully designing your application. In general, you won't need that.

28.2.10 Accessibility Rules and Anonymous Access-To-Subprograms

In principle, the accessibility rules for anonymous access types (page 896) that we've seen before apply to anonymous access-to-subprograms as well. Also, we had a discussion about accessibility rules and access-to-subprograms (page 845) in the previous chapter. In this section, we review some of the rules that we already know and discuss how they relate to anonymous access-to-subprograms.

In the Ada Reference Manual

• 3.10 Access Types

Named vs. anonymous access-to-subprograms

Let's see an example of a named access-to-subprogram type:

Listing 281: show_access_to_subprogram_error.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Access_To_Subprogram_Error is
  type PI is access
    procedure (I : in out Integer);
  P : PI;
  I : Integer := 0;
begin
  declare
    procedure Add_One (I : in out Integer) is
      begin
        I := I + 1;
    end Add_One;
```

Learning Ada

(continued from previous page)

begin
    P := Add_One’Access;
end;
end Show_Access_To_Subprogram_Error;

Code block metadata

Accessibility_Rules_Anonymous_Access_To_Subprograms.Simple_Example_Named
MD5: 41c36426112e799210b7704dd43b6217

Build output

show_access_to_subprogram_error.adb:18:12: error: subprogram must not be deeper
   than access type
gprbuild: *** compilation phase failed

In this example, we get a compilation error because the lifetime of the Add_One procedure is shorter than the access type P.

In contrast, using an anonymous access-to-subprogram type eliminates the compilation error, i.e. the assignment P := Add_One’Access becomes legal:

Listing 282: show_access_to_subprogram_error.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Access_To_Subprogram_Error is
    P : access procedure (I : in out Integer);
    I : Integer := 0;
begin
    declare
        procedure Add_One (I : in out Integer) is
        begin
            I := I + 1;
        end Add_One;
    begin
        P := Add_One’Access;
        -- RUNTIME ERROR: Add One is out-of-scope
        -- after this line.
    end;
end Show_Access_To_Subprogram_Error;

Code block metadata

Accessibility_Rules_Anonymous_Access_To_Subprograms.Simple_Example_Anonymous
MD5: a5eeb4a716b4f6a932dd74c580a07b66

Runtime output

raised PROGRAM_ERROR : show_access_to_subprogram_error.adb:14 accessibility check
   failed

In this case, the compiler introduces an accessibility check, which fails at runtime because the lifetime of Add_One is shorter than the lifetime of the access object P.

28.2. Anonymous Access Types
**Named vs. anonymous access-to-subprograms as parameters**

Using anonymous access-to-subprograms as parameters allows us to pass subprograms at any level. For certain applications, the restrictions that are applied to named access types might be too strict, so using anonymous access-to-subprograms might be a good way to circumvent those restrictions. They also allow the component developer to be independent of the clients’ specific access types.

Note that the increased flexibility for anonymous access-to-subprograms means that some of the checks that are performed at compile time for named access-to-subprograms are done at runtime for anonymous access-to-subprograms.

**Named access-to-subprograms as a parameter**

Let's see an example using a named access-to-procedure type:

Listing 283: access_to_subprogram_types.ads

```ada
package Access_To_Subprogram_Types is

    type Integer_Array is
        array (Positive range <>) of Integer;

    type Process_Procedure is
        access
            procedure (Arr : in out Integer_Array);

    procedure Process
        (Arr : in out Integer_Array;
         P : Process_Procedure);

end Access_To_Subprogram_Types;
```

Listing 284: access_to_subprogram_types.adb

```ada
package body Access_To_Subprogram_Types is

    procedure Process
        (Arr : in out Integer_Array;
         P : Process_Procedure)
    is
    begin
        P (Arr);
    end Process;

end Access_To_Subprogram_Types;
```

Listing 285: show_access_to_subprogram_error.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Access_To_Subprogram_Types; use Access_To_Subprogram_Types;

procedure Show_Access_To_Subprogram_Error is

    procedure Add_One
        (Arr : in out Integer_Array)
    is
    begin
        for E of Arr loop
```

(continues on next page)
In this example, we declare the Process_Procedure type in the Access_To_Subprogram_Types package and use it in the Process procedure, which we call in the Show_Access_To_Subprogram_Error procedure. The accessibility rules trigger a compilation error because the accesses (Add_One'Access and Display'Access) are at a deeper level than the access-to-procedure type (Process_Procedure).

As we know already, there's no Unchecked_Access attribute that we could use here. An easy way to make this code compile could be to move Add_One and Display to the library level.
Anonymous access-to-subprograms as a parameter

To circumvent the compilation error, we could also use anonymous access-to-subprograms instead:

Listing 286: access_to_subprogram_types.ads

```ada
package Access_To_Subprogram_Types is
  type Integer_Array is
    array (Positive range <>) of Integer;

  procedure Process
    (Arr : in out Integer_Array;
     P   : access procedure
         (Arr : in out Integer_Array));
end Access_To_Subprogram_Types;
```

Listing 287: access_to_subprogram_types.adb

```ada
package body Access_To_Subprogram_Types is
  procedure Process
    (Arr : in out Integer_Array;
     P   : access procedure
         (Arr : in out Integer_Array)) is
  begin
    P (Arr);
  end Process;
end Access_To_Subprogram_Types;
```

Listing 288: show_access_to_subprogram_error.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Access_To_Subprogram_Types; use Access_To_Subprogram_Types;

procedure Show_Access_To_Subprogram_Error is
  procedure Add_One
    (Arr : in out Integer_Array) is
  begin
    for E of Arr loop
      E := E + 1;
    end loop;
  end Add_One;

  procedure Display
    (Arr : in out Integer_Array) is
  begin
    for I in Arr'Range loop
      Put_Line ("Arr (" & Integer'Image (I) & ") := " & Integer'Image (Arr (I)));
    end loop;
  end Display;
```

(continues on next page)
Arr : Integer_Array (1 .. 3) := (1, 2, 3);

begin
  Process (Arr, Display'Access);
  Put_Line ("Add_One...");
  Process (Arr, Add_One'Access);
  Process (Arr, Display'Access);
end Show_Access_To_Subprogram_Error;

Now, the code is accepted by the compiler because anonymous access-to-subprograms used as parameters allow passing of subprograms at any level. Also, we don't see a runtime exception because the subprograms are still accessible when we call Process.

**Iterator**

A typical example that illustrates well the necessity of using anonymous access-to-subprograms is that of a container iterator. In fact, many of the standard Ada containers — the child packages of Ada.Containers — make use of anonymous access-to-subprograms for their Iterate subprograms.

---

**In the Ada Reference Manual**

- A.18.2 The Package Containers.Vectors
- A.18.4 Maps
- A.18.7 Sets

---

Using named access-to-subprograms

Let's start with a simplified container type (Data_Container) using a named access-to-subprogram type (Process_Element) for iteration:

Listing 289: data_processing.ads

```ada
generic
  type Element is private;
package Data_Processing is

  type Data_Container (Last : Positive) is
    private;
  Data_CONTAINER_FULL : exception;

  procedure Append (D : in out Data_Container;
                    E : Element);

  type Process_Element is
  not null access procedure (E : Element);

  procedure Iterate
    (D : Data_Container;
     Proc : Process_Element);

private

  type Data_Container_Storage is
    array (Positive range <>) of Element;
  type Data_Container (Last : Positive) is
    record
      S : Data_Container_Storage (1 .. Last);
      Curr : Natural := 0;
    end record;
end Data_Processing;
```

Listing 290: data_processing.adb

```ada
package body Data_Processing is

  procedure Append (D : in out Data_Container;
                    E : Element) is
  begin
    if D.Curr < D.S'Last then
      D.Curr := D.Curr + 1;
      D.S (D.Curr) := E;
    else
      raise Data_CONTAINER_FULL;
      -- NOTE: This is just a dummy
      -- implementation. A better
      -- strategy is to add actual error
      -- handling when the container is
      -- full.
    end if;
  end Append;

  procedure Iterate
    (D : Data_Container;
     Proc : Process_Element) is
```

(continues on next page)
begin
    for I in D.'First .. D.Curr loop
        Proc (D.S (I));
    end loop;
end Iterate;
end Data_Processing;

Code block metadata
   Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Named
MD5: e48e8200e571b62d027753ee96c47fc

In this example, we declare the Process_Element type in the generic Data_Processing package, and we use it in the Iterate procedure. We then instantiate this package as Float_Data_Processing, and we use it in the Show_Access_To_Subprograms procedure:

Listing 291: float_data_processing.ads

with Data_Processing;
package Float_Data_Processing is
    new Data_Processing (Element => Float);

Listing 292: show_access_to_subprograms.adb

with Ada.Text_IO; use Ada.Text_IO;
with Float_Data_Processing;
use Float_Data_Processing;

procedure Show_Access_To_Subprograms is
    procedure Display (F : Float) is
        begin
            Put_Line (“F :” & Float'Image (F));
        end Display;
        D : Data_Container (5);
        begin
            Append (D, 1.0);
            Append (D, 2.0);
            Append (D, 3.0);
            Iterate (D, Display'Access);
            end Show_Access_To_Subprograms;

Code block metadata
   Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Named
MD5: 64ee435aac57281b7d9cecf538a14e

Build output
show_access_to_subprograms.adb:19:17: error: subprogram must not be deeper than an access type
gprbuild: *** compilation phase failed

Using Display'Access in the call to Iterate triggers a compilation error because its life-
time is shorter than the lifetime of the Process_Element type.

**Using anonymous access-to-subprograms**

Now, let’s use an anonymous access-to-subprogram type in the Iterate procedure:

Listing 293: data_processing.ads

```ada
generic
  type Element is private;
package Data_Processing is
  type Data_Container (Last : Positive) is
    private;
  Data_Container_Full : exception;
  procedure Append (D : in out Data_Container; E : Element);
  procedure Iterate
    (D : Data_Container;
     Proc : not null access procedure (E : Element));
private
  type Data_Container_Storage is
    array (Positive range <>) of Element;
  type Data_Container (Last : Positive) is
    record
      S : Data_Container_Storage (1 .. Last);
      Curr : Natural := 0;
    end record;
end Data_Processing;
```

Listing 294: data_processing.adb

```ada
package body Data_Processing is
  procedure Append (D : in out Data_Container; E : Element) is
  begin
    if D.Curr < D.S'Last then
      D.Curr := D.Curr + 1;
      D.S (D.Curr) := E;
    else
      raise Data_Container_Full;
      -- NOTE: This is just a dummy
      -- implementation. A better
      -- strategy is to add actual error
      -- handling when the container is
      -- full.
    end if;
  end Append;
  procedure Iterate
    (D : Data_Container;
```

(continues on next page)
Proc : not null access
  procedure (E : Element) is
begin
  for I in D.S'First .. D.Curr loop
    Proc (D.S (I));
  end loop;
end Iterate;
end Data_Processing;

Code block metadata
  Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Anonymous
MD5: fa56595ef1734f2f07ad719c36d6d8b5

Note that the only changes we did to the package were to remove the Process_Element type and replace the type of the Proc parameter of the Iterate procedure from a named type (Process_Element) to an anonymous type (not null access procedure (E : Element)).

Now, the same test application we used before (Show_Access_To_Subprograms) compiles as expected:

Listing 295: float_data_processing.ads

with Data_Processing;
package Float_Data_Processing is
new Data_Processing (Element => Float);

Listing 296: show_access_to_subprograms.adb

with Ada.Text_IO; use Ada.Text_IO;
with Float_Data_Processing;
use Float_Data_Processing;

procedure Show_Access_To_Subprograms is
  procedure Display (F : Float) is
begin
  Put_Line ("F :" & Float'Image (F));
end Display;

  D : Data_Container (5);
begin
  Append (D, 1.0);
  Append (D, 2.0);
  Append (D, 3.0);
  Iterate (D, Display'Access);
end Show_Access_To_Subprograms;

Code block metadata
  Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Anonymous
MD5: 64ee435aac5f2817b7d9cecf538a1e4c

Runtime output
Remember that the compiler introduces an accessibility check in the call to Iterate, which is successful because the lifetime of Display'Access is the same as the lifetime of the Proc parameter of Iterate.

28.3 Limited Types

So far, we discussed nonlimited types in most cases. In this chapter, we discuss limited types.

We can think of limited types as an easy way to avoid inappropriate semantics. For example, a lock should not be copied — neither directly, via assignment, nor with pass-by-copy. Similarly, a file, which is really a file descriptor, should not be copied. In this chapter, we'll see example of unwanted side-effects that arise if we don't use limited types for these cases.

28.3.1 Assignment and equality

Limited types have the following restrictions, which we discussed in the Introduction to Ada (page 116) course:

- copying objects of limited types via direct assignments is forbidden; and
- there's no predefined equality operator for limited types.

(Of course, in the case of nonlimited types, assignments are possible and the equality operator is available.)

By having these restrictions for limited types, we avoid inappropriate side-effects for assignment and equality operations. As an example of inappropriate side-effects, consider the case when we apply those operations on record types that have components of access types:

Listing 297: nonlimited_types.ads

```ada
package Nonlimited_Types is

  type Simple_Rec is private;
  type Integer_Access is access Integer;
  function Init (I : Integer) return Simple_Rec;
  procedure Set (E : Simple_Rec;
      I : Integer);
  procedure Show (E : Simple_Rec;
      E_Name : String);

private

  type Simple_Rec is record
    V : Integer_Access;
  end record;
```

(continues on next page)
end Nonlimited_Types;

Listing 298: nonlimited_types.adb

with Ada.Text_IO;  use Ada.Text_IO;
package body Nonlimited_Types is
  function Init (I : Integer) return Simple_Rec is
    return E : Simple_Rec do
      E.V := new Integer' (I);
    end return;
  end Init;

  procedure Set (E : Simple_Rec; I : Integer) is
    E.V.all := I;
  end Set;

  procedure Show (E : Simple_Rec; E_Name : String) is
    Put_Line (E_Name & ".V.all = " & Integer' Image (E.V.all));
  end Show;
end Nonlimited_Types;

Listing 299: show_wrong_assignment_equality.adb

with Ada.Text_IO;  use Ada.Text_IO;
with Nonlimited_Types; use Nonlimited_Types;

procedure Show_Wrong_Assignment_Equality is
  A, B : Simple_Rec := Init (0);

  procedure Show_Compare is
    begin
      if A = B then
        Put_Line ("A = B");
      else
        Put_Line ("A /= B");
      end if;
  end Show_Compare;

begin
  Put_Line ("A := Init (0); A := Init (0);");
  Show (A, "A");
  Show (B, "B");
  Show_Compare;
  Put_Line ("--------");

  Put_Line ("Set (A, 2); Set (B, 3);");
  Set (A, 2);
  Set (B, 3);
end Show_Wrong_Assignment_Equality;
In this code, we declare the Simple_Rec type in the Nonlimited_Types package and use it in the Show_Wrong_Assignment_Equality procedure. In principle, we're already doing many things right here. For example, we're declaring the Simple_Rec type private, so that the component V of access type is encapsulated. Programmers that declare objects of this type cannot simply mess up with the V component. Instead, they have to call the Init function and the Set procedure to initialize and change, respectively, objects of the Simple_Rec type. That being said, there are two problems with this code, which we discuss next.

The first problem we can identify is that the first call to Show_Compare shows that A and B
are different, although both have the same value in the V component \((A.V.all = 0\) and \(B.V.all = 0\)) — this was set by the call to the Init function. What's happening here is that the \(A = B\) expression is comparing the access values \((A.V = B.V)\), while we might have been expecting it to compare the actual integer values after dereferencing \((A.V.all = B.V.all)\). Therefore, the predefined equality function of the Simple_Rec type is useless and dangerous for us, as it misleads us to expect something that it doesn't do.

After the assignment of \(A\) to \(B\) \((B := A)\), the information that the application displays seems to be correct — both \(A.V.all\) and \(B.V.all\) have the same value of two. However, when assigning the value seven to \(B\) by calling \(Set (B, 7)\), we see that the value of \(A.V.all\) has also changed. What's happening here is that the previous assignment \((B := A)\) has actually assigned access values \((B.V := A.V)\), while we might have been expecting it to assign the dereferenced values \((B.V.all := A.V.all)\). Therefore, we cannot simply directly assign objects of Simple_Rec type, as this operation changes the internal structure of the type due to the presence of components of access type.

For these reasons, forbidding these operations for the Simple_Rec type is the most appropriate software design decision. If we still need assignment and equality operators, we can implement custom subprograms for the limited type. We'll discuss this topic in the next sections.

In addition to the case when we have components of access types, limited types are useful for example when we want to avoid the situation in which the same information is copied to multiple objects of the same type.

---

**In the Ada Reference Manual**

- **7.5 Limited Types**

---

**Assignments**

Assignments are forbidden when using objects of limited types. For example:

```
package Limited_Types is

  type Simple_Rec is limited private;
  type Integer_Access is access Integer;
  function Init (I : Integer) return Simple_Rec;

private

  type Simple_Rec is limited record
    V : Integer_Access;
  end record;

end Limited_Types;
```

```
package body Limited_Types is

  function Init (I : Integer) return Simple_Rec
  is
    (continues on next page)

219 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html

28.3. Limited Types 923
```
begin
  return E : Simple_Rec do
  E.V := new Integer'(I);
  end return;
end Init;
end Limited_Types;

Listing 302: show_limited_assignment.adb

with Limited_Types; use Limited_Types;
procedure Show_Limited_Assignment is
  A, B : Simple_Rec := Init (0);
begin
  B := A;
end Show_Limited_Assignment;

Code block metadata

MD5: 019c16f7feac896fd8c37d40d0522dc8

Build output

show_limited_assignment.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

In this example, we declare the limited private type Simple_Rec and two objects of this type (A and B) in the Show_Limited_Assignment procedure. (We discuss more about limited private types later (page 928)).

As expected, we get a compilation error for the B := A statement (in the Show_Limited_Assignment procedure). If we need to copy two objects of limited type, we have to provide a custom procedure to do that. For example, we can implement a Copy procedure for the Simple_Rec type:

Listing 303: limited_types.ads

package Limited_Types is
  type Integer_Access is access Integer;
  type Simple_Rec is limited private;
  function Init (I : Integer) return Simple_Rec;
  procedure Copy (From : Simple_Rec;
                  To : in out Simple_Rec);
private
  type Simple_Rec is limited record
    V : Integer_Access;
  end record;
end Limited_Types;
package body Limited_Types is

function Init (I : Integer) return Simple_Rec
is
begin
  return E : Simple_Rec do
    E.V := new Integer'(I);
  end return;
end Init;

procedure Copy (From : Simple_Rec; To : in out Simple_Rec)
is
begin
  -- Copying record components
  To.V.all := From.V.all;
end Copy;
end Limited_Types;

with Limited_Types; use Limited_Types;

procedure Show_Limited_Assignment is
A, B : Simple_Rec := Init (0);
begin
  Copy (From => A, To => B);
end Show_Limited_Assignment;

The Copy procedure from this example copies the dereferenced values of From to To, which matches our expectation for the Simple_Rec. Note that we could have also implemented a Shallow_Copy procedure to copy the actual access values (i.e. To.V := From.V). However, having this kind of procedure can be dangerous in many cases, so this design decision must be made carefully. In any case, using limited types ensures that only the assignment subprograms that are explicitly declared in the package specification are available.

Equality

Limited types don’t have a predefined equality operator. For example:

package Limited_Types is

  type Integer_Access is access Integer;
  type Simple_Rec is limited private;

  function Init (I : Integer) return Simple_Rec;

end Limited_Types;
private

    type Simple_Rec is limited record
        V : Integer_Access;
    end record;
end Limited_Types;

Listing 307: limited_types.adb

package body Limited_Types is

    function Init (I : Integer) return Simple_Rec is
        begin
            return E : Simple_Rec do
                E.V := new Integer'(I);
            end return;
        end Init;
end Limited_Types;

Listing 308: show_limited_equality.adb

with Ada.Text_IO; use Ada.Text_IO;
with Limited_Types; use Limited_Types;

procedure Show_Limited_Equality is
    A : Simple_Rec := Init (5);
    B : Simple_Rec := Init (6);
    begin
        if A = B then
            Put_Line ("A = B");
        else
            Put_Line ("A /= B");
        end if;
    end Show_Limited_Equality;

Code block metadata

MD5: dad31b5e36de0b3b7824f723a60e5aa0

Build output

show_limited_equality.adb:8:09: error: there is no applicable operator "=" for private type "Simple_Rec" defined at limited_types.ads:5
gprbuild: *** compilation phase failed

As expected, the comparison A = B triggers a compilation error because no predefined = operator is available for the Simple_Rec type. If we want to be able to compare objects of this type, we have to implement the = operator ourselves. For example, we can do that for the Simple_Rec type:

Listing 309: limited_types.ads

package Limited_Types is

    type Integer_Access is access Integer;

(continues on next page)
Learning Ada

(continued from previous page)

type Simple_Rec is limited private;

function Init (I : Integer) return Simple_Rec;

function "=" (Left, Right : Simple_Rec)
return Boolean;

private

type Simple_Rec is limited record
  V : Integer_Access;
end record;

end Limited_Types;

Listing 310: limited_types.adb

package body Limited_Types is

  function Init (I : Integer) return Simple_Rec
is
    begin
      return E : Simple_Rec do
        E.V := new Integer'(I);
      end return;
    end Init;

  function "=" (Left, Right : Simple_Rec)
begin
  -- Comparing record components
return Left.V.all = Right.V.all;
end ";

end Limited_Types;

Listing 311: show_limited_equality.adb

with Ada.Text_IO; use Ada.Text_IO;
with Limited_Types; use Limited_Types;

procedure Show_Limited_Equality is
  A : Simple_Rec := Init (5);
  B : Simple_Rec := Init (6);
  begin
    if A = B then
      Put_Line ("A = B");
    else
      Put_Line ("A /= B");
    end if;
  end Show_Limited_Equality;

Code block metadata
MD5: f56b2229443a5e4e33c402b41b02d318

Runtime output
Here, the \( \neq \) operator compares the dereferenced values of Left\( .V \) and Right\( .V \), which matches our expectation for the Simple\_Rec type. Declaring types as limited ensures that we don't have unreasonable equality comparisons, and allows us to create reasonable replacements when required.

### In other languages

In C++, you can overload the assignment operator. For example:

```cpp
class Simple\_Rec
{
public:
  // Overloaded assignment
  Simple\_Rec\& operator= (const Simple\_Rec\& obj);
private:
  int *V;
};
```

In Ada, however, we can only define the equality operator (=). Defining the assignment operator (:=) is not possible. The following code triggers a compilation error as expected:

```ada
package Limited\_Types is
  type Integer\_Access is access Integer;
  type Simple\_Rec is limited private;
  procedure ":=" (To : in out Simple\_Rec
                From : Simple\_Rec);
  -- ...
end Limited\_Types;
```

#### 28.3.2 Limited private types

As we've seen in code examples from the previous section, we can apply information hiding (page 307) to limited types. In other words, we can declare a type as limited\_private instead of just limited. For example:

```
package Simple\_Recs is
  type Rec is limited private;
private
  type Rec is limited record
    I : Integer;
  end record;
end Simple\_Recs;
```

**Code block metadata**
In this case, in addition to the fact that assignments are forbidden for objects of this type (because Rec is limited), we cannot access the record components.

Note that in this example, both partial and full views of the Rec record are of limited type. In the next sections, we discuss how the partial and full views can have non-matching declarations.

In the Ada Reference Manual

- 7.5 Limited Types

Non-Record Limited Types

In principle, only record types can be declared limited, so we cannot use scalar or array types. For example, the following declarations won't compile:

Listing 313: non_record_limited_error.ads

```ada
package Non_Record_Limited_Error is

  type Limited_Enumeration is
    limited (Off, On);

  type Limited_Integer is new
    limited Integer;

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Rec is new
    limited Integer_Array (1 .. 2);

end Non_Record_Limited_Error;
```

However, we've mentioned in a previous chapter (page 309) that private types don't have to be record types necessarily. In this sense, limited private types makes it possible for us to use types other than record types in the full view and still benefit from the restrictions of limited types. For example:

Listing 314: simple_recs.ads

```ada
package Simple_Recs is

  type Limited_Enumeration is
    limited private;

  type Limited_Integer is

  (continues on next page)
```

220 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
Here, Limited_Enumeration, Limited_Integer, and Limited_Integer_Array_2 are limited private types that encapsulate an enumeration type, an integer type, and a constrained array type, respectively.

**Partial and full view of limited types**

In the previous example, both partial and full views of the Rec type were limited. We may actually declare a type as `limited private` (in the public part of a package), while its full view is nonlimited. For example:

```ada
package Simple_Recs is

  type Rec is limited private;
  -- Partial view of Rec is limited

  private

  type Rec is record
  -- Full view of Rec is nonlimited
    I : Integer;
  end record;

end Simple_Recs;
```

In this case, only the partial view of Rec is limited, while its full view is nonlimited. When deriving from Rec, the view of the derived type is the same as for the parent type:
Listing 316: simple_recs-child.ads

```ada
package Simple_Recs.Child is
type Rec_Derived is new Rec;
-- As for its parent, the
-- partial view of Rec_Derived
-- is limited, but the full view
-- is nonlimited.
end Simple_Recs.Child;
```

Clients must nevertheless comply with their partial view, and treat the type as if it is in fact limited. In other words, if you use the Rec type in a subprogram or package outside of the Simple_Recs package (or its child packages), the type is limited from that perspective:

Listing 317: use_rec_in_subprogram.adb

```ada
with Simple_Recs; use Simple_Recs;
procedure Use_Rec_In_Subprogram is
  R1, R2 : Rec;
begin
  R1.I := 1;
  R2 := R1;
end Use_Rec_In_Subprogram;
```

Here, compilation fails because the type Rec is limited from the procedure's perspective.

Limitations

Note that the opposite — declaring a type as `private` and its full full view as `limited private` — is not possible. For example:

Listing 318: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is private;
```

(continues on next page)
private

   type Rec is limited record
      I : Integer;
   end record;
end Simple_Recs;

Code block metadata

MD5: ed1c8a2dcf3cc2c49b1497cf4c9d3a5a

Build output

simple_recs.ads:7:09: error: completion of nonlimited type cannot be limited
gprbuild: *** compilation phase failed

As expected, we get a compilation error in this case. The issue is that the partial view cannot be allowed to mislead the client about what's possible. In this case, if the partial view allows assignment, then the full view must actually provide assignment. But the partial view can restrict what is actually possible, so a limited partial view need not be completed in the full view as a limited type.

In addition, tagged limited private types cannot have a nonlimited full view. For example:

Listing 319: simple_recs.ads

package Simple_Recs is

   type Rec is tagged limited private;

private

   type Rec is tagged record
      I : Integer;
   end record;
end Simple_Recs;

Code block metadata

MD5: cadb9ca1346a98fb65f9059f9b29f865

Build output

use_rec_in_subprogram.adb:6:06: error: no selector "I" for private type "Rec"
   defined at simple_recs.ads:3
use_rec_in_subprogram.adb:7:04: error: left hand of assignment must not be limited
   type
simple_recs.ads:7:09: error: completion of limited tagged type must be limited
gprbuild: *** compilation phase failed

Here, compilation fails because the type Rec is nonlimited in its full view.
Limited and nonlimited in full view

Declaring the full view of a type as limited or nonlimited has implications in the way we can use objects of this type in the package body. For example:

Listing 320: simple_recs.ads

```ada
package Simple_Recs is

  type Rec_Limited_Full is limited private;
  type Rec_Nonlimited_Full is limited private;

  procedure Copy
  (From : Rec_Limited_Full;
   To : in out Rec_Limited_Full);

  procedure Copy
  (From : Rec_Nonlimited_Full;
   To : in out Rec_Nonlimited_Full);

private

  type Rec_Limited_Full is limited record
     I : Integer;
   end record;

  type Rec_Nonlimited_Full is record
     I : Integer;
   end record;

end Simple_Recs;
```

Listing 321: simple_recs.adb

```ada
package body Simple_Recs is

  procedure Copy
  (From : Rec_Limited_Full;
   To : in out Rec_Limited_Full)
  is
    To := From;
    -- ERROR: assignment is forbidden because
    -- Rec_Limited_Full is limited in
    -- its full view.
  end Copy;

  procedure Copy
  (From : Rec_Nonlimited_Full;
   To : in out Rec_Nonlimited_Full)
  is
    To := From;
    -- OK: assignment is allowed because
    -- Rec_Nonlimited_Full is
    -- nonlimited in its full view.
  end Copy;

end Simple_Recs;
```

Code block metadata

28.3. Limited Types
Here, both Rec_Limited_Full and Rec_Nonlimited_Full are declared as private limited. However, Rec_Limited_Full type is limited in its full view, while Rec_Nonlimited_Full is nonlimited. As expected, the compiler complains about the To := From assignment in the Copy procedure for the Rec_Limited_Full type because its full view is limited (so no assignment is possible). Of course, in the case of the objects of Rec_Nonlimited_Full type, this assignment is perfectly fine.

### Limited private component

Another example mentioned by the Ada Reference Manual (7.3.1\(^{221}\), 5/1) is about an array type whose component type is limited private, but nonlimited in its full view. Let's see a complete code example for that:

```ada
package Limited_Nonlimited_Arrays is

    type Limited_Private is
        limited private;

    function Init return Limited_Private;

    -- The array type Limited_Private_Array is limited because the type of its
    -- component is limited.
    type Limited_Private_Array is
        array (Positive range <>) of
            Limited_Private;

    limited private

    type Limited_Private is
        record
            A : Integer;
        end record;

    -- Limited_Private_Array type is
    -- nonlimited at this point because
    -- its component is nonlimited.
    --
    -- The assignments below are OK:
    A1 : Limited_Private_Array (1 .. 5);
    A2 : Limited_Private_Array := A1;

end Limited_Nonlimited_Arrays;
```

---

\(^{221}\) [Ada Reference Manual](http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html)
As we can see in this example, the limitedness of the array type Limited_Private_Array depends on the limitedness of its component type Limited_Private. In the private part of Limited_Nonlimited_Arrays package, where Limited_Private is nonlimited, the array type Limited_Private_Array becomes nonlimited as well. In contrast, in the Show_Limited_Nonlimited_Array, the array type is limited because its component is limited in that scope.

### In the Ada Reference Manual

- **7.3.1 Private Operations**

  [222](http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html)
Tagged limited private types

For tagged private types, the partial and full views must match: if a tagged type is limited in the partial view, it must be limited in the full view. For example:

Listing 325: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is tagged limited private;
private
  type Rec is tagged limited record
    I : Integer;
  end record;
end Simple_Recs;
```

Here, the tagged Rec type is limited both in its partial and full views. Any mismatch in one of the views triggers a compilation error. (As an exercise, you may remove any of the `limited` keywords from the code example and try to compile it.)

For further reading...

This rule is for the sake of dynamic dispatching and classwide types. The compiler must not allow any of the types in a derivation class — the set of types related by inheritance — to be different regarding assignment and equality (and thus inequality). That’s necessary because we are meant to be able to manipulate objects of any type in the entire set of types via the partial view presented by the root type, without knowing which specific tagged type is involved.

### 28.3.3 Explicitly limited types

Under certain conditions, limited types can be called explicitly limited — note that using the `limited` keyword in a part of the declaration doesn’t necessary ensure this, as we’ll see later.

Let’s start with an example of an explicitly limited type:

Listing 326: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is limited record
    I : Integer;
  end record;
end Simple_Recs;
```
The Rec type is also explicitly limited when it's declared limited in the private type's completion (in the package's private part):

```ada
package Simple_Recs is
  type Rec is limited private;
private
  type Rec is limited record
    I : Integer;
  end record;
end Simple_Recs;
```

In this case, Rec is limited both in the partial and in the full view, so it's considered explicitly limited.

However, *as we've learned before* (page 930), we may actually declare a type as **limited private** in the public part of a package, while its full view is nonlimited. In this case, the limited type is not considered explicitly limited anymore.

For example, if we make the full view of the Rec nonlimited (by removing the **limited** keyword in the private part), then the Rec type isn't explicitly limited anymore:

```ada
package Simple_Recs is
  type Rec is limited private;
private
  type Rec is record
    I : Integer;
  end record;
end Simple_Recs;
```

Now, even though the Rec type was declared as limited private, the full view indicates that it's actually a nonlimited type, so it isn't explicitly limited.

Note that **tagged limited private types** (page 936) are always explicitly limited types — because, as we've learned before, they cannot have a nonlimited type declaration in its full view.

28.3. **Limited Types**
28.3.4 Subtypes of Limited Types

We can declare subtypes of limited types. For example:

```ada
package Simple_Recs is

  type Limited_Integer_Array (L : Positive) is
    limited private;

  subtype Limited_Integer_Array_2 is
    Limited_Integer_Array (2);

private

  type Integer_Array is
    array (Positive range <>) of Integer;

  type Limited_Integer_Array (L : Positive) is
    limited record
      Arr : Integer_Array (1 .. L);
    end record;

end Simple_Recs;
```

Here, `Limited_Integer_Array_2` is a subtype of the `Limited_Integer_Array` type. Since `Limited_Integer_Array` is a limited type, the `Limited_Integer_Array_2` subtype is limited as well. A subtype just introduces a name for some constraints on an existing type. As such, a subtype doesn't change the limitedness of the constrained type.

We can test this in a small application:

```ada
with Simple_Recs; use Simple_Recs;

procedure Test_Limitedness is
  Dummy_1, Dummy_2 : Limited_Integer_Array_2;
begin
  Dummy_2 := Dummy_1;
end Test_Limitedness;
```
As expected, compilations fails because Limited_Integer_Array_2 is a limited (sub)type.

28.3.5 Deriving from limited types

In this section, we discuss the implications of deriving from limited types. As usual, let's start with a simple example:

Listing 331: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is limited null record;
  type Rec_Derived is new Rec;
end Simple_Recs;
```

In this example, the Rec_Derived type is derived from the Rec type. Note that the Rec_Derived type is limited because its ancestor is limited, even though the `limited` keyword doesn't show up in the declaration of the Rec_Derived type. Note that we could have actually used the `limited` keyword here:

```ada
type Rec_Derived is limited new Rec;
```

Therefore, we cannot use the assignment operator for objects of Rec_Derived type:

Listing 332: test_limitedness.adb

```ada
with Simple_Recs; use Simple_Recs;
procedure Test_Limitedness is
  Dummy_1, Dummy_2 : Rec_Derived;
begin
  Dummy_2 := Dummy_1;
end Test_Limitedness;
```
Learning Ada

test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

Note that we cannot derive a limited type from a nonlimited ancestor:

Listing 333: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is null record;
  type Rec_Derived is limited new Rec;
end Simple_Recs;
```

Code block metadata

MD5: 78a7574cc6233ddc826359acb6e644ee

Build output

simple_recs.ads:5:04: error: parent type "Rec" of limited type must be limited
gprbuild: *** compilation phase failed

As expected, the compiler indicates that the ancestor Rec should be of limited type.
In fact, all types in a derivation class are the same — either limited or not. (That is especially important with dynamic dispatching via tagged types. We discuss this topic in another chapter.)

In the Ada Reference Manual

- 7.3 Private Types and Private Extensions
- 7.5 Limited Types

Deriving from limited private types

Of course, we can also derive from limited private types. However, there are more rules in this case than the ones we've seen so far. Let's start with an example:

Listing 334: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is limited private;
  private
    type Rec is limited null record;
end Simple_Recs;
```

227 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
Learning Ada

Listing 335: simple_recs-ext.ads

```ada
package Simple_Recs.Ext is
  type Rec_Derived is new Rec;
  -- OR:
  -- type Rec_Derived is
  -- limited new Rec;
end Simple_Recs.Ext;
```

Listing 336: test_limitedness.adb

```ada
with Simple_Recs.Ext; use Simple_Recs.Ext;

procedure Test_Limitedness is
  Dummy_1, Dummy_2 : Rec_Derived;
begin
  Dummy_2 := Dummy_1;
end Test_Limitedness;
```

**Code block metadata**

**Project:** Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_Limited_Types.Derived_Limited_Private_Type  
**MD5:** c6eed14520589b9c1e11c17bd6179c19

**Build output**

```
test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

Here, Rec_Derived is a limited type derived from the (limited private) Rec type. We can verify that Rec_Derived type is limited because the compilation of the Test_Limitedness procedure fails.

**Deriving from non-explicitly limited private types**

Up to this point, we have discussed *explicitly limited types* (page 936). Now, let's see how derivation works with *non-explicitly* limited types.

Any type derived from a limited type is always limited, even if the full view of its ancestor is nonlimited. For example, let's modify the full view of Rec and make it nonlimited (i.e. make it *not explicitly* limited):

Listing 337: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is limited private;
private
  type Rec is null record;
end Simple_Recs;
```

**Code block metadata**
Here, Rec_Derived is a limited type because the partial view of Rec is limited. The fact that the full view of Rec is nonlimited doesn't affect the Rec_Derived type — as we can verify with the compilation error in the Test_Limitedness procedure.

Note, however, that a derived type becomes nonlimited in the **private part or the body** of a child package if it isn't explicitly limited. In this sense, the derived type inherits the *nonlimitedness* of the parent's full view. For example, because we're declaring Rec_Derived as **is new** Rec in the child package (Simple_Recs.Ext), we're saying that Rec_Derived is limited *outside* this package, but nonlimited in the private part and body of the Simple_Recs.Ext package. We can verify this by copying the code from the Test_Limitedness procedure to a new procedure in the body of the Simple_Recs.Ext package:

Listing 338: simple_recs-ext.ads

```ada
package Simple_Recs.Ext
with Elaborate_Body is

-- Rec_Derived is derived from Rec, which is a
-- limited private type that is nonlimited in
-- its full view.
--
-- Rec_Derived isn't explicitly limited.
-- Therefore, it's nonlimited in the private
-- part of Simple_Recs.Ext and its package
-- body.

-- type Rec_Derived is new Rec;

end Simple_Recs.Ext;
```

Listing 339: simple_recs-ext.adb

```ada
package body Simple_Recs.Ext is

procedure Test_Child_Limitedness is
    Dummy_1, Dummy_2 : Rec_Derived;
begin
    -- Here, Rec_Derived is a nonlimited
    -- type because Rec is nonlimited in
    -- its full view.
    Dummy_2 := Dummy_1;
end Test_Child_Limitedness;
end Simple_Recs.Ext;
```

Listing 340: test_limitedness.adb

```ada
-- We copied the code to the
-- Test_Child_Limitedness procedure (in the
-- body of the Simple_Recs.Ext package) and

(continues on next page)
Learning Ada

(continued from previous page)

--- commented it out here.

--- You may uncomment the code to verify that Rec_Derived is limited in this procedure.

---

-- with Simple_Recs.Ext; use Simple_Recs.Ext;

procedure Test_Limitedness is
  -- Dummy_1, Dummy_2 : Rec_Derived;
begin
  -- Dummy_2 := Dummy_1;
  null;
end Test_Limitedness;

Code block metadata

MD5: f480cd05afff622e451684a0293cb982

In the Test_Child_Limitedness procedure of the Simple_Recs.Ext package, we can use the Rec_Derived as a nonlimited type because its ancestor Rec is nonlimited in its full view. (As we've learned before (page 933), if a limited type is nonlimited in its full view, we can copy objects of this type in the private part of the package specification or in the package body.)

Outside of the package, both Rec and Rec_Derived types are limited types. Therefore, if we uncomment the code in the Test_Limitedness procedure, compilation fails there (because Rec_Derived is viewed as descending from a limited type).

Deriving from tagged limited private types

The rules for deriving from tagged limited private types are slightly different than the rules we've seen so far. This is because tagged limited types are always explicitly limited types (page 936).

Let's look at an example:

Listing 341: simple_recs.ads

```ada
package Simple_Recs is
  type Tagged_Rec is tagged limited private;
  private
  type Tagged_Rec is tagged limited null record;
end Simple_Recs;
```

Listing 342: simple_recs-ext.ads

```ada
package Simple_Recs.Ext is
  type Rec_Derived is new Tagged_Rec with private;
end Simple_Recs.Ext;
```

(continues on next page)
private

    type Rec_Derived is new
    Tagged_Rec with null record;

end Simple_Recs.Ext;

Listing 343: test_limitedness.adb

with Simple_Recs.Ext; use Simple_Recs.Ext;

procedure Test_Limitedness is
    Dummy_1, Dummy_2 : Rec_Derived;
begin
    Dummy_2 := Dummy_1;
end Test_Limitedness;

In this example, Rec_Derived is a tagged limited type derived from the Tagged_Rec type.
(Again, we can verify the limitedness of the Rec_Derived type with the Test_Limitedness procedure.)

As explained previously, the derived type (Rec_Derived) is a limited type, even though the limited keyword doesn't appear in its declaration. We could, of course, include the limited keyword in the declaration of Rec_Derived:

Listing 344: simple_recs-ext.ads

package Simple_Recs.Ext is

    type Rec_Derived is limited new
    Tagged_Rec with private;

private

    type Rec_Derived is limited new
    Tagged_Rec with null record;

end Simple_Recs.Ext;

Code block metadata

MD5: b82a58a4bf9701b321000c52bf121977

Build output

simple_recs-ext.ads:1: Simple Recs.ext cannot be used as a main program
gprbind: invocation of gnatbind failed
gprbuild: unable to bind simple_recs-ext.ads
(Obviously, if we include the `limited` keyword in the partial view of the derived type, we must include it in its full view as well.)

**Deriving from limited interfaces**

The rules for limited interfaces are different from the ones for limited tagged types. In contrast to the rule we've seen in the previous section, a type that is derived from a limited type isn't automatically limited. In other words, it does **not** inherit the `limitedness` from the interface. For example:

Listing 345: simple_recs.ads

```ada
package Simple_Recs is
   type Limited_IF is limited interface;
end Simple_Recs;
```

Listing 346: simple_recs-ext.ads

```ada
package Simple_Recs.Ext is
   type Rec_Derived is new Limited_IF with private;
private
   type Rec_Derived is new Limited_IF with null record;
end Simple_Recs.Ext;
```

Listing 347: test_limitedness.adb

```ada
with Simple_Recs.Ext; use Simple_Recs.Ext;
procedure Test_Limitedness is
   Dummy_1, Dummy_2 : Rec_Derived;
beg
   Dummy_2 := Dummy_1;
end Test_Limitedness;
```

**Code block metadata**

MD5: d9cf0bd26b86d0caec82eff2a2ec6ead

Here, `Rec_Derived` is derived from the limited `Limited_IF` interface. As we can see, the `Test_Limitedness` compiles fine because `Rec_Derived` is nonlimited.

Of course, if we want `Rec_Derived` to be limited, we can make this explicit in the type declaration:

Listing 348: simple_recs-ext.ads

```ada
package Simple_Recs.Ext is
   type Rec_Derived is limited new Limited_IF with private;
```

(continues on next page)
Learning Ada

(continued from previous page)

private

    type Rec_Derived is limited new
    Limited_IF with null record;

end Simple_Recs.Ext;

Listing 349: test_limitedness.adb

with Simple_Recs.Ext; use Simple_Recs.Ext;

procedure Test_Limitedness is
    Dummy_1, Dummy_2 : Rec_Derived;
begin
    Dummy_2 := Dummy_1;
end Test_Limitedness;

Code block metadata

MD5: abb295cbfd5ade5f351991c2fba5f19c

Build output

test_limitedness.adb:6:04: error: left hand of assignment must not be limited type

gprbuild: *** compilation phase failed

Now, compilation of Test_Limitedness fails because Rec_Derived is explicitly limited.

28.3.6 Immutably Limited Types

According to the Annotated Ada Reference Manual, "an immutably limited type is a type that cannot become nonlimited subsequently in a private part or in a child unit." In fact, while we were talking about partial and full view of limited types (page 930), we've seen that limited private types can become nonlimited in their full view. Such limited types are not immutably limited.

The Annotated Ada Reference Manual also says that "if a view of the type makes it immutably limited, then no copying (assignment) operations are ever available for objects of the type. This allows other properties; for instance, it is safe for such objects to have access discriminants that have defaults or designate other limited objects." We'll see examples of this later on.

Immutably limited types include:

- explicitly limited types
- tagged limited types (i.e. with the keyword limited);
- tagged limited private type;
- limited private type that have at least one access discriminant (page 868) with a default expression;
- task types, protected types, and synchronized interfaces;
- any types derived from immutably limited types.

Let's look at a code example that shows instances of immutably limited types:
package Show_Immutably_Limited_Types is

-- Explicitly limited type
--
type Explicitly_Limited_Rec is limited record
  A : Integer;
end record;

-- Tagged limited type
--
type Limited_Tagged_Rec is tagged limited record
  A : Integer;
end record;

-- Tagged limited private type
--
type Limited_Tagged_Private is
tagged limited private;

-- Limited private type with an access discriminant that has a default expression
--
type Limited_Rec_Access_D
  (AI : access Integer := new Integer) is
  limited private;

-- Task type
--
task type TT is
  entry Start;
  entry Stop;
end TT;

-- Protected type
--
protected type PT is
  function Value return Integer;
private
  A : Integer;
end PT;

-- Synchronized interface
--
type SI is synchronized interface;

-- A type derived from an immutably limited type
--
type DerivedImmutable is new

(continues on next page)
Explicitly_Limited_Rec;

private

type Limited_Tagged_Private is tagged limited
record
A : Integer;
end record;

type Limited_Rec_Access_D
(AI : access Integer := new Integer)
is limited
record
A : Integer;
end record;

end Show_Immutably_Limited_Types;

Listing 351: show_immutably_limited_types.adb

package body Show_Immutably_Limited_Types is

task body TT is
begin
accept Start;
accept Stop;
end TT;

protected body PT is
function Value return Integer is
(PT.A);
end PT;

end Show_Immutably_Limited_Types;

Code block metadata

MD5: 6bcb9582a10eedc96040ab11cd320153

Build output

show_immutably_limited_types.ads:31:30: warning: coextension will not be deallocated when its associated owner is deallocated [enabled by default]

In the Show_Immutably_Limited_Types package above, we see multiple instances of immutably limited types. (The comments in the source code indicate each type.)

In the Ada Reference Manual

- 7.5 Limited Types

\[228\] http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
Non immutably limited types

Not every limited type is immutably limited. We already mentioned untagged private limited types, which can become nonlimited in their full view. In addition, we have nonsynchronized limited interface types. As mentioned earlier in this chapter, a type derived from a nonsynchronized limited interface (page 945), can be nonlimited, so it's not immutably limited.

In the Ada Reference Manual
- 7.3.1 Private Operations
- 7.5 Limited Types

28.3.7 Limited Types and Discriminants

28.3.8 Record components of limited type

In this section, we discuss the implications of using components of limited type. Let's start by declaring a record component of limited type:

Listing 352: simple_recs.ads

```ada
package Simple_Recs is
  type Int_Rec is limited record
    V : Integer;
  end record;

  type Rec is limited record
    IR : Int_Rec;
  end record;

end Simple_Recs;
```

As soon as we declare a record component of some limited type, the whole record is limited. In this example, the Rec record is limited due to the presence of the IR component of limited type.

Also, if we change the declaration of the Rec record from the previous example and remove the limited keyword, the type itself remains implicitly limited. We can see that when trying to assign to objects of Rec type in the Show_Implicitly_Limited procedure:

Listing 353: simple_recs.ads

```ada
package Simple_Recs is
  type Int_Rec is limited record
    V : Integer;
  end record;

end Simple_Recs;
```

229 http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html
230 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
Learning Ada

Listing 354: show_implicitly_limited.adb

```ada
with Simple_Recs; use Simple_Recs;

procedure Show_Implicitly_Limited is
   A, B : Rec;
begin
   B := A;
end Show_Implicitly_Limited;
```

Code block metadata

MD5: 39770daecfc4579407a799e14f9feff9

Build output

show_implicitly_limited.adb:6:04: error: left hand of assignment must not be limited type
show_implicitly_limited.adb:6:04: error: component "IR" of type "Rec" has limited type
gprbuild: *** compilation phase failed

Here, the compiler indicates that the assignment is forbidden because the Rec type has a component of limited type. The rationale for this rule is that an object of a limited type doesn't allow assignment or equality, including the case in which that object is a component of some enclosing composite object. If we allowed the enclosing object to be copied or tested for equality, we'd be doing it for all the components, too.

In the Ada Reference Manual

- 3.8 Record Types

28.3.9 Limited types and aggregates

Note: This section was originally written by Robert A. Duff and published as Gem #1: Limited Types in Ada 2005 and Gem #2.

In this section, we focus on using aggregates to initialize limited types.

Historically

Prior to Ada 2005, aggregates were illegal for limited types. Therefore, we would be faced with a difficult choice: Make the type limited, and initialize it like this:

232 https://www.adacore.com/gems/gem-1
233 https://www.adacore.com/gems/gem-2
Listing 355: persons.ads

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

package Persons is
  type Limited_Person;
  type Limited_Person_Access is
    access all Limited_Person;
  type Limited_Person is limited record
    Name : Unbounded_String;
    Age  : Natural;
  end record;
end Persons;
```

Listing 356: show_non_aggregate_init.adb

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;
with Persons; use Persons;

procedure Show_Non_Aggregate_Init is
  X : Limited_Person;
begin
  X.Name := To_Unbounded_String ("John Doe");
  X.Age := 25;
end Show_Non_Aggregate_Init;
```

**Code block metadata**


MD5: fd3dcb6251f7b6912dafcca052932be2

which has the maintenance problem the full coverage rules are supposed to prevent. Or, make the type nonlimited, and gain the benefits of aggregates, but lose the ability to prevent copies.

**Full coverage rules for limited types**

Previously, we discussed full coverage rules for aggregates (page 448). They also apply to limited types.

**Historically**

The full coverage rules have been aiding maintenance since Ada 83. However, prior to Ada 2005, we couldn't use them for limited types.

Suppose we have the following limited type:
This type has a self-reference; it doesn't make sense to copy objects, because Self would end up pointing to the wrong place. Therefore, we would like to make the type limited, to prevent developers from accidentally making copies. After all, the type is probably private, so developers using this package might not be aware of the problem. We could also solve that problem with controlled types, but controlled types are expensive, and add unnecessary complexity if not needed.

We can initialize objects of limited type with an aggregate. Here, we can say:

```
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

with Persons; use Persons;

procedure Show_Aggregate_Box_Init is
  X : aliased Limited_Person :=
    (Self => <>,
     Name => To_Unbounded_String ("John Doe"),
     Age => 25,
     Shoe_Size => 10);
begin
  null;
end Show_Aggregate_Box_Init;
```

The Self => <> means use the default value of Limited_Person'Unchecked_Access. Since Limited_Person appears inside the type declaration, it refers to the "current instance" of
one very important requirement should be noted: the implementation is required to build the value of X in place; it cannot construct the aggregate in a temporary variable and then copy it into X, because that would violate the whole point of limited objects — you can't copy them.

Historically
Since Ada 2005, an aggregate is allowed to be limited; we can say:

```
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;
with Persons; use Persons;

procedure Show_Aggregate_Init is
    X : aliased Limited_Person :=
        (Self => null, -- Wrong!
         Name => To_Unbounded_String ("John Doe"),
         Age  => 25,
         Shoe_Size => 10);
    begin
        X.Self := X'Unchecked_Access;
    end Show_Aggregate_Init;
```

Code block metadata

MD5: 793ee000fd777d0aa5c15e16132ec411

It seems uncomfortable to set the value of Self to the wrong value (null) and then correct it. It also seems annoying that we have a (correct) default value for Self, but prior to Ada 2005, we couldn't use defaults with aggregates. Since Ada 2005, a new syntax in aggregates is available: <> means "use the default value, if any". Therefore, we can replace

```
Self => null by Self => <>.
```

Important
Note that using <> in an aggregate can be dangerous, because it can leave some components uninitialized. <> means "use the default value". If the type of a component is scalar, and there is no record-component default, then there is no default value.

For example, if we have an aggregate of type String, like this:

```
procedure Show_String_Box_Init is
    Uninitialized_Const_Str : constant String :=
        (1 .. 10 => <>);
    begin
        null;
    end Show_String_Box_Init;
```

Code block metadata
we end up with a 10-character string all of whose characters are invalid values. Note that this is no more nor less dangerous than this:

```
procedure Show_Dangerous_String is
  Uninitialized_String_Var : String (1 .. 10);
    -- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ no initialization
  Uninitialized_Const_Str : constant String :=
    Uninitialized_String_Var;
begin
  null;
end Show_Dangerous_String;
```

As always, one must be careful about uninitialized scalar objects.

### 28.3.10 Constructor functions for limited types

**Note:** This section was originally written by Robert A. Duff and published as Gem #3. Given that we can use build-in-place aggregates for limited types, the obvious next step is to allow such aggregates to be wrapped in an abstraction — namely, to return them from functions. After all, interesting types are usually private, and we need some way for clients to create and initialize objects.

**Historically**

Prior to Ada 2005, constructor functions (that is, functions that create new objects and return them) were not allowed for limited types. Since Ada 2005, fully-general constructor functions are allowed.

Let's see an example:

---

<sup>234</sup> [https://www.adacore.com/gems/gem-3](https://www.adacore.com/gems/gem-3)
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

package P is
  task type Some_Task_Type;

  protected type Some_Protected_Type is
    -- dummy type
  end Some_Protected_Type;

  type T (<>) is limited private;
  function Make_T (Name : String) return T;
    -- constructor function
private
  type T is limited
  record
    Name : Unbounded_String;
    My_Task : Some_Task_Type;
    My_Prot : Some_Protected_Type;
  end record;
end P;

package body P is
  task body Some_Task_Type is
    begin
      null;
    end Some_Task_Type;

  protected body Some_Protected_Type is
    end Some_Protected_Type;

  function Make_T (Name : String) return T is
    begin
      return (Name =>
                To_Unbounded_String (Name),
                others => <>);
    end Make_T;
end P;

Given the above, clients can say:

procedure Show_Constructor_Function is
  My_T : T := Make_T
    (Name => "Bartholomew Cubbins");
begin
As for aggregates, the result of Make_T is built in place (that is, in My_T), rather than being created and then copied into My_T. Adding another level of function call, we can do:

Listing 365: show_rumplestiltskin_constructor.adb

```ada
with P; use P;

procedure Show_Rumplestiltskin_Constructor is
  function Make_Rumplestiltskin return T is
    begin
      return Make_T (Name => "Rumplestiltskin");
    end Make_Rumplestiltskin;

  Rumplestiltskin_Is_My_Name : constant T :=
    Make_Rumplestiltskin;

  begin
    null;
  end Show_Rumplestiltskin_Constructor;
```

Historically

Note that Rumplestiltskin_Is_My_Name is constant. Prior to Ada 2005, it was impossible to create a constant limited object, because there was no way to initialize it.

The (<>) on type T means that it has unknown discriminants from the point of view of the client. This is a trick that prevents clients from creating default-initialized objects (that is, X : T; is illegal). Thus clients must call Make_T whenever an object of type T is created, giving package P full control over initialization of objects.

Ideally, limited and nonlimited types should be just the same, except for the essential difference: you can't copy limited objects (and there's no language-defined equality operator). By allowing functions and aggregates for limited types, we're very close to this goal. Some languages have a specific feature called constructor. In Ada, a constructor is just a function that creates a new object.
Historically

Prior to Ada 2005, constructors only worked for nonlimited types. For limited types, the only way to construct on declaration was via default values, which limits you to one constructor. And the only way to pass parameters to that construction was via discriminants.

Consider the following package:

```ada
with Ada.Containers.Ordered_Sets;

package Aux is
  generic
    with package OS is new
      Ada.Containers.Ordered_Sets (<>);
  function Gen_Singleton_Set (Element : OS.Element_Type) return OS.Set;
end Aux;
```

Since Ada 2005, we can say:

```ada
with Ada.Containers.Ordered_Sets;
with Aux;

procedure Show_Set_Decl is
  package Integer_Sets is new
    Ada.Containers.Ordered_Sets
      (Element_Type => Integer);
  use Integer_Sets;

  function Singleton_Set is new
    Aux.Gen_Singleton_Set
      (OS => Integer_Sets);

  This_Set : Set := Empty_Set;
  That_Set : Set := Singleton_Set
    (Element => 42);
```

(continues on next page)
begin
  null;
end Show_Set_Decl;

Code block metadata
MD5: 443fc3390b0f3e5516d91c80f16bed3f

whether or not Set is limited. This_Set : Set := Empty_Set; seems clearer than:

Listing 369: show_set_decl.adb

with Ada.Containers.Ordered_Sets;

procedure Show_Set_Decl is
  package Integer_Sets is new Ada.Containers.Ordered_Sets
    (Element_Type => Integer);
  use Integer_Sets;

  This_Set : Set;
begin
  null;
end Show_Set_Decl;

Code block metadata
MD5: e5b6c0e148cfdb1987ab302ec1f53bd

which might mean "default-initialize to the empty set" or might mean "leave it uninitialized, and we'll initialize it in later".

28.3.11 Return objects

Extended return statements for limited types

Note: This section was originally written by Robert A. Duff and published as Gem #10: Limited Types in Ada 2005.

Previously, we discussed extended return statements (page 615). For most types, extended return statements are no big deal — it's just syntactic sugar. But for limited types, this syntax is almost essential:

Listing 370: task_construct_error.ads

package Task_Construct_Error is

  task type Task_Type (Discriminant : Integer);

  (continues on next page)

235 https://www.adacore.com/gems/ada-gem-10
function Make_Task (Val : Integer)
    return Task_Type;
end Task_Construct_Error;

Listing 371: task_construct_error.adb

package body Task_Construct_Error is

    task body Task_Type is
        begin
            null;
        end Task_Type;

    function Make_Task (Val : Integer)
        return Task_Type
    is
        Result : Task_Type
            (Discriminant => Val * 3);
        begin
            -- some statements...
            return Result; -- Illegal!
        end Make_Task;

end Task_Construct_Error;

Code block metadata

MD5: f55b1c367d2931ece4d352d209fe6b3b

The return statement here is illegal, because Result is local to Make_Task, and returning it would involve a copy, which makes no sense (which is why task types are limited). Since Ada 2005, we can write constructor functions for task types:

Listing 372: task_construct.ads

package Task_Construct is

    task type Task_Type (Discriminant : Integer);

    function Make_Task (Val : Integer)
        return Task_Type;

end Task_Construct;

Listing 373: task_construct.adb

package body Task_Construct is

    task body Task_Type is
        begin
            null;
        end Task_Type;

    function Make_Task (Val : Integer)
        return Task_Type is
        begin
            return Result : Task_Type

(continues on next page)
(Discriminant => Val * 3)

  do
  -- some statements...
  null;
  end return;
  end Make_Task;
end Task_Construct;

Code block metadata
MD5: c91a24f09a76afe1c25d1a55bcbee910

If we call it like this:

Listing 374: show_task_construct.adb

with Task_Construct; use Task_Construct;

procedure Show_Task_Construct is
  My_Task : Task_Type := Make_Task (Val => 42);
begin
  null;
end Show_Task_Construct;

Code block metadata
MD5: 01809b031a844c829f2ead253864ca75

Result is created in place in My_Task. Result is temporarily considered local to Make_Task during the -- some statements part, but as soon as Make_Task returns, the task becomes more global. Result and My_Task really are one and the same object.

When returning a task from a function, it is activated after the function returns. The -- some statements part had better not try to call one of the task's entries, because that would deadlock. That is, the entry call would wait until the task reaches an accept statement, which will never happen, because the task will never be activated.

Initialization and function return

As mentioned in the previous section, the object of limited type returned by the initialization function is built in place. In other words, the return object is built in the object that is the target of the assignment statement.

For example, we can see this when looking at the address of the object returned by the Init function, which we call to initialize the limited type Simple_Rec:

Listing 375: limited_types.ads

package Limited_Types is

  type Integer_Access is access Integer;

  type Simple_Rec is limited private;

(continues on next page)
function Init (I : Integer) return Simple_Rec;

private

    type Simple_Rec is limited record
        V : Integer_Access;
    end record;

end Limited_Types;

Listing 36: limited_types.adb

with Ada.Text_IO; use Ada.Text_IO;
with System;
with System.Address_Image;

package body Limited_Types is
    function Init (I : Integer) return Simple_Rec is begin return E : Simple_Rec do E.V := new Integer'(I);
        Put_Line ("E'Address (Init): " & System.Address_Image (E'Address));
    end return;
end Init;
end Limited_Types;

Listing 37: show_limited_init.adb

with Ada.Text_IO; use Ada.Text_IO;
with System;
with System.Address_Image;
with Limited_Types; use Limited_Types;

procedure Show_Limited_Init is begin declare A : Simple_Rec := Init (0); begin Put_Line ("A'Address (local): 
        & System.Address_Image (A'Address));
        Put_Line ("----");
    declare B : Simple_Rec := Init (0);
    begin Put_Line ("B'Address (local): 
        & System.Address_Image (B'Address));
    end; end Show_Limited_Init;

Code block metadata

28.3. Limited Types
When running this code example and comparing the address of the object \( E \) in the \texttt{Init} function and the object that is being initialized in the \texttt{Show_Limited_Init} procedure, we see that the return object \( E \) (of the \texttt{Init} function) and the local object in the \texttt{Show_Limited_Init} procedure are the same object.

\textbf{Important}

When we use nonlimited types, we’re actually copying the returned object — which was locally created in the function — to the object that we’re assigning the function to.

For example, let’s modify the previous code and make \texttt{Simple_Rec} nonlimited:

\begin{verbatim}
package Non_Limited_Types is
  type Integer_Access is access Integer;
  type Simple_Rec is private;
  function Init (I : Integer) return Simple_Rec is
    return E : Simple_Rec do
      E.V := new Integer'(I);
      Put_Line ("E'Address (Init): ", System.Address_Image (E'Address));
    end return;
end Non_Limited_Types;
\end{verbatim}
Listing 380: show_non_limited_init_by_copy.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System;
with System.Address_Image;
with Non_Limited_Types; use Non_Limited_Types;

procedure Show_Non_Limited_Init_By_Copy is
   A, B : Simple_Rec;
begin
   declare
      A : Simple_Rec := Init (0);
   begin
      Put_Line ("A'Address (local): " & System.Address_Image (A'Address));
   end;
   Put_Line ("----");

   declare
      B : Simple_Rec := Init (0);
   begin
      Put_Line ("B'Address (local): " & System.Address_Image (B'Address));
   end;
end Show_Non_Limited_Init_By_Copy;
```

In this case, we see that the local object `E` in the `Init` function is not the same as the object it's being assigned to in the `Show_Non_Limited_Init_By_Copy` procedure. In fact, `E` is being copied to `A` and `B`. 

28.3. Limited Types
28.3.12 Building objects from constructors

Note: This section was originally written by Robert A. Duff and published as Gem #11: Limited Types in Ada 2005\(^{236}\).

We've earlier seen examples of constructor functions for limited types similar to this:

Listing 381: p.ads

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;
package P is
  task type Some_Task_Type;
  protected type Some_Protected_Type is
    -- dummy type
  end Some_Protected_Type;

  type T is limited private;
  function Make_T (Name : String) return T;
  -- ^^^^^^ constructor function
private
  type T is limited
    record
      Name : Unbounded_String;
      My_Task : Some_Task_Type;
      My_Prot : Some_Protected_Type;
    end record;
end P;
```

Listing 382: p.adb

```ada
package body P is
  task body Some_Task_Type is
  begin
    null;
  end Some_Task_Type;

  protected body Some_Protected_Type is
  end Some_Protected_Type;

  function Make_T (Name : String) return T is
  begin
    return (Name => To_Unbounded_String (Name),
             others => <>);
  end Make_T;
end P;
```

Listing 383: p-aux.ads

```ada
package P.Aux is
  function Make_Rumplestiltskin return T;
end P.Aux;
```

\(^{236}\) https://www.adacore.com/gems/ada-gem-11
It is useful to consider the various contexts in which these functions may be called. We've already seen things like:

```ada
with P; use P;
with P.Aux; use P.Aux;

procedure Show_Rumplestiltskin_Constructor is
    Rumplestiltskin_Is_My_Name : constant T := Make_Rumplestiltskin;
begin
    null;
end Show_Rumplestiltskin_Constructor;
```

Code block metadata
MD5: 2fe193516df6452eccece88132660f8e5

in which case the limited object is built directly in a standalone object. This object will be finalized whenever the surrounding scope is left.

We can also do:

```ada
with P; use P;
with P.Aux; use P.Aux;

procedure Show_Parameter_Constructor is
    procedure Do_Something (X : T) is null;
begin
    Do_Something (X => Make_Rumplestiltskin);
end Show_Parameter_Constructor;
```

Code block metadata
MD5: 61ccaefb4b7cfc42c065aa15543fc13b

Here, the result of the function is built directly in the formal parameter X of Do_Something. X will be finalized as soon as we return from Do_Something.

28.3. Limited Types
We can allocate initialized objects on the heap:

Listing 387: show_heap_constructor.adb

```ada
with P; use P;
with P.Aux; use P.Aux;

procedure Show_Heap_Constructor is

  type T_Ref is access all T;
  Global : T_Ref;

  procedure Heap_Alloc is
    Local : T_Ref;
    To_Global : Boolean := True;
  begin
    Local := new T'(Make_Rumplestiltskin);
    if To_Global then
      Global := Local;
    end if;
    end Heap_Alloc;
  begin
    null;
  end Show_Heap_Constructor;
```

The result of the function is built directly in the heap-allocated object, which will be finalized when the scope of T_Ref is left (long after Heap_Alloc returns).

We can create another limited type with a component of type T, and use an aggregate:

Listing 388: show_outer_type.adb

```ada
with P; use P;
with P.Aux; use P.Aux;

procedure Show_Outer_Type is

  type Outer_Type is limited record
    This : T;
    That : T;
  end record;

  Outer_Obj : Outer_Type :=
    (This => Make_Rumplestiltskin,
     That => Make_T (Name => ""));

  begin
    null;
  end Show_Outer_Type;
```

---

Chapter 28. Resource Management
As usual, the function results are built in place, directly in Outer_Obj. This and Outer_Obj. That, with no copying involved.

The one case where we cannot call such constructor functions is in an assignment statement:

Listing 389: show_illegal_constructor.adb

```ada
with P; use P;
with P.Aux; use P.Aux;
procedure Show_Illegal_Constructor is
   Rumplestiltskin_Is_My_Name: T;
begin
   Rumplestiltskin_Is_My_Name :=
      Make_T (Name => ""); -- Illegal!
end Show_Illegal_Constructor;
```

Code block metadata

MD5: f7b0c78e9fe2e104b82dfff25ac3e3a

Build output

show_illegal_constructor.adb:7:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

which is illegal because assignment statements involve copying. Likewise, we can’t copy a limited object into some other object:

Listing 390: show_illegal_constructor.adb

```ada
with P; use P;
with P.Aux; use P.Aux;
procedure Show_Illegal_Constructor is
   Rumplestiltskin_Is_My_Name: constant T :=
      Make_T (Name => "");
   Other: T :=
      Rumplestiltskin_Is_My_Name; -- Illegal!
begin
   null;
end Show_Illegal_Constructor;
```

28.3.13 Limited types as parameter

Previously, we saw that parameters can be passed by copy or by reference (page 618). Also, we discussed the concept of by-copy and by-reference types. Explicitly limited types (page 936) are by-reference types. Consequently, parameters of these types are always passed by reference.

For further reading...

As an example of the importance of this rule, consider the case of a lock (as an abstract data type). If such a lock object were passed by copy, the Acquire and Release operations...
would be working on copies of this object, not on the original one. This would lead to
timing-dependent bugs.

Let's reuse an example of an explicitly limited type:

Listing 391: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is limited record
    I : Integer;
  end record;
end Simple_Recs;
```

**Code block metadata**


MD5: de73a20140628420830ed9fe0b2dedb5

In this example, Rec is a by-reference type because the type declaration is an explicit limited
record. Therefore, the parameter R of the Proc procedure is passed by reference.

We can run the Test application below and compare the address of the R object from Test
to the address of the R parameter of Proc to determine whether both Rs refer to the same
object or not:

Listing 392: simple_recs.ads

```ada
with System;
package Simple_Recs is
  type Rec is limited record
    I : Integer;
  end record;
  procedure Proc (R : in out Rec;
                  A : out System.Address);
end Simple_Recs;
```

Listing 393: simple_recs.adb

```ada
package body Simple_Recs is
  procedure Proc (R : in out Rec;
                  A : out System.Address) is
    begin
      R.I := 0;
      A := R'Address;
    end Proc;
end Simple_Recs;
```

Listing 394: test.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System; use System;
with System.Address_Image;
```

(continues on next page)
with Simple_Recs; use Simple_Recs;

procedure Test is
R : Rec; AR_Proc, AR_Test : System.Address;
begin
AR_Proc := R'Address;
Proc (R, AR_Test);
Put_Line ("R'Address (Proc): ",
   & System.Address_Image (AR_Proc));
Put_Line ("R'Address (Test): ",
   & System.Address_Image (AR_Test));
if AR_Proc = AR_Test then
   Put_Line ("R was passed by reference.");
else
   Put_Line ("R was passed by copy.");
end if;
end Test;

When running the Test application, we confirm that R was passed by reference. Note, however, that the fact that R was passed by reference doesn't automatically imply that Rec is a by-reference type: the type could have been ambiguous, and the compiler could have just decided to pass the parameter by reference in this case.

Therefore, we have to rely on the rules specified in the Ada Reference Manual:

1. If a limited type is explicitly limited, a parameter of this type is a by-reference type.
   • The rule applies to all kinds of explicitly limited types. For example, consider private limited types where the type is declared limited in the private type's completion (in the package's private part): a parameter of this type is a by-reference type.

2. If a limited type is not explicitly limited, a parameter of this type is neither a by-copy nor a by-reference type.
   • In this case, the decision whether the parameter is passed by reference or by copy is made by the compiler.

In the Ada Reference Manual

• 6.2 Formal Parameter Modes

237 http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
Learning Ada

- 6.4.1 Parameter Associations^{238}
- 7.5 Limited Types^{239}

^{238} http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
^{239} http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
Part III

Introduction To SPARK
This tutorial is an interactive introduction to the SPARK programming language and its formal verification tools. You will learn the difference between Ada and SPARK and how to use the various analysis tools that come with SPARK.

This document was prepared by Claire Dross and Yannick Moy.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

**Note:** Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](https://learn.adacore.com/zip/learning-ada_code.zip). The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- **Project:** Courses.Intro_To_Ada.Imperative_Language.Greet
- **MD5:** cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/
cia89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).
This tutorial is an introduction to the SPARK programming language and its formal verification tools. You need not know any specific programming language (although going over the *Introduction to Ada course* (page 5) first may help) or have experience in formal verification.

### 29.1 What is it?

SPARK refers to two different things:

- a programming language targeted at functional specification and static verification,
- a set of development and verification tools for that language.

The SPARK language is based on a subset of the Ada language. Ada is particularly well suited to formal verification since it was designed for critical software development. SPARK builds on that foundation.

Version 2012 of Ada introduced the use of *aspects*, which can be used for subprogram contracts, and version 2014 of SPARK added its own aspects to further aid static analysis.
29.2 What do the tools do?

We start by reviewing static verification of programs, which is verification of the source code performed without compiling or executing it. Verification uses tools that perform static analysis. These can take various forms. They include tools that check types and enforce visibility rules, such as the compiler, in addition to those that perform more complex reasoning, such as abstract interpretation, as done by a tool like CodePeer\textsuperscript{242} from AdaCore. The tools that come with SPARK perform two different forms of static analysis:

- \textit{flow analysis} is the fastest form of analysis. It checks initializations of variables and looks at data dependencies between inputs and outputs of subprograms. It can also find unused assignments and unmodified variables.
- \textit{proof} checks for the absence of runtime errors as well as the conformance of the program with its specifications.

29.3 Key Tools

The tool for formal verification of the SPARK language is called \textit{GNATprove}. It checks for conformance with the SPARK subset and performs flow analysis and proof of the source code. Several other tools support the SPARK language, including both the GNAT compiler\textsuperscript{243} and the GNAT Studio integrated development environment\textsuperscript{244}.

29.4 A trivial example

We start with a simple example of a subprogram in Ada that uses SPARK aspects to specify verifiable subprogram contracts. The subprogram, called \textit{Increment}, adds 1 to the value of its parameter \( X \):

\begin{verbatim}
Listing 1: increment.ads

procedure Increment
(X : in out Integer)
with
Global => null,
Depends => (X => X),
Pre => X < Integer'Last,
Post => X = X'Old + 1;

Listing 2: increment.adb

procedure Increment
(X : in out Integer)
is
begin
X := X + 1;
end Increment;
\end{verbatim}

\textbf{Code block metadata}

Project: Courses.Intro_To_Spark.Overview.Trivial_Example
MD5: ce28b1fabc44917b6cc208639c187064

\textsuperscript{242} https://www.adacore.com/codepeer
\textsuperscript{243} https://www.adacore.com/gnatpro
\textsuperscript{244} https://www.adacore.com/gnatpro/toolsuite/gps
Prover output

Phase 1 of 2: generation of Global contracts...
Phase 2 of 2: flow analysis and proof...
increment.adb:5:10: info: overflow check proved
increment.ads:4:03: info: data dependencies proved
increment.ads:5:03: info: flow dependencies proved
increment.ads:7:14: info: postcondition proved
increment.ads:7:24: info: overflow check proved

The contracts are written using the Ada aspect feature and those shown specify several properties of this subprogram:

• The SPARK Global aspect says that Increment does not read or write any global variables.

• The SPARK Depend aspect is especially interesting for security: it says that the value of the parameter \( X \) after the call depends only on the (previous) value of \( X \).

• The Pre and Post aspects of Ada specify functional properties of Increment:
  - Increment is only allowed to be called if the value of \( X \) prior to the call is less than \text{Integer}'Last. This ensures that the addition operation performed in the subprogram body doesn't overflow.
  - Increment does indeed perform an increment of \( X \): the value of \( X \) after a call is one greater than its value before the call.

GNATprove can verify all of these contracts. In addition, it verifies that no error can be raised at runtime when executing Increment's body.

29.5 The Programming Language

It's important to understand why there are differences between the SPARK and Ada languages. The aim when designing the SPARK subset of Ada was to create the largest possible subset of Ada that was still amenable to simple specification and sound verification.

The most notable restrictions from Ada are related to exceptions and access types, both of which are known to considerably increase the amount of user-written annotations required for full support. Backwards goto statements and controlled types are also not supported since they introduce non-trivial control flow. The two remaining restrictions relate to side-effects in expressions and aliasing of names, which we now cover in more detail.

29.6 Limitations

29.6.1 No side-effects in expressions

The SPARK language doesn't allow side-effects in expressions. In other words, evaluating a SPARK expression must not update any object. This limitation is necessary to avoid unpredictable behavior that depends on order of evaluation, parameter passing mechanisms, or compiler optimizations. The expression for Dummy below is non-deterministic due to the order in which the two calls to \( F \) are evaluated. It's therefore not legal SPARK.
Learning Ada

Listing 3: show_illegal_ada_code.adb

```
procedure Show_Illegal_Ada_Code is
  function F (X : in out Integer) return Integer is
    Tmp : constant Integer := X;
    begin
      X := X + 1;
      return Tmp;
    end F;
    Dummy : Integer := 0;
    begin
      Dummy := F(Dummy) - F(Dummy);  -- ??
    end Show_Illegal_Ada_Code;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Illegal_Ada_Code
MD5: a5cbf1824526857da94791ac1790200c

Build output

show_illegal_ada_code.adb:13:28: error: value may be affected by call to "F"
  because order of evaluation is arbitrary
  gprbuild: *** compilation phase failed

Prover output

Phase 1 of 2: generation of Global contracts ...
show_illegal_ada_code.adb:13:28: error: value may be affected by call to "F"
  because order of evaluation is arbitrary
  gnatprove: error during generation of Global contracts

In fact, the code above is not even legal Ada, so the same error is generated by the GNAT compiler. But SPARK goes further and GNATprove also produces an error for the following equivalent code that is accepted by the Ada compiler:

Listing 4: show_illegal_spark_code.adb

```
procedure Show_Illegal_SPARK_Code is
  Dummy : Integer := 0;
  function F return Integer is
    Tmp : constant Integer := Dummy;
    begin
      Dummy := Dummy + 1;
      return Tmp;
    end F;
    begin
      Dummy := F - F;  -- ??
    end Show_Illegal_SPARK_Code;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Illegal_SPARK_Code
MD5: e747edb6ee147adb7fba97c9e7c8d5ef

Prover output
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_illegal_spark_code.adb:5:13: error: function with output global "Dummy" is not allowed in SPARK
gnatprove: error during analysis of data and information flow

The SPARK languages enforce the lack of side-effects in expressions by forbidding side-effects in functions, which include modifications to either parameters or global variables. As a consequence, SPARK forbids functions with **out** or **in out** parameters in addition to functions modifying a global variable. Function F below is illegal in SPARK, while Function Incr might be legal if it doesn't modify any global variables and function Incr_And_Log might be illegal if it modifies global variables to perform logging.

```ada
function F (X : in out Integer) return Integer; -- Illegal
function Incr (X : Integer) return Integer; -- OK?
function Incr_And_Log (X : Integer) return Integer; -- OK?
```

In most cases, you can easily replace these functions by procedures with an **out** parameter that returns the computed value.

When it has access to function bodies, GNATprove verifies that those functions are indeed free from side-effects. Here for example, the two functions Incr and Incr_And_Log have the same signature, but only Incr is legal in SPARK. Incr_And_Log isn't: it attempts to update the global variable Call_Count.

Listing 5: side_effects.ads

```ada
package Side_Effects is
  function Incr (X : Integer) return Integer; -- OK?
  function Incr_And_Log (X : Integer) return Integer; -- OK?
end Side_Effects;
```

Listing 6: side_effects.adb

```ada
package body Side_Effects is
  function Incr (X : Integer) return Integer
    is (X + 1); -- OK
  Call_Count : Natural := 0;
  function Incr_And_Log (X : Integer) return Integer is
    begin
      Call_Count := Call_Count + 1; -- Illegal
      return X + 1;
    end Incr_And_Log;
end Side_Effects;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.Overview.Side_Effects
MD5: 1b555e4b7bb519eea4df718a9356a2ed

**Prover output**

---

29.6. Limitations
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
side_effects.ads:5:13: error: function with output global "Call_Count" is not allowed in SPARK
  gnatprove: error during analysis of data and information flow

29.6.2 No aliasing of names

Another restriction imposed by the SPARK subset concerns aliasing\(^{245}\). We say that two names are aliased if they refer to the same object. There are two reasons why aliasing is forbidden in SPARK:

- It makes verification more difficult because it requires taking into account the fact that modifications to variables with different names may actually update the same object.
- Results may seem unexpected from a user point of view. The results of a subprogram call may depend on compiler-specific attributes, such as parameter passing mechanisms, when its parameters are aliased.

Aliasing can occur as part of the parameter passing that occurs in a subprogram call. Functions have no side-effects in SPARK, so aliasing of parameters in function calls isn't problematic; we need only consider procedure calls. When a procedure is called, SPARK verifies that no out or in out parameter is aliased with either another parameter of the procedure or a global variable modified in the procedure's body.

Procedure Move_To_Total is an example where the possibility of aliasing wasn't taken into account by the programmer:

Listing 7: no_aliasing.adb

```ada
procedure No_Aliasing is
  Total : Natural := 0;
  procedure Move_To_Total (Source : in out Natural)
    with Post => Total = Total'Old + Source'Old and Source = 0
  is
  begin
    Total := Total + Source;
    Source := 0;
  end Move_To_Total;

  X : Natural := 3;
begin
  Move_To_Total (X); -- OK
  pragma Assert (Total = 3); -- OK
  Move_To_Total (Total); -- flow analysis error
  pragma Assert (Total = 6); -- runtime error
end No_Aliasing;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Aliasing
MD5: 91038ef030fe27e3b000ab3db9c134ad

Prover output

\(^{245}\) https://en.wikipedia.org/wiki/Aliasing_(computing)
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
no_aliasing.adb:18:19: high: formal parameter "Source" and global "Total" are aliased (SPARK RM 6.4.2)
gnatprove: unproved check messages considered as errors

Runtime output

raised ADAASSERTIONSASSERTION_ERROR : no_aliasing.adb:19

Move_To_Total adds the value of its input parameter Source to the global variable Total and then resets Source to 0. The programmer has clearly not taken into account the possibility of an aliasing between Total and Source. (This sort of error is quite common.)

This procedure itself is valid SPARK. When doing verification, GNATprove assumes, like the programmer did, that there's no aliasing between Total and Source. To ensure this assumption is valid, GNATprove checks for possible aliasing on every call to Move_To_Total. Its final call in procedure No_Aliasing violates this assumption, which produces both a message from GNATprove and a runtime error (an assertion violation corresponding to the expected change in Total from calling Move_To_Total). Note that the postcondition of Move_To_Total is not violated on this second call since integer parameters are passed by copy and the postcondition is checked before the copy-back from the formal parameters to the actual arguments.

Aliasing can also occur as a result of using access types (pointers246 in Ada). These are restricted in SPARK so that only benign aliasing is allowed, when both names are only used to read the data. In particular, assignment between access objects operates at transfer of ownership, where the source object loses its permission to read or write the underlying allocated memory.

Procedure Ownership_Transfer is an example of code that is legal in Ada but rejected in SPARK due to aliasing:

Listing 8: ownership_transfer.adb

```ada
procedure Ownership_Transfer is
  type Int_Ptr is access Integer;
  X : Int_Ptr;
  Y : Int_Ptr;
  Dummy : Integer;
begin
  X := new Integer'(1);
  X.all := X.all + 1;
  Y := X;
  Y.all := Y.all + 1;
  X.all := X.all + 1;  -- illegal
  X.all := 1;  -- illegal
  Dummy := X.all;  -- illegal
end Ownership_Transfer;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Ownership_Transfer
MD5: 951fe1c930d43a5009e607994ae0dd03

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

(continues on next page)

246 https://en.wikipedia.org/wiki/Pointer_(computer_programming)
Learning Ada

ownership_transfer.adb:11:06: error: dereference from "X" is not writable
ownership_transfer.adb:11:06: error: object was moved at line 9
ownership_transfer.adb:11:15: error: dereference from "X" is not readable
ownership_transfer.adb:11:15: error: object was moved at line 9
ownership_transfer.adb:12:06: error: dereference from "X" is not writable
ownership_transfer.adb:12:06: error: object was moved at line 9
ownership_transfer.adb:13:15: error: dereference from "X" is not readable
ownership_transfer.adb:13:15: error: object was moved at line 9
gnatprove: error during analysis of data and information flow

After the assignment of X to Y, variable X cannot be used anymore to read or write the underlying allocated memory.

**Note:** For more details on these limitations, see the SPARK User's Guide[247].

### 29.7 Designating SPARK Code

Since the SPARK language is restricted to only allow easily specifiable and verifiable constructs, there are times when you can't or don't want to abide by these limitations over your entire code base. Therefore, the SPARK tools only check conformance to the SPARK subset on code which you identify as being in SPARK.

You do this by using an aspect named SPARK_Mode. If you don't explicitly specify otherwise, SPARK_Mode is Off, meaning you can use the complete set of Ada features in that code and that it should not be analyzed by GNATprove. You can change this default either selectively (on some units or subprograms or packages inside units) or globally (using a configuration pragma, which is what we're doing in this tutorial). To allow simple reuse of existing Ada libraries, entities declared in imported units with no explicit SPARK_Mode can still be used from SPARK code. The tool only checks for SPARK conformance on the declaration of those entities which are actually used within the SPARK code.

Here's a common case of using the SPARK_Mode aspect:

```ada
package P
with SPARK_Mode => On
is
  -- package spec is IN SPARK, so can be used by SPARK clients
end P;

package body P
with SPARK_Mode => Off
is
  -- body is NOT IN SPARK, so is ignored by GNATprove
end P;
```

The package P only defines entities whose specifications are in the SPARK subset. However, it wants to use all Ada features in its body. Therefore the body should not be analyzed and has its SPARK_Mode aspect set to Off.

You can specify SPARK_Mode in a fine-grained manner on a per-unit basis. An Ada package has four different components: the visible and private parts of its specification and the declarative and statement parts of its body. You can specify SPARK_Mode as being either On or Off on any of those parts. Likewise, a subprogram has two parts: its specification and its body.

A general rule in SPARK is that once SPARK_MODE has been set to Off, it can never be switched On again in the same part of a package or subprogram. This prevents setting SPARK_MODE to On for local units of a unit with SPARK_MODE Off and switching back to SPARK_MODE On for a part of a given unit where it was set to Off in a previous part.

**Note:** For more details on the use of SPARK_MODE, see the SPARK User’s Guide\(^\text{248}\).

### 29.8 Code Examples / Pitfalls

#### 29.8.1 Example #1

Here’s a package defining an abstract stack type (defined as a private type in SPARK) of Element objects along with some subprograms providing the usual functionalities of stacks. It’s marked as being in the SPARK subset.

```ada
package Stack_Package
  with SPARK_MODE => On
is
  type Element is new Natural;
  type Stack is private;

  function Empty return Stack;
  procedure Push (S : in out Stack; E : Element);
  function Pop (S : in out Stack) return Element;

private
  type Stack is record
    Top : Integer;
    -- ...
  end record;
end Stack_Package;
```

---

29.8.2 Example #2

Let's turn to an abstract state machine version of a stack, where the unit provides a single instance of a stack. The content of the stack (global variables Content and Top) is not directly visible to clients. In this stripped-down version, only the function Pop is available to clients. The package spec and body are marked as being in the SPARK subset.

Listing 10: global_stack.ads

```ada
package Global_Stack
  with SPARK_Mode => On
is
  type Element is new Integer;
  function Pop return Element;
end Global_Stack;
```

Listing 11: global_stack.adb

```ada
package body Global_Stack
  with SPARK_Mode => On
is
  Max : constant Natural := 100;
  type Element_Array is array (1 .. Max) of Element;
  Content : Element_Array;
  Top : Natural;
  function Pop return Element is
    E : constant Element := Content (Top);
  begin
    Top := Top - 1;
    return E;
  end Pop;
end Global_Stack;
```

Code block metadata

Project: Courses.Intro.To_Spark.Overview.Example_02
MD5: 8c4eb564643eeef48264b5e43a6f580b9

Prover output

As above, functions should be free from side-effects. Here, Pop updates the global variable Top, which is not allowed in SPARK.
29.8.3 Example #3

We now consider two procedures: Permute and Swap. Permute applies a circular permutation to the value of its three parameters. Swap then uses Permute to swap the value of X and Y.

Listing 12: p.ads

```ada
package P
  with SPARK_Mode => On
is
  procedure Permute (X, Y, Z : in out Positive);
  procedure Swap (X, Y : in out Positive);
end P;
```

Listing 13: p.adb

```ada
package body P
  with SPARK_Mode => On
is
  procedure Permute (X, Y, Z : in out Positive) is
      Tmp : constant Positive := X;
  begin
      X := Y;
      Y := Z;
      Z := Tmp;
  end Permute;

  procedure Swap (X, Y : in out Positive) is
  begin
      Permute (X, Y, Y);
  end Swap;
end P;
```

Listing 14: test_swap.adb

```ada
with P; use P;

procedure Test_Swap
  with SPARK_Mode => On
is
  A : Integer := 1;
  B : Integer := 2;
begin
  Swap (A, B);
end Test_Swap;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_03
MD5: 0868a806061d86af4d2a03b1e7dc83c2

Build output

p.adb:14:19: error: writable actual for "Y" overlaps with actual for "Z"
gprbuild: *** compilation phase failed

Prover output

Phase 1 of 2: generation of Global contracts ...
p.adb:14:19: error: writable actual for "Y" overlaps with actual for "Z"
gnatprove: error during generation of Global contracts

29.8. Code Examples / Pitfalls
Learning Ada

Here, the values for parameters Y and Z are aliased in the call to Permute, which is not allowed in SPARK. In fact, in this particular case, this is even a violation of Ada rules so the same error is issued by the Ada compiler.

In this example, we see the reason why aliasing is not allowed in SPARK: since Y and Z are Positive, they are passed by copy and the result of the call to Permute depends on the order in which they're copied back after the call.

### 29.8.4 Example #4

Here, the Swap procedure is used to swap the value of the two record components of R.

**Listing 15: p.ads**

```ada
package P
   with SPARK_Mode => On
is
   type Rec is record
      F1 : Positive;
      F2 : Positive;
   end record;

   procedure Swap_Field (R : in out Rec);
   procedure Swap (X, Y : in out Positive);
end P;
```

**Listing 16: p.adb**

```ada
package body P
   with SPARK_Mode => On
is
   procedure Swap (X, Y : in out Positive) is
      Tmp : constant Positive := X;
   begin
      X := Y;
      Y := Tmp;
   end Swap;

   procedure Swap_Field (R : in out Rec) is
   begin
      Swap (R.F1, R.F2);
   end Swap_Field;
end P;
```

**Code block metadata**

- **Project:** Courses.Intro_To_Spark.Overview.Example_04
- **MD5:** ae4d3ebe8dd1a8f67f35cedffdea2ac9

**Prover output**

- Phase 1 of 2: generation of Global contracts ...
- Phase 2 of 2: analysis of data and information flow ...

This code is correct. The call to Swap is safe: two different components of the same record can't refer to the same object.
29.8.5 Example #5

Here's a slight modification of the previous example using an array instead of a record: Swap_Indexes calls Swap on values stored in the array A.

Listing 17: p.ads

```ada
package P
  with SPARK_Mode => On
is
  type P_Array is array (Natural range <>) of Positive;
  procedure Swap_Indexes (A : in out P_Array; I, J : Natural);
  procedure Swap (X, Y : in out Positive);
end P;
```

Listing 18: p.adb

```ada
package body P
  with SPARK_Mode => On
is
  procedure Swap (X, Y : in out Positive) is
    Tmp : constant Positive := X;
    begin
    X := Y;
    Y := Tmp;
    end Swap;
  procedure Swap_Indexes (A : in out P_Array; I, J : Natural) is
    begin
    Swap (A (I), A (J));
    end Swap_Indexes;
end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_05
MD5: 62a95179572e36443995f5a2d5ef08

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
p.adb:13:13: medium: formal parameters "X" and "Y" might be aliased (SPARK RM 6.4.2)
gnatprove: unproved check messages considered as errors

GNATprove detects a possible case of aliasing. Unlike the previous example, it has no way of knowing that the two elements A (I) and A (J) are actually distinct when we call Swap. GNATprove issues a check message here instead of an error, giving you the possibility of justifying the message after review (meaning that you've verified manually that this can't, in fact, occur).
29.8.6 Example #6

We now consider a package declaring a type Dictionary, an array containing a word per letter. The procedure Store allows us to insert a word at the correct index in a dictionary.

Listing 19: p.ads

```ada
with Ada.Finalization;

package P
  with SPARK_Mode => On
is
  subtype Letter is Character range 'a' .. 'z';
  type String_Access is new Ada.Finalization.Controlled with record
    Ptr : access String;
    end record;
  type Dictionary is array (Letter) of String_Access;
  procedure Store (D : in out Dictionary; W : String);
end P;
```

Listing 20: p.adb

```ada
package body P
  with SPARK_Mode => On
is
  procedure Store (D : in out Dictionary; W : String) is
    First_Letter : constant Letter := W (W'First);
  begin
    D (First_Letter).Ptr := new String'(W);
  end Store;
end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_06
MD5: 9175bcd1474e2143462b860c01d8602e

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
p.adb:7:07: error: "String_Access" is not allowed in SPARK (due to controlled types)
p.adb:7:07: error: violation of aspect SPARK_Mode at line 2
p.adb:7:31: error: borrow or observe of an expression which is not part of stand-alone object or parameter is not allowed in SPARK (SPARK RM 3.10(3))
p.adb:7:31: error: violation of aspect SPARK_Mode at line 2
p.ads:7:09: error: "Controlled" is not allowed in SPARK (due to controlled types)
p.ads:7:09: error: violation of aspect SPARK_Mode at line 4
p.ads:10:04: error: "String_Access" is not allowed in SPARK (due to controlled types)
p.ads:10:04: error: violation of aspect SPARK_Mode at line 4
gnatprove: error during analysis of data and information flow

This code is not correct: controlled types are not part of the SPARK subset. The solution here is to use SPARK_Mode to separate the definition of String_Access from the rest of the code in a fine grained manner.
29.8.7 Example #7

Here's a new version of the previous example, which we've modified to hide the controlled type inside the private part of package P, using pragma SPARK_Mode (Off) at the start of the private part.

Listing 21: p.ads

```ada
with Ada.Finalization;

package P
    with SPARK_Mode => On
is
    subtype Letter is Character range 'a' .. 'z';
    type String_Access is private;
    type Dictionary is array (Letter) of String_Access;
    function New_String_Access (W : String) return String_Access;
    procedure Store (D : in out Dictionary; W : String);

private
    pragma SPARK_Mode (Off);

    type String_Access is new Ada.Finalization.Controlled with record
       Ptr : access String;
    end record;

    function New_String_Access (W : String) return String_Access is
        (Ada.Finalization.Controlled with Ptr => new String'(W));
end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_07
MD5: cb04206c9734eb95f6444757d005dae2

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

Since the controlled type is defined and used inside of a part of the code ignored by GNAT-prove, this code is correct.

29.8.8 Example #8

Let's put together the new spec for package P with the body of P seen previously.

Listing 22: p.ads

```ada
with Ada.Finalization;

package P
    with SPARK_Mode => On
is
    subtype Letter is Character range 'a' .. 'z';
    type String_Access is private;
    type Dictionary is array (Letter) of String_Access;
```

(continues on next page)
function New_String_Access (W : String) return String_Access;

procedure Store (D : in out Dictionary; W : String);

private
pragma SPARK_Mode (Off);

type String_Access is new Ada.Finalization.Controlled with record
Ptr : access String;
end record;

function New_String_Access (W : String) return String_Access is
(Ada.Finalization.Controlled with Ptr => new String'(W));
end P;

Listing 23: p.adb

package body P
with SPARK_Mode => On
is
procedure Store (D : in out Dictionary; W : String) is
    First_Letter : constant Letter := W (W'First);
begin
    D (First_Letter) := New_String_Access (W);
end Store;
end P;

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_08
MD5: dacb2d50d0ddc6c620ee9945cb819369

Prover output

Phase 1 of 2: generation of Global contracts ...
p.adb:1:01: error: incorrect application of SPARK_Mode at /vagrant/frontend/dist/
test_output/projects/Courses/Intro_To_Spark/Overview/Example_08/
dacb2d50d0ddc6c620ee9945cb819369/main_spark.adc:12
p.adb:1:01: error: value Off was set for SPARK_Mode on "P" at p.ads:15
p.adb:2:08: error: incorrect use of SPARK_Mode
p.adb:2:08: error: value Off was set for SPARK_Mode on "P" at p.ads:15
gnatprove: error during generation of Global contracts

The body of Store doesn't actually use any construct that's not in the SPARK subset, but we
nevertheless can't set SPARK_Mode to On for P's body because it has visibility to P's private
part, which is not in SPARK, even if we don't use it.

29.8.9 Example #9

Next, we moved the declaration and the body of the procedure Store to another package
named Q.

Listing 24: p.ads

with Ada.Finalization;

package P
with SPARK_Mode => On
is
(continues on next page)
subtype Letter is Character range 'a' .. 'z';
type String_Access is private;
type Dictionary is array (Letter) of String_Access;

function New_String_Access (W : String) return String_Access;

private
  pragma SPARK_Mode (Off);
  type String_Access is new Ada.Finalization.Controlled with record
    Ptr : access String;
  end record;

function New_String_Access (W : String) return String_Access is
  (Ada.Finalization.Controlled with Ptr => new String'(W));
end P;

Listing 25: q.ads

with P; use P;
package Q
  with SPARK_Mode => On
is
  procedure Store (D : in out Dictionary; W : String);
end Q;

Listing 26: q.adb

package body Q
  with SPARK_Mode => On
is
  procedure Store (D : in out Dictionary; W : String) is
    First_Letter : constant Letter := W (W'First);
  begin
    D (First_Letter) := New_String_Access (W);
  end Store;
end Q;

And now everything is fine: we've managed to retain the use of the controlled type while having most of our code in the SPARK subset so GNATprove is able to analyze it.
29.8.10 Example #10

Our final example is a package with two functions to search for the value 0 inside an array A. The first raises an exception if 0 isn’t found in A while the other simply returns 0 in that case.

Listing 27: p.ads

```ada
package P with SPARK_Mode => On is
  type N_Array is array (Positive range <>) of Natural;
  Not_Found : exception;

  function Search_Zero_P (A : N_Array) return Positive;
  function Search_Zero_N (A : N_Array) return Natural;
end P;
```

Listing 28: p.adb

```ada
package body P with SPARK_Mode => On is

  function Search_Zero_P (A : N_Array) return Positive is
  begin
    for I in A’Range loop
      if A (I) = 0 then
        return I;
      end if;
    end loop;
    raise Not_Found;
  end Search_Zero_P;

  function Search_Zero_N (A : N_Array) return Natural with SPARK_Mode => Off is
  begin
    return Search_Zero_P (A);
  exception
    when Not_Found => return 0;
  end Search_Zero_N;
end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_10
MD5: 4b9656690ab1d42cebc72817f8a00637

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
gnatprove: unproved check messages considered as errors

This code is perfectly correct, despite the use of exception handling, because we’ve carefully isolated this non-SPARK feature in a function body marked with a SPARK_Mode of Off so it’s ignored by GNATprove. However, GNATprove tries to show that Not_Found is never raised in `Search_Zero_P`, producing a message about a possible exception being raised. Looking at `Search_Zero_N`, it’s indeed likely that an exception is meant to be raised in some cases, which means you need to verify that Not_Found is only raised when appropriate using other methods such as peer review or testing.
FLOW ANALYSIS

In this section we present the flow analysis capability provided by the GNATprove tool, a critical tool for using SPARK.

30.1 What does flow analysis do?

Flow analysis concentrates primarily on variables. It models how information flows through them during a subprogram's execution, connecting the final values of variables to their initial values. It analyzes global variables declared at library level, local variables, and formal parameters of subprograms.

Nesting of subprograms creates what we call scope variables: variables declared locally to an enclosing unit. From the perspective of a nested subprogram, scope variables look very much like global variables.

Flow analysis is usually fast, roughly as fast as compilation. It detects various types of errors and finds violations of some SPARK legality rules, such as the absence of aliasing and freedom of expressions from side-effects. We discussed these rules in the SPARK Overview (page 975).

Flow analysis is sound: if it doesn't detect any errors of a type it's supposed to detect, we know for sure there are no such errors.

30.2 Errors Detected

30.2.1 Uninitialized Variables

We now present each class of errors detected by flow analysis. The first is the reading of an uninitialized variable. This is nearly always an error: it introduces non-determinism and breaks the type system because the value of an uninitialized variable may be outside the range of its subtype. For these reasons, SPARK requires every variable to be initialized before being read.

Flow analysis is responsible for ensuring that SPARK code always fulfills this requirement. For example, in the function Max_Array shown below, we've neglected to initialize the value of Max prior to entering the loop. As a consequence, the value read by the condition of the if statement may be uninitialized. Flow analysis detects and reports this error.

Listing 1: show_uninitialized.ads

```
package Show_Uninitialized is
  type Array_Of_Naturals is array (Integer range <>) of Natural;
```

(continues on next page)
function Max_Array (A : Array_Of_Naturals) return Natural;
end Show_Uninitialized;

Listing 2: show_uninitialized.adb

package body Show_Uninitialized is

function Max_Array (A : Array_Of_Naturals) return Natural is
Max : Natural;
begin
  for I in A'Range loop
    if A (I) > Max then -- Here Max may not be initialized
      Max := A (I);
    end if;
  end loop;
  return Max;
end Max_Array;
end Show_Uninitialized;

30.2.2 Ineffective Statements

Ineffective statements are different than dead code: they're executed, and often even modify the value of variables, but have no effect on any of the subprogram's visible outputs: parameters, global variables or the function result. Ineffective statements should be avoided because they make the code less readable and more difficult to maintain.

More importantly, they're often caused by errors in the program: the statement may have been written for some purpose, but isn't accomplishing that purpose. These kinds of errors can be difficult to detect in other ways.

For example, the subprograms Swap1 and Swap2 shown below don't properly swap their two parameters X and Y. This error caused a statement to be ineffective. That ineffective statement is not an error in itself, but flow analysis produces a warning since it can be indicative of an error, as it is here.

So far, we've seen examples where flow analysis warns about ineffective statements and unused variables.
30.2.3 Incorrect Parameter Mode

Parameter modes are an important part of documenting the usage of a subprogram and affect the code generated for that subprogram. Flow analysis checks that each specified parameter mode corresponds to the usage of that parameter in the subprogram's body. It checks that an in parameter is never modified, either directly or through a subprogram call, checks that the initial value of an out parameter is never read in the subprogram (since it may not be defined on subprogram entry), and warns when an in out parameter isn't modified or when its initial value isn't used. All of these may be signs of an error.

We see an example below. The subprogram Swap is incorrect and GNATprove warns about an input which isn't read:

Listing 5: show_incorrect_param_mode.ads

```ada
package Show_Incorrect_Param_Mode is
  type T is new Integer;
  procedure Swap (X, Y : in out T);
end Show_Incorrect_Param_Mode;
```

Listing 6: show_incorrect_param_mode.adb

```ada
package body Show_Incorrect_Param_Mode is
  procedure Swap (X, Y : in out T) is
    Tmp : T := X;
    begin
      Y := X;  -- The initial value of Y is not used
      X := Tmp; -- Y is computed to be an out parameter
    end Swap;
end Show_Incorrect_Param_Mode;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Incorrect_Param_Mode
MD5: 1e33dbf461daab0daed01c83025232fc

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_incorrect_param_mode.ads:5:23: warning: unused initial value of "Y"

In SPARK, unlike Ada, you should declare an out parameter to be in out if it's not modified on every path, in which case its value may depend on its initial value. SPARK is stricter than Ada to allow more static detection of errors. This table summarizes SPARK's valid parameter modes as a function of whether reads and writes are done to the parameter.

<table>
<thead>
<tr>
<th>Initial value read</th>
<th>Written on some path</th>
<th>Written on every path</th>
<th>Parameter mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>in</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td>in out</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td>in out</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>in out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>out</td>
</tr>
</tbody>
</table>
30.3 Additional Verifications

30.3.1 Global Contracts

So far, none of the verifications we’ve seen require you to write any additional annotations. However, flow analysis also checks flow annotations that you write. In SPARK, you can specify the set of global and scoped variables accessed or modified by a subprogram. You do this using a contract named Global.

When you specify a Global contract for a subprogram, flow analysis checks that it’s both correct and complete, meaning that no variables other than those stated in the contract are accessed or modified, either directly or through a subprogram call, and that all those listed are accessed or modified. For example, we may want to specify that the function Get_Value_Of_X reads the value of the global variable X and doesn’t access any other global variable. If we do this through a comment, as is usually done in other languages, GNATprove can’t verify that the code complies with this specification:

```ada
package Show_Global_Contracts is
    X : Natural := 0;
    function Get_Value_Of_X return Natural;
        -- Get_Value_Of_X reads the value of the global variable X
end Show_Global_Contracts;
```

You write global contracts as part of the subprogram specification. In addition to their value in flow analysis, they also provide useful information to users of a subprogram. The value you specify for the Global aspect is an aggregate-like list of global variable names, grouped together according to their mode.

In the example below, the procedure Set_X_To_Y_Plus_Z reads both Y and Z. We indicate this by specifying them as the value for Input. It also writes X, which we specify using Output. Since Set_X_To_X_Plus_Y both writes X and reads its initial value, X’s mode is In_Out. Like parameters, if no mode is specified in a Global aspect, the default is Input. We see this in the case of the declaration of Get_Value_Of_X. Finally, if a subprogram, such as Incr_Parameter_X, doesn’t reference any global variables, you set the value of the global contract to null.

```
package Show_Global_Contracts is
    X, Y, Z : Natural := 0;
    function Get_Value_Of_X return Natural;
        -- Get_Value_Of_X reads the value of the global variable X
end Show_Global_Contracts;

procedure Set_X_To_Y_Plus_Z with
    Global => (Input => (Y, Z), -- reads values of Y and Z
                Output => X); -- modifies value of X

procedure Set_X_To_X_Plus_Y with
    Global => (Input => Y, -- reads value of Y
                In_Out => X); -- modifies value of X and
                -- also reads its initial value

function Get_Value_Of_X return Natural with
    Global => X; -- reads the value of the global variable X

procedure Incr_Parameter_X (X : in out Natural) with
    Global => null; -- do not reference any global variable
```

(continues on next page)
30.3.2 Depends Contracts

You may also supply a Depends contract for a subprogram to specify dependencies between its inputs and outputs. These dependencies include not only global variables but also parameters and the function's result. When you supply a Depends contract for a subprogram, flow analysis checks that it's correct and complete, that is, for each dependency you list, the variable depends on those listed and on no others.

For example, you may want to say that the new value of each parameter of Swap, shown below, depends only on the initial value of the other parameter and that the value of X after the return of Set_X_To_Zero doesn't depend on any global variables. If you indicate this through a comment, as you often do in other languages, GNATprove can't verify that this is actually the case.

```ada
package Show_Depends_Contracts is

   type T is new Integer;

   procedure Swap (X, Y : in out T);
   -- The value of X (resp. Y) after the call depends only
   -- on the value of Y (resp. X) before the call
   X : Natural;
   procedure Set_X_To_Zero;
   -- The value of X after the call depends on no input

end Show_Depends_Contracts;
```

Like Global contracts, you specify a Depends contract in subprogram declarations using an aspect. Its value is a list of one or more dependency relations between the outputs and inputs of the subprogram. Each relation is represented as two lists of variable names separated by an arrow. On the left of each arrow are variables whose final value depends on the initial value of the variables you list on the right.

For example, here we indicate that the final value of each parameter of Swap depends only on the initial value of the other parameter. If the subprogram is a function, we list its result as an output, using the Result attribute, as we do for Get_Value_0f_X below.

---

package Show_Depend_Contracts is
  
  type T is new Integer;

  X, Y, Z : T := 0;

  procedure Swap (X, Y : in out T) with
  Depends => (X => Y,
             -- X depends on the initial value of Y
             Y => X);
             -- Y depends on the initial value of X

  function Get_Value_Of_X return T with
  Depends => (Get_Value_Of_X'Result => X);
             -- result depends on the initial value of X

  procedure Set_X_To_Y_Plus_Z with
  Depends => (X => (Y, Z));
             -- X depends on the initial values of Y and Z

  procedure Set_X_To_X_Plus_Y with
  Depends => (X => + Y);
             -- X depends on Y and X's initial value

  procedure Do_Nothing (X : T) with
  Depends => (null => X);
             -- no output is affected by X

  procedure Set_X_To_Zero with
  Depends => (X => null);
             -- X depends on no input

end Show_Depend_Contracts;

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Depend_Contracts
MD5: 290866cc4208b6def7717a402bc2aef34

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

Often, the final value of a variable depends on its own initial value. You can specify this in a concise way using the + character, as we did in the specification of Set_X_To_X_Plus_Y above. If there's more than one variable on the left of the arrow, a + means each variable depends on itself, not that they all depend on each other. You can write the corresponding dependency with (==>+) or without (==>) whitespace.

If you have a program where an input isn't used to compute the final value of any output, you express that by writing null on the left of the dependency relation, as we did for the Do_Nothing subprogram above. You can only write one such dependency relation, which lists all unused inputs of the subprogram, and it must be written last. Such an annotation also silences flow analysis' warning about unused parameters. You can also write null on the right of a dependency relation to indicate that an output doesn't depend on any input. We do that above for the procedure Set_X_To_Zero.

30.3. Additional Verifications
Note: For more details on depends contracts, see the SPARK User's Guide. 

30.4 Shortcomings

30.4.1 Modularity

Flow analysis is sound, meaning that if it doesn't output a message on some analyzed SPARK code, you can be assured that none of the errors it tests for can occur in that code. On the other hand, flow analysis often issues messages when there are, in fact, no errors. The first, and probably most common reason for this relates to modularity.

To scale flow analysis to large projects, verifications are usually done on a per-subprogram basis, including detection of uninitialized variables. To analyze this modularly, flow analysis needs to assume the initialization of inputs on subprogram entry and modification of outputs during subprogram execution. Therefore, each time a subprogram is called, flow analysis checks that global and parameter inputs are initialized and each time a subprogram returns, it checks that global and parameter outputs were modified.

This can produce error messages on perfectly correct subprograms. An example is Set_X_To_Y_Plus_Z below, which only sets its out parameter X when Overflow is False.

Listing 9: set_x_to_y_plus_z.adb

```ada
procedure Set_X_To_Y_Plus_Z
    (Y, Z : Natural;
     X : out Natural;
     Overflow : out Boolean)
is
begin
    if Natural'Last - Z < Y
       then Overflow := True; -- X should be initialized on every path
    else Overflow := False;
        X := Y + Z;
    end if;
end Set_X_To_Y_Plus_Z;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Set_X_To_Y_Plus_Z
MD5: be47cd769d2a7267c0bd1bb2ef0d6328

Prover output

The message means that flow analysis wasn't able to verify that the program didn't read an uninitialized variable. To solve this problem, you can either set X to a dummy value when there's an overflow or manually verify that X is never used after a call to Set_X_To_Y_Plus_Z that returned True as the value of Overflow.

30.4.2 Composite Types

Another common cause of false alarms is caused by the way flow analysis handles composite types. Let's start with arrays.

Flow analysis treats an entire array as single object instead of one object per element, so it considers modifying a single element to be a modification of the array as a whole. Obviously, this makes reasoning about which global variables are accessed less precise and hence the dependencies of those variables are also less precise. This also affects the ability to accurately detect reads of uninitialized data.

It's sometimes impossible for flow analysis to determine if an entire array object has been initialized. For example, after we write code to initialize every element of an unconstrained array \( A \) in chunks, we may still receive a message from flow analysis claiming that the array isn't initialized. To resolve this issue, you can either use a simpler loop over the full range of the array, or (even better) an aggregate assignment, or, if that's not possible, verify initialization of the object manually.

Listing 10: show_composite_types_shortcoming.ads

```
package Show_Composite_Types_Shortcoming is
  type T is array (Natural range <> ) of Integer;
  procedure Init_Chunks (A : out T);
  procedure Init_Loop (A : out T);
  procedure Init_Aggregate (A : out T);
end Show_Composite_Types_Shortcoming;
```

Listing 11: show_composite_types_shortcoming.adb

```
package body Show_Composite_Types_Shortcoming is
  procedure Init_Chunks (A : out T) is
    begin
      A (A'First) := 0;
      for I in A'First + 1 .. A'Last loop
        A (I) := 0;
      end loop;
      -- flow analysis doesn't know that A is initialized
    end Init_Chunks;
  procedure Init_Loop (A : out T) is
    begin
      for I in A'Range loop
        A (I) := 0;
      end loop;
      -- flow analysis knows that A is initialized
    end Init_Loop;
  procedure Init_Aggregate (A : out T) is
    begin
      A := (others => 0);
      -- flow analysis knows that A is initialized
    end Init_Aggregate;
end Show_Composite_Types_Shortcoming;
```
Flow analysis is more precise on record objects because it tracks the value of each component of a record separately within a single subprogram. So when a record object is initialized by successive assignments of its components, flow analysis knows that the entire object is initialized. However, record objects are still treated as single objects when analyzed as an input or output of a subprogram.

Listing 12: show_record_flow_analysis.ads

```ada
package Show_Record_Flow_Analysis is

  type Rec is record
    F1 : Natural;
    F2 : Natural;
  end record;

  procedure Init (R : out Rec);

end Show_Record_Flow_Analysis;
```

Listing 13: show_record_flow_analysis.adb

```ada
package body Show_Record_Flow_Analysis is

  procedure Init (R : out Rec) is
  begin
    R.F1 := 0;
    R.F2 := 0;
    -- R is initialized
  end Init;

end Show_Record_Flow_Analysis;
```

Flow analysis complains when a procedure call initializes only some components of a record object. It'll notify you of uninitialized components, as we see in subprogram Init_F2 below.
Listing 14: show_record_flow_analysis.ads

```ada
package Show_Record_Flow_Analysis is

  type Rec is record
    F1 : Natural;
    F2 : Natural;
  end record;

  procedure Init (R : out Rec);
  procedure Init_F2 (R : in out Rec);

end Show_Record_Flow_Analysis;
```

Listing 15: show_record_flow_analysis.adb

```ada
package body Show_Record_Flow_Analysis is

  procedure Init_F2
  (R : in out Rec) is
  begin
    R.F2 := 0;
  end Init_F2;

  procedure Init (R : out Rec) is
  begin
    R.F1 := 0;
    Init_F2 (R); -- R should be initialized before this call
  end Init;

end Show_Record_Flow_Analysis;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.Flow_Analysis.Record_Flow_Analysis_2

MD5: efeecb787bf9d68977ed9701689cd6c4

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_record_flow_analysis.adb:12:16: high: "R.F2" is not initialized

**30.4.3 Value Dependency**

Flow analysis is not value-dependent: it never reasons about the values of expressions, only whether they have been set to some value or not. As a consequence, if some execution path in a subprogram is impossible, but the impossibility can only be determined by looking at the values of expressions, flow analysis still considers that path feasible and may emit messages based on it believing that execution along such a path is possible.

For example, in the version of Absolute_Value below, flow analysis computes that R is uninitialized on a path that enters neither of the two conditional statements. Because it doesn't consider values of expressions, it can't know that such a path is impossible.
Learning Ada

Listing 16: absolute_value.adb

```ada
procedure Absolute_Value
begin
  begin
    if X < 0 then
      R := -X;
    end if;
    if X >= 0 then
      R := X;
    end if;
  end;
end Absolute_Value;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Absolute_Value_1
MD5: 69c233d22afdfdac679bf379b353a8d4

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
absolute_value.adb:3:04: medium: "R" might not be initialized in "Absolute_Value". [reason for check: OUT parameter should be initialized on return] [possible fix: initialize "R" on all paths or make "R" an IN OUT parameter]
gnatprove: unproved check messages considered as errors

To avoid this problem, you should make the control flow explicit, as in this second version of Absolute_Value:

Listing 17: absolute_value.adb

```ada
procedure Absolute_Value
begin
  begin
    if X < 0 then
      R := -X;
    else
      R := X;
    end if;
  end Absoute_Value;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Absolute_Value_2
MD5: 9c773547f81e02a7aa1b5532b05937

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
30.4.4 Contract Computation

The final cause of unexpected flow messages that we'll discuss also comes from inaccuracy in computations of contracts. As we explained earlier, both Global and Depends contracts are optional, but GNATprove uses their data for some of its analysis.

For example, flow analysis can't detect reads from uninitialized variables without knowing the set of variables accessed. It needs to analyze and check both the Depends contracts you wrote for a subprogram and those you wrote for callers of that subprogram. Since each flow contract on a subprogram depends on the flow contracts of all the subprograms called inside its body, this computation can often be quite time-consuming. Therefore, flow analysis sometimes trades-off the precision of this computation against the time a more precise computation would take.

This is the case for Depends contracts, where flow analysis simply assumes the worst, that each subprogram's output depends on all of that subprogram's inputs. To avoid this assumption, all you have to do is supply contracts when default ones are not precise enough. You may also want to supply Global contracts to further speed up flow analysis on larger programs.

30.5 Code Examples / Pitfalls

30.5.1 Example #1

The procedure Search_Array searches for an occurrence of element E in an array A. If it finds one, it stores the index of the element in Result. Otherwise, it sets Found to False.

**Listing 18: show_search_array.ads**

```ada
package Show_Search_Array is

  type Array_Of_Positives is array (Natural range <>) of Positive;

  procedure Search_Array
    (A : Array_Of_Positives; E : Positive; Result : out Integer; Found : out Boolean);

end Show_Search_Array;
```

**Listing 19: show_search_array.adb**

```ada
package body Show_Search_Array is

  procedure Search_Array
    (A : Array_Of_Positives; E : Positive; Result : out Integer; Found : out Boolean) is
  begin
    for I in A'Range loop
      if A (I) = E then
        Result := I;
        Found := True;
        return;
      end if;
    end loop;
  end if;
end loop;
```

(continues on next page)
Learning Ada

(continued from previous page)

    Found := False;
end Search_Array;
end Show_Search_Array;

Code block metadata

Project: Courses.Intro.To_Spark.Flow Analysis.Example_01
MD5: d2a27a5bde247767e2f6cd2d42a2d629

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_search_array.ads:8:07: medium: "Result" might not be initialized in "Search_..."
  [reason for check: OUT parameter should be initialized on return]
  [possible fix: initialize "Result" on all paths or make "Result" an IN OUT parameter]
gnatprove: unproved check messages considered as errors

GNATprove produces a message saying that Result is possibly uninitialized on return. There are perfectly legal uses of the function Search_Array, but flow analysis detects that Result is not initialized on the path that falls through from the loop. Even though this program is correct, you shouldn’t ignore the message: it means flow analysis cannot guarantee that Result is always initialized at the call site and so assumes any read of Result at the call site will read initialized data. Therefore, you should either initialize Result when Found is false, which silences flow analysis, or verify this assumption at each call site by other means.

30.5.2 Example #2

To avoid the message previously issued by GNATprove, we modify Search_Array to raise an exception when E isn’t found in A:

Listing 20: show_search_array.ads

    package Show_Search_Array is
    
      type Array_Of_Positives is array (Natural range <>) of Positive;

      Not_Found : exception;

      procedure Search_Array
        (A : Array_Of_Positives;
         E : Positive;
         Result : out Integer);
end Show_Search_Array;

Listing 21: show_search_array.adb

    package body Show_Search_Array is
    
      procedure Search_Array
        (A : Array_Of_Positives;
         E : Positive;
         Result : out Integer) is
      begin
        for I in A’Range loop
          (continues on next page)
if A (I) = E then
    Result := I;
    return;
end if;
end loop;
raise Not_Found;
end Search_Array;
end Show_Search_Array;

Code block metadata
Project: Courses.Intro_To_Spark.Flow_Analysis.Example_02
MD5: fa159faeb68974b1af3de2112e086b16

Prover output
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_search_array.adb:14:07: medium: exception might be raised
gnatprove: unproved check messages considered as errors

Flow analysis doesn't emit any messages in this case, meaning it can verify that Result can't be read in SPARK code while uninitialized. But why is that, since Result is still not initialized when E is not in A? This is because the exception, Not_Found, can never be caught within SPARK code (SPAK doesn't allow exception handlers). However, the GNATprove tool also tries to ensure the absence of runtime errors in SPARK code, so tries to prove that Not_Found is never raised. When it can't do that here, it produces a different message.

30.5.3 Example #3

In this example, we're using a discriminated record for the result of Search_Array instead of conditionally raising an exception. By using such a structure, the place to store the index at which E was found exists only when E was indeed found. So if it wasn't found, there's nothing to be initialized.

Listing 22: show_search_array.ads
package Show_Search_Array is
    type Array_Of_Positives is array (Natural range <>) of Positive;

    type Search_Result (Found : Boolean := False) is record
        case Found is
            when True =>
                Content : Integer;
            when False => null;
        end case;
    end record;

    procedure Search_Array
    (A : Array_Of_Positives;
    E : Positive;
    Result : out Search_Result)
    with Pre => not Result'Constrained;
end Show_Search_Array;
Learning Ada

Listing 23: show_search_array.adb

```ada
package body Show_Search_Array is
  procedure Search_Array
  (A : Array_Of_Positives;
   E : Positive;
   Result : out Search_Result) is
  begin
    for I in A'Range loop
      if A (I) = E then
        Result := (Found => True,
                   Content => I);
        return;
      end if;
    end loop;
    Result := (Found => False);
  end Search_Array;
end Show_Search_Array;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_03
MD5: 1d5ec5d78185fd75499b90b3d21f8ae2

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_search_array.adb:10:20: info: discriminant check proved
show_search_array.adb:15:14: info: discriminant check proved
show_search_array.ads:16:07: info: initialization of "Result" proved

This example is correct and flow analysis doesn't issue any message: it can verify both that no uninitialized variables are read in Search_Array's body, and that all its outputs are set on return. We've used the attribute Constrained in the precondition of Search_Array to indicate that the value of the Result in argument can be set to any variant of the record type Search_Result, specifically to either the variant where E was found and where it wasn't.

30.5.4 Example #4

The function Size_Of_Biggest_Increasing_Sequence is supposed to find all sequences within its parameter A that contain elements with increasing values and returns the length of the longest one. To do this, it calls a nested procedure Test_Index iteratively on all the elements of A. Test_Index checks if the sequence is still increasing. If so, it updates the largest value seen so far in this sequence. If not, it means it's found the end of a sequence, so it computes the size of that sequence and stores it in Size_Of_Seq.

Listing 24: show_biggest_increasing_sequence.ads

```ada
package Show_Biggest_Increasing_Sequence is
  type Array_Of_Positives is array (Integer range <>) of Positive;
  function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
    return Natural;
(continues on next page)
```

1008 Chapter 30. Flow Analysis
end Show_Biggest_Increasing_Sequence;

Listing 25: show_biggest_increasing_sequence.adb

package body Show_Biggest_Increasing_Sequence is

function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives) return Natural is
  Max : Natural;
  End_Of_Seq : Boolean;
  Size_Of_Seq : Natural;
  Beginning : Integer;

  procedure Test_Index (Current_Index : Integer) is
    begin
      if A (Current_Index) >= Max then
        Max := A (Current_Index);
        End_Of_Seq := False;
      else
        Max := 0;
        End_Of_Seq := True;
        Size_Of_Seq := Current_Index - Beginning;
        Beginning := Current_Index;
      end if;
    end Test_Index;

  Biggest_Seq : Natural := 0;

  begin
    for I in A'Range loop
      Test_Index (I);
      if End_Of_Seq then
        Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
      end if;
    end loop;
    return Biggest_Seq;
  end Size_Of_Biggest_Increasing_Sequence;
end Show_Biggest_Increasing_Sequence;

Code block metadata

Project: Courses.Intro.To_Spark.Flow_Analysis.Example_04
MD5: e6083665827d9dee4e00bdce48e962f

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_biggest_increasing_sequence.adb:13:34: medium: "Max" might not be initialized,
in call inlined at show_biggest_increasing_sequence.adb:28
show_biggest_increasing_sequence.adb:19:44: medium: "Beginning" might not be initialized,
in call inlined at show_biggest_increasing_sequence.adb:28
show_biggest_increasing_sequence.adb:30:41: medium: "Size_Of_Seq" might not be initialized
gnatprove: unproved check messages considered as errors

However, this example is not correct. Flow analysis emits messages for Test_Index stating that Max, Beginning, and Size_Of_Seq should be initialized before being read. Indeed,
when you look carefully, you see that both Max and Beginning are missing initializations because they are read in Test_Index before being written. As for Size_Of_Seq, we only read its value when End_Of_Seq is true, so it actually can’t be read before being written, but flow analysis isn’t able to verify its initialization by using just flow information.

The call to Test_Index is automatically inlined by GNATprove, which leads to another messages above. If GNATprove couldn’t inline the call to Test_Index, for example if it was defined in another unit, the same messages would be issued on the call to Test_Index.

### 30.5.5 Example #5

In the following example, we model permutations as arrays where the element at index I is the position of the I'th element in the permutation. The procedure Init initializes a permutation to the identity, where the I'th element's is at the I'th position. Cyclic_Permutation calls Init and then swaps elements to construct a cyclic permutation.

#### Listing 26: show_permutation.ads

```adl
package Show_Permutation is

    type Permutation is array (Positive range <>) of Positive;

    procedure Swap (A : in out Permutation;
                     I, J : Positive);

    procedure Init (A : out Permutation);

    function Cyclic_Permutation (N : Natural) return Permutation;

end Show_Permutation;
```

#### Listing 27: show_permutation.adb

```adl
package body Show_Permutation is

    procedure Swap (A : in out Permutation;
                    I, J : Positive)
    is
        Tmp : Positive := A (I);
    begin
        A (I) := A (J);
        A (J) := Tmp;
    end Swap;

    procedure Init (A : out Permutation) is
    begin
        A (A'First) := A'First;
        for I in A'First + 1 .. A'Last loop
            A (I) := I;
        end loop;
    end Init;

    function Cyclic_Permutation (N : Natural) return Permutation is
    begin
        A : Permutation (1 .. N);
        Init (A);
        for I in A'First .. A'Last - 1 loop
            Swap (A, I, I + 1);
        end loop;
    return A;
```

(continues on next page)
Learning Ada

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_05
MD5: 219b06617c636c18543128d77f90fcee

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_permutation.ads:8:20: medium: "A" might not be initialized in "Init" [reason:
 → for check: OUT parameter should be fully initialized on return] [possible fix:
 → initialize "A" on all paths, make "A" an IN OUT parameter or annotate it with
 → aspect Relaxed Initialization]
gnatprove: unproved check messages considered as errors

This program is correct. However, flow analysis will nevertheless still emit messages because it can't verify that every element of A is initialized by the loop in Init. This message is a false alarm. You can either ignore it or justify it safely.

30.5.6 Example #6

This program is the same as the previous one except that we've changed the mode of A in the specification of Init to **in out** to avoid the message from flow analysis on array assignment.

Listing 28: show_permutation.ads

```ada
package Show_Permutation is
  type Permutation is array (Positive range <>) of Positive;
  procedure Swap (A : in out Permutation;
                  I, J : Positive);
  procedure Init (A : in out Permutation);
  function Cyclic_Permutation (N : Natural) return Permutation;
end Show_Permutation;
```

Listing 29: show_permutation.adb

```ada
package body Show_Permutation is
  procedure Swap (A : in out Permutation;
                  I, J : Positive)
  is
    Tmp : Positive := A (I);
  begin
    A (I) := A (J);
    A (J) := Tmp;
  end Swap;
  procedure Init (A : in out Permutation) is
```

(continues on next page)
begin
  A (A'First) := A'First;
  for I in A'First + 1 .. A'Last loop
    A (I) := I;
  end loop;
end Init;

function Cyclic_Permutation (N : Natural) return Permutation is
  A : Permutation (1 .. N);
begin
  Init (A);
  for I in A'First .. A'Last - 1 loop
    Swap (A, I, I + 1);
  end loop;
  return A;
end Cyclic_Permutation;

end Show_Permutation;

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_06
MD5: 61406d9a66dda71630c74c12f3d67936

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_permutation.adb:23:13: high: "A" is not initialized
gnatprove: unproved check messages considered as errors

This program is not correct. Changing the mode of a parameter that should really be out to in out to silence a false alarm is not a good idea. Not only does this obfuscate the specification of Init, but flow analysis emits a message on the procedure where A is not initialized, as shown by the message in Cyclic_Permutation.

30.5.7 Example #7

Incr_Step_Function takes an array A as an argument and iterates through A to increment every element by the value of Increment, saturating at a specified threshold value. We specified a Global contract for Incr_Until_Threshold.

Listing 30: show_increments.ads

package Show_Increments is
  type Array_Of_Positives is array (Natural range <>) of Positive;
  Increment : constant Natural := 10;
  procedure Incr_Step_Function (A : in out Array_Of_Positives);
end Show_Increments;

Listing 31: show_increments.adb

package body Show_Increments is

(continues on next page)
procedure Incr_Step_Function (A : in out Array_Of_Positives) is

Threshold : Positive := Positive'Last;

procedure Incr_Until_Threshold (I : Integer) with
  Global => (Input => Threshold,
  In_Out => A);

procedure Incr_Until_Threshold (I : Integer) is
begin
  if Threshold - Increment <= A (I) then
    A (I) := Threshold;
  else
    A (I) := A (I) + Increment;
  end if;
end Incr_Until_Threshold;

begin
  for I in A'Range loop
    if I > A'First then
      Threshold := A (I - 1);
    end if;
    Incr_Until_Threshold (I);
  end loop;
end Incr_Step_Function;

end Show_Increments;

30.5.8 Example #8

We now go back to the procedure Test_Index from Example #4 (page 1008) and correct the missing initializations. We want to know if the Global contract of Test_Index is correct.

Listing 32: show_biggest_increasing_sequence.ads

package Show_Biggest_Increasing_Sequence is

  type Array_Of_Positives is array (Integer range <>) of Positive;

  function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
    return Natural;

(continues on next page)
end Show_Biggest_Increasing_Sequence;

Listing 33: show_biggest_increasing_sequence.adb

package body Show_Biggest_Increasing_Sequence is

function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
return Natural
is
    Max : Natural := 0;
    End_Of_Seq : Boolean;
    Size_Of_Seq : Natural := 0;
    Beginning : Integer := A'First - 1;

    procedure Test_Index (Current_Index : Integer) with
    Global => (In_Out => (Beginning, Max, Size_Of_Seq),
      Output => End_Of_Seq,
      Input => Current_Index)
    is
        begin
            if A (Current_Index) >= Max then
                Max := A (Current_Index);
                End_Of_Seq := False;
            else
                Max := 0;
                End_Of_Seq := True;
                Size_Of_Seq := Current_Index - Beginning;
                Beginning := Current_Index;
            end if;
        end Test_Index;

    Biggest_Seq : Natural := 0;

        begin
            for I in A'Range loop
                Test_Index (I);
                if End_Of_Seq then
                    Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
                end if;
            end loop;
        return Biggest_Seq;
    end Size_Of_Biggest_Increasing_Sequence;

end Show_Biggest_Increasing_Sequence;

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_08
MD5: 86fb934c32a38f6841ef736780b2e3b2

Prover output

Phase 1 of 2: generation of Global contracts ... show_biggest_increasing_sequence.adb:14:30: error: global item cannot reference parameter of subprogram "Test Index"
gnatprove: error during generation of Global contracts

The contract in this example is not correct: Current_Index is a parameter of Test_Index, so we shouldn't reference it as a global variable. Also, we should have listed variable A from the outer scope as an Input in the Global contract.
### 30.5.9 Example #9

Next, we change the Global contract of Test_Index into a Depends contract. In general, we don’t need both contracts because the set of global variables accessed can be deduced from the Depends contract.

**Listing 34: show_biggest_increasing_sequence.ads**

```ada
package Show_Biggest_Increasing_Sequence is

  type Array_Of_Positives is array (Integer range <>) of Positive;

  function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
    return Natural;

end Show_Biggest_Increasing_Sequence;
```

**Listing 35: show_biggest_increasing_sequence.adb**

```ada
package body Show_Biggest_Increasing_Sequence is

  function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
    return Natural is
      Max : Natural := 0;
      End_Of_Seq : Boolean;
      Size_Of_Seq : Natural := 0;
      Beginning : Integer := A'First - 1;

      procedure Test_Index (Current_Index : Integer) with
        Depends => (Max, End_Of_Seq) => (A, Current_Index, Max),
                    (Size_Of_Seq, Beginning) =>
                      + (A, Current_Index, Max, Beginning))
        is
          begin
            if A (Current_Index) >= Max then
              Max := A (Current_Index);
              End_Of_Seq := False;
            else
              Max := 0;
              End_Of_Seq := True;
              Size_Of_Seq := Current_Index - Beginning;
              Beginning := Current_Index;
            end if;
            end Test_Index;

            Biggest_Seq : Natural := 0;

            begin
              for I in A'Range loop
                Test_Index (I);
                if End_Of_Seq then
                  Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
                end if;
              end loop;
              return Biggest_Seq;
            end Size_Of_Biggest_Increasing_Sequence;

end Show_Biggest_Increasing_Sequence;
```
Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_biggest_increasing_sequence.adb:7:07: info: initialization of "End_Of_Seq"_proved
show_biggest_increasing_sequence.adb:11:17: info: initialization of "End_Of_Seq"_proved
show_biggest_increasing_sequence.adb:12:09: info: flow dependencies proved

This example is correct. Some of the dependencies, such as Size_Of_Seq depending on Beginning, come directly from the assignments in the subprogram. Since the control flow influences the final value of all of the outputs, the variables that are being read, A, Current_Index, and Max, are present in every dependency relation. Finally, the dependencies of Size_Of_Seq and Beginning on themselves are because they may not be modified by the subprogram execution.

30.5.10 Example #10

The subprogram Identity swaps the value of its parameter two times. Its Depends contract says that the final value of X only depends on its initial value and likewise for Y.

Listing 36: show_swap.ads

```ada
package Show_Swap is
  procedure Swap (X, Y : in out Positive);
  procedure Identity (X, Y : in out Positive) with
    Depends => (X => X,
                Y => Y);
end Show_Swap;
```

Listing 37: show_swap.adb

```ada
package body Show_Swap is
  procedure Swap (X, Y : in out Positive) is
    Tmp : constant Positive := X;
  begin
    X := Y;
    Y := Tmp;
  end Swap;
  procedure Identity (X, Y : in out Positive) is
  begin
    Swap (X, Y);
    Swap (Y, X);
  end Identity;
end Show_Swap;
```
**Prover output**

<table>
<thead>
<tr>
<th>Phase 1 of 2: generation of Global contracts ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 2 of 2: analysis of data and information flow ...</td>
</tr>
<tr>
<td>show_swap.ads:6:18: medium: missing dependency &quot;X =&gt; Y&quot;</td>
</tr>
<tr>
<td>show_swap.ads:7:18: medium: missing dependency &quot;Y =&gt; X&quot;</td>
</tr>
<tr>
<td>gnatprove: unproved check messages considered as errors</td>
</tr>
</tbody>
</table>

This code is correct, but flow analysis can't verify the Depends contract of Identity because we didn't supply a Depends contract for Swap. Therefore, flow analysis assumes that all outputs of Swap, X and Y, depend on all its inputs, both X and Y’s initial values. To prevent this, we should manually specify a Depends contract for Swap.
This section presents the proof capability of GNATprove, a major tool for the SPARK language. We focus here on the simpler proofs that you'll need to write to verify your program's integrity. The primary objective of performing proof of your program's integrity is to ensure the absence of runtime errors during its execution.

The analysis steps discussed here are only sound if you've previously performed Flow Analysis (page 993). You shouldn't proceed further if you still have unjustified flow analysis messages for your program.

### 31.1 Runtime Errors

There's always the potential for errors that aren't detected during compilation to occur during a program's execution. These errors, called runtime errors, are those targeted by GNATprove.

There are various kinds of runtime errors, the most common being references that are out of the range of an array (buffer overflow in Ada), subtype range violations, overflows in computations, and divisions by zero. The code below illustrates many examples of possible runtime errors, all within a single statement. Look at the assignment statement setting the \( I + J \)'th cell of an array \( A \) to the value \( P / Q \).

**Listing 1: show_runtime_errors.ads**

```ada
package Show_Runtime_Errors is
  type Nat_Array is array (Integer range <>) of Natural;
  procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);
end Show_Runtime_Errors;
```

**Listing 2: show_runtime_errors.adb**

```ada
package body Show_Runtime_Errors is
  procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
  begin
    A (I + J) := P / Q;
  end Update;
end Show_Runtime_Errors;
```

There are quite a number of errors that may occur when executing this code. If we don't know anything about the values of \( I, J, P, \) and \( Q \), we can't rule out any of those errors.

First, the computation of \( I + J \) can overflow, for example if \( I \) is \texttt{Integer'}Last and \( J \) is positive.

\[
A (\texttt{Integer'}Last + 1) := P / Q;
\]

Next, the sum, which is used as an array index, may not be in the range of the index of the array.

\[
A (A'Last + 1) := P / Q;
\]

On the other side of the assignment, the division may also overflow, though only in the very special case where \( P \) is \texttt{Integer'}First and \( Q \) is \(-1\) because of the asymmetric range of signed integer types.

\[
A (I + J) := \texttt{Integer'}First / -1;
\]

The division is also not allowed if \( Q \) is 0.

\[
A (I + J) := P / 0;
\]

Finally, since the array contains natural numbers, it's also an error to store a negative value in it.

\[
A (I + J) := 1 / -1;
\]

The compiler generates checks in the executable code corresponding to each of those runtime errors. Each check raises an exception if it fails. For the above assignment statement, we can see examples of exceptions raised due to failed checks for each of the different cases above.
Learning Ada

These runtime checks are costly, both in terms of program size and execution time. It may be appropriate to remove them if we can statically ensure they aren't needed at runtime, in other words if we can prove that the condition tested for can never occur.

This is where the analysis done by GNATprove comes in. It can be used to demonstrate statically that none of these errors can ever occur at runtime. Specifically, GNATprove logically interprets the meaning of every instruction in the program. Using this interpretation, GNATprove generates a logical formula called a verification condition for each check that would otherwise be required by the Ada (and hence SPARK) language.

GNATprove then passes these verification conditions to an automatic prover, stated as conditions that must be true to avoid the error. If every such condition can be validated by a prover (meaning that it can be mathematically shown to always be true), we've been able to prove that no error can ever be raised at runtime when executing that program.

31.2 Modularity

To scale to large programs, GNATprove performs proofs on a per-subprogram basis by relying on preconditions and postconditions to properly summarize the input and output state of each subprogram. More precisely, when verifying the body of a subprogram, GNATprove assumes it knows nothing about the possible initial values of its parameters and of the global variables it accesses except what you state in the subprogram's precondition. If you don't specify a precondition, it can't make any assumptions.

For example, the following code shows that the body of Increment can be successfully verified: its precondition constrains the value of its parameter X to be less than Integer'Last so we know the overflow check is always false.
In the same way, when a subprogram is called, GNATprove assumes its out and in out parameters and the global variables it writes can be modified in any way compatible with their postconditions. For example, since Increment has no postcondition, GNATprove doesn't know that the value of X after the call is always less than Integer'Last. Therefore, it can't prove that the addition following the call to Increment can't overflow.

**Listing 3: show_modularity.adb**

```ada
procedure Show_Modularity is

   procedure Increment (X : in out Integer) with
     Pre => X < Integer'Last is
     begin
       X := X + 1;
       -- info: overflow check proved
     end Increment;

   begin
     X : Integer;
     begin
       X := Integer'Last - 2;
       Increment (X);
       -- After the call, GNATprove no longer knows the value of X
       X := X + 1;
       -- medium: overflow check might fail
     end Show_Modularity;

end Show_Modularity;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Modularity_1
MD5: ca8ff8d29792fd5a06f7cb0158e13609

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_modularity.adb:6:14: info: overflow check proved
show_modularity.adb:10:04: info: initialization of "X" proved
show_modularity.adb:13:04: info: precondition proved
show_modularity.adb:16:11: medium: overflow check might fail, cannot prove upper bound for X + 1 [reason for check: result of addition must fit in a 32-bits machine integer] [possible fix: call at line 13 should mention X (for argument X) in a postcondition]
gnatprove: unproved check messages considered as errors

### 31.2.1 Exceptions

There are two cases where GNATprove doesn't require modularity and hence doesn't make the above assumptions. First, local subprograms without contracts can be inlined if they're simple enough and are neither recursive nor have multiple return points. If we remove the contract from Increment, it fits the criteria for inlining.

**Listing 4: show_modularity.adb**

```ada
procedure Show_Modularity is

   procedure Increment (X : in out Integer) is
     begin
       X := X + 1;
       -- info: overflow check proved, in call inlined at...

end Show_Modularity;
```

(continues on next page)
end Increment;

X : Integer;

begin
  X := Integer'Last - 2;
  Increment (X);
  X := X + 1;
  -- info: overflow check proved
end Show_Modularity;

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Modularity_2
MD5: 448d576897c3e4606cd4b90621aad63a

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_modularity.adb:5:14: info: overflow check proved, in call inlined at show_modularity.adb:12
show_modularity.adb:9:04: info: initialization of "X" proved
show_modularity.adb:13:11: info: overflow check proved

GNATprove now sees the call to Increment exactly as if the increment on X was done outside that call, so it can successfully verify that neither addition can overflow.

Note: For more details on contextual analysis of subprograms, see the SPARK User's Guide253.

The other case involves functions. If we define a function as an expression function, with or without contracts, GNATprove uses the expression itself as the postcondition on the result of the function.

In our example, replacing Increment with an expression function allows GNATprove to successfully verify the overflow check in the addition.

Listing 5: show_modularity.adb

procedure Show_Modularity is
  function Increment (X : Integer) return Integer is
    (X + 1)
    -- info: overflow check proved
    with Pre => X < Integer'Last;
    X : Integer;
  begin
    X := Integer'Last - 2;
    X := Increment (X);
    X := X + 1;
    -- info: overflow check proved
end Show_Modularity;

Code block metadata


31.2. Modularity 1023
### 31.3 Contracts

Ada contracts are perfectly suited for formal verification, but are primarily designed to be checked at runtime. When you specify the -gnata switch, the compiler generates code that verifies the contracts at runtime. If an Ada contract isn't satisfied for a given subprogram call, the program raises the Assert_Failure exception. This switch is particularly useful during development and testing, but you may also retain run-time execution of assertions, and specifically preconditions, during the program's deployment to avoid an inconsistent state.

Consider the incorrect call to `Increment` below, which violates its precondition. One way to detect this error is by compiling the function with assertions enabled and testing it with inputs that trigger the violation. Another way, one that doesn't require guessing the needed inputs, is to run GNATprove.

**Listing 6: show_precondition_violation.adb**

```ada
procedure Show_Precondition_Violation is

  procedure Increment (X : in out Integer) with
    Pre => X < Integer'Last is

begin
  X := X + 1;
end Increment;

begin
  X := Integer'Last;
  Increment (X);
end Show_Precondition_Violation;
```

---

**Note:** For more details on expression functions, see the SPARK User's Guide[^1].

Similarly, consider the incorrect implementation of function Absolute below, which violates its postcondition. Likewise, one way to detect this error is by compiling the function with assertions enabled and testing with inputs that trigger the violation. Another way, one which again doesn't require finding the inputs needed to demonstrate the error, is to run GNATprove.

**Listing 7: show_postcondition_violation.adb**

```ada
procedure show_postcondition_violation is
  procedure Absolute (X : in out Integer) with
  Post => X > 0 => 0 is
  begin
    if X > 0 then
      X := -X;
    end if;
  end Absolute;
  X : Integer;
begin
  X := 1;
  Absolute (X);
end show_postcondition_violation;
```

The benefits of dynamically checking contracts extends beyond making testing easier. Early failure detection also allows an easier recovery and facilitates debugging, so you may want to enable these checks at runtime to terminate execution before some damaging or hard-to-debug action occurs.

GNATprove statically analyses preconditions and postconditions. It verifies preconditions every time a subprogram is called, which is the runtime semantics of contracts. Postconditions, on the other hand, are verified once as part of the verification of the subprogram's
body. For example, GNATprove must wait until Increment is improperly called to detect the precondition violation, since a precondition is really a contract for the caller. On the other hand, it doesn't need Absolute to be called to detect that its postcondition doesn't hold for all its possible inputs.

**Note:** For more details on pre and postconditions, see the SPARK User's Guide255.

### 31.3.1 Executable Semantics

Expressions in Ada contracts have the same semantics as Boolean expressions elsewhere, so runtime errors can occur during their computation. To simplify both debugging of assertions and combining testing and static verification, the same semantics are used by GNATprove.

While proving programs, GNATprove verifies that no error can ever be raised during the execution of the contracts. However, you may sometimes find those semantics too heavy, in particular with respect to overflow checks, because they can make it harder to specify an appropriate precondition. We see this in the function Add below.

```
procedure Show_Executable_Semantics
with SPARK_Mode => On
is
  function Add (X, Y : Integer) return Integer is (X + Y)
    with Pre => X + Y in Integer;

  X : Integer;
begin
  X := Add (Integer'Last, 1);
end Show_Executable_Semantics;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Executable_Semantics
MD5: d85fa0507d7c35fb98ade7815020117e

**Build output**

show_executable_semantics.adb:5:24: warning: explicit membership test may be optimized away [enabled by default]
show_executable_semantics.adb:5:24: warning: use 'Valid attribute instead [enabled, by default]

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_executable_semantics.adb:5:20: medium: overflow check might fail, cannot prove lower bound for X + Y [reason for check: result of addition must fit in a 32-bits machine integer] [possible fix: use pragma Overflow_Mode or switch -gnato13 or unit Ada.Numerics.Big_Numerics.Big_Integers]
show_executable_semantics.adb:9:09: medium: precondition might fail, cannot prove upper bound for Add (Integer'Last, 1)
gnatprove: unproved check messages considered as errors

**Runtime output**

GNATprove issues a message on this code warning about a possible overflow when computing the sum of X and Y in the precondition. Indeed, since expressions in assertions have normal Ada semantics, this addition can overflow, as you can easily see by compiling and running the code that calls Add with arguments Integer’Last and 1.

On the other hand, you sometimes may prefer GNATprove to use the mathematical semantics of addition in contracts while the generated code still properly verifies that no error is ever raised at runtime in the body of the program. You can get this behavior by using the compiler switch -gnato?? (for example -gnato13), which allows you to independently set the overflow mode in code (the first digit) and assertions (the second digit). For both, you can either reduce the number of overflow checks (the value 2), completely eliminate them (the value 3), or preserve the default Ada semantics (the value 1).

**Note:** For more details on overflow modes, see the SPARK User's Guide\(^\text{256}\).

### 31.3.2 Additional Assertions and Contracts

As we've seen, a key feature of SPARK is that it allows us to state properties to check using assertions and contracts. SPARK supports preconditions and postconditions as well as assertions introduced by the Assert pragma.

The SPARK language also includes new contract types used to assist formal verification. The new pragma Assume is treated as an assertion during execution but introduces an assumption when proving programs. Its value is a Boolean expression which GNATprove assumes to be true without any attempt to verify that it's true. You'll find this feature useful, but you must use it with great care. Here's an example of using it.

```
procedure Incr (X : in out Integer) is
begin
  pragma Assume (X < Integer’Last);
  X := X + 1;
end Incr;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Pragma_Assume
MD5: bfbc4b8aca259d7516b6acaaee571f8c2

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
incr.adb:4:11: info: overflow check proved

**Note:** For more details on pragma Assume, see the SPARK User's Guide\(^\text{257}\).

The Contract_Cases aspect is another construct introduced for GNATprove, but which also acts as an assertion during execution. It allows you to specify the behavior of a subprogram

using a disjunction of cases. Each element of a Contract_Cases aspect is a guard, which
is evaluated before the call and may only reference the subprogram's inputs, and a conse-
quence. At each call of the subprogram, one and only one guard is permitted to evaluate
to True. The consequence of that case is a contract that's required to be satisfied when
the subprogram returns.

Listing 10: absolute.adb

```
procedure Absolute (X : in out Integer) with
  Pre => X > Integer'First,
  Contract_Cases => (X < 0 => X = -X'Old,
                     X >= 0 => X = X'Old)
is
begin
  if X < 0 then
    X := -X;
  end if;
end Absolute;
```

Note: For more details on Contract_Cases, see the SPARK User's Guide.\(^{258}\)

### 31.4 Debugging Failed Proof Attempts

GNATprove may report an error while verifying a program for any of the following reasons:

- there might be an error in the program; or
- the property may not be provable as written because more information is required; or
- the prover used by GNATprove may be unable to prove a perfectly valid property.

We spend the remainder of this section discussing the sometimes tricky task of debugging
failed proof attempts.

---

31.4.1 Debugging Errors in Code or Specification

First, let’s discuss the case where there’s indeed an error in the program. There are two possibilities: the code may be incorrect or, equally likely, the specification may be incorrect. As an example, there’s an error in our procedure `Incr_Until` below which makes its `Contract_Cases` unprovable.

**Listing 11: show_failed_proof_attempt.ads**

```ada
package Show_Failed_Proof_Attempt is
  Incremented : Boolean := False;

  procedure Incr_Until (X : in out Natural) with
    Contract_Cases =>
      (Incremented => X > X'Old,
       others => X = X'Old);
end Show_Failed_Proof_Attempt;
```

**Listing 12: show_failed_proof_attempt.adb**

```ada
package body Show_Failed_Proof_Attempt is

  procedure Incr_Until (X : in out Natural) is
    begin
      if X < 1000 then
        X := X + 1;
        Incremented := True;
      else
        Incremented := False;
      end if;
    end Incr_Until;
end Show_Failed_Proof_Attempt;
```

Since this is an assertion that can be executed, it may help you find the problem if you run the program with assertions enabled on representative sets of inputs. This allows you to find bugs in both the code and its contracts. In this case, testing `Incr_Until` with an input greater than 1000 raises an exception at runtime.

**Listing 13: show_failed_proof_attempt.ads**

```ada
package Show_Failed_Proof_Attempt is
  Incremented : Boolean := False;

  procedure Incr_Until (X : in out Natural) with
```

(continues on next page)
Listing 14: show_failed_proof_attempt.adb

```ada
package body Show_Failed_Proof_Attempt is

  procedure Incr_Until (X : in out Natural) is
  begin
    if X < 1000 then
      X := X + 1;
      Incremented := True;
    else
      Incremented := False;
    end if;
  end Incr_Until;

end Show_Failed_Proof_Attempt;
```

Listing 15: main.adb

```ada
with Show_Failed_Proof_Attempt; use Show_Failed_Proof_Attempt;

procedure Main is
  Dummy : Integer;
  begin
    Dummy := 0;
    Incr_Until (Dummy);
    Dummy := 1000;
    Incr_Until (Dummy);
  end Main;
```

The error message shows that the first contract case is failing, which means that Incremented is True. However, if we print the value of Incremented before returning, we see that it's False, as expected for the input we provided. The error here is that guards of contract cases are evaluated before the call, so our specification is wrong! To correct this, we should either write \( X < 1000 \) as the guard of the first case or use a standard postcondition with an if-expression.
31.4.2 Debugging Cases where more Information is Required

Even if both the code and the assertions are correct, GNATprove may still report that it can't prove a verification condition for a property. This can happen for two reasons:

- The property may be unprovable because the code is missing some assertion. One category of these cases is due to the modularity of the analysis which, as we discussed above, means that GNATprove only knows about the properties of your subprograms that you have explicitly written.
- There may be some information missing in the logical model of the program used by GNATprove.

Let's look at the case where the code and the specification are correct but there's some information missing. As an example, GNATprove finds the postcondition of Increase to be unprovable.

Listing 16: show_failed_proof_attempt.ads

```ada
package Show_Failed_Proof_Attempt is

  C : Natural := 100;

  procedure Increase (X : in out Natural) with
  Post => (if X'Old < C then X > X'Old else X = C);

end Show_Failed_Proof_Attempt;
```

Listing 17: show_failed_proof_attempt.adb

```ada
package body Show_Failed_Proof_Attempt is

  procedure Increase (X : in out Natural) is
  begin
    if X < 90 then
      X := X + 10;
    elsif X >= C then
      X := C;
    else
      X := X + 1;
    end if;
  end Increase;

end Show_Failed_Proof_Attempt;
```

This postcondition is a conditional. It says that if the parameter (X) is less than a certain value (C), its value will be increased by the procedure while if it's greater, its value will be set to C (saturated). When C has the value 100, the code of Increases adds 10 to the value of X if it was initially less than 90, increments X by 1 if it was between 90 and 99, and sets
X to 100 if it was greater or equal to 100. This behavior does satisfy the postcondition, so why is the postcondition not provable?

The values in the counterexample returned by GNATprove in its message gives us a clue: \( C = 0 \) and \( X = 10 \) and \( X' = 0 \). Indeed, if \( C \) is not equal to 100, our reasoning above is incorrect: the values of 0 for \( C \) and \( X \) on entry indeed result in \( X \) being 10 on exit, which violates the postcondition!

We probably didn't expect the value of \( C \) to change, or at least not to go below 90. But, in that case, we should have stated so by either declaring \( C \) to be constant or by adding a precondition to the *Increase* subprogram. If we do either of those, GNATprove is able to prove the postcondition.

### 31.4.3 Debugging Prover Limitations

Finally, there are cases where GNATprove provides a perfectly valid verification condition for a property, but it's nevertheless not proved by the automatic prover that runs in the later stages of the tool's execution. This is quite common. Indeed, GNATprove produces its verification conditions in first-order logic, which is not decidable, especially in combination with the rules of arithmetic. Sometimes, the automatic prover just needs more time. Other times, the prover will abandon the search almost immediately or loop forever without reaching a conclusive answer (either a proof or a counterexample).

For example, the postcondition of our GCD function below — which calculates the value of the GCD of two positive numbers using Euclidean's algorithm — can't be verified with GNATprove's default settings.

```
package Show_Failed_Proof_Attempt is

   function GCD (A, B : Positive) return Positive with
     Post =>
       A mod GCD'Result = 0
     and B mod GCD'Result = 0;
end Show_Failed_Proof_Attempt;
```

```
package body Show_Failed_Proof_Attempt is

   function GCD (A, B : Positive) return Positive is
     begin
       if A > B then
         return GCD (A - B, B);
       elsif B > A then
         return GCD (A, B - A);
       else
         return A;
       end if;
     end GCD;
end Show_Failed_Proof_Attempt;
```

Prover output
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_failed_proof_attempt.ads:5:08: medium: postcondition might fail, cannot prove,
A mod GCD'Result = 0
gnatprove: unproved check messages considered as errors

The first thing we try is increasing the amount of time the prover is allowed to spend on
each verification condition using the --timeout option of GNATprove (e.g., by using the
dialog box in GNAT Studio). In this example, increasing it to one minute, which is relatively
high, doesn't help. We can also specify an alternative automatic prover — if we have one
— using the option --prover of GNATprove (or the dialog box). For our postcondition, we
tried Alt-Ergo, cvc5, and Z3 without any luck.

Listing 20: show_failed_proof_attempt.ads

```ada
package Show_Failed_Proof_Attempt is
  function GCD (A, B : Positive) return Positive with
    Post =>
    A mod GCD'Result = 0
    and B mod GCD'Result = 0;
end Show_Failed_Proof_Attempt;
```

Listing 21: show_failed_proof_attempt.adb

```ada
package body Show_Failed_Proof_Attempt is
  function GCD (A, B : Positive) return Positive is
    Result : Positive;
beging
    if A > B then
      Result := GCD (A - B, B);
      pragma Assert ((A - B) mod Result = 0);
      pragma Assert (B mod Result = 0);
      pragma Assert (A mod Result = 0);
      pragma Assert (A mod Result = 0);
    elsif B > A then
      Result := GCD (A, B - A);
      pragma Assert ((B - A) mod Result = 0);
      pragma Assert (B mod Result = 0);
    else
      Result := A;
    end if;
    return Result;
end GCD;
end Show_Failed_Proof_Attempt;
```

31.4. Debugging Failed Proof Attempts
To better understand the reason for the failure, we added intermediate assertions to simplify the proof and pin down the part that's causing the problem. Adding such assertions is often a good idea when trying to understand why a property is not proved. Here, provers can't verify that if both A - B and B can be divided by Result so can A. This may seem surprising, but non-linear arithmetic, involving, for example, multiplication, modulo, or exponentiation, is a difficult topic for provers and is not handled very well in practice by any of the general-purpose ones like Alt-Ergo, cvc5, or Z3.

**Note:** For more details on how to investigate unproved checks, see the SPARK User's Guide.259

### 31.5 Code Examples / Pitfalls

We end with some code examples and pitfalls.

#### 31.5.1 Example #1

The package Lists defines a linked-list data structure. We call Link(I,J) to make a link from index I to index J and call Goes_To(I,J) to determine if we've created a link from index I to index J. The postcondition of Link uses Goes_To to state that there must be a link between its arguments once Link completes.

Listing 22: lists.ads

```ada
package Lists with SPARK_Mode is

  type Index is new Integer;

  function Goes_To (I, J : Index) return Boolean;

  procedure Link (I, J : Index) with Post => Goes_To (I, J);

private

(continues on next page)
```

type Cell (Is_Set : Boolean := True) is record
  case Is_Set is
  when True =>
    Next : Index;
  when False =>
    null;
  end case;
end record;

type Cell_Array is array (Index) of Cell;

Memory : Cell_Array;

package body Lists with SPARK_Mode is

  function Goes_To (I, J : Index) return Boolean is begin
    if Memory (I).Is_Set then
      return Memory (I).Next = J;
    end if;
    return False;
  end Goes_To;

  procedure Link (I, J : Index) is begin
    Memory (I) := (Is_Set => True, Next => J);
  end Link;

end Lists;

This example is correct, but can't be verified by GNATprove. This is because Goes_To itself has no postcondition, so nothing is known about its result.
Learning Ada

31.5.2 Example #2

We now redefine Goes_To as an expression function.

Listing 24: lists.ads

```ada
package Lists with SPARK_Mode is

    type Index is new Integer;

    function Goes_To (I, J : Index) return Boolean;

    procedure Link (I, J : Index) with Post => Goes_To (I, J);

private

    type Cell (Is_Set : Boolean := True) is record
        case Is_Set is
        when True =>
            Next : Index;
        when False =>
            null;
        end case;
    end record;

    type Cell_Array is array (Index) of Cell;

    Memory : Cell_Array;

    function Goes_To (I, J : Index) return Boolean is
        (Memory (I).Is_Set and then Memory (I).Next = J);
end Lists;
```

Listing 25: lists.adb

```ada
package body Lists with SPARK_Mode is

    procedure Link (I, J : Index) is
    begin
        Memory (I) := (Is_Set => True, Next => J);
    end Link;
end Lists;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_02
MD5: c65953bbe8a5f9fb77a4d94e2dd87f9

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
lists.adb:5:18: info: discriminant check proved
lists.ads:7:47: info: postcondition proved
lists.ads:25:44: info: discriminant check proved

GNATprove can fully prove this version: Goes_To is an expression function, so its body is available for proof (specifically, for creating the postcondition needed for the proof).
31.5.3 Example #3

The package Stacks defines an abstract stack type with a Push procedure that adds an element at the top of the stack and a function Peek that returns the content of the element at the top of the stack (without removing it).

Listing 26: stacks.ads

```ada
package Stacks with SPARK_Mode is

  type Stack is private;

  function Peek (S : Stack) return Natural;

  procedure Push (S : in out Stack; E : Natural) with
  Post => Peek (S) = E;

private

  Max : constant := 10;

  type Stack_Array is array (1 .. Max) of Natural;

  type Stack is record
    Top : Positive;
    Content : Stack_Array;
  end record;

  function Peek (S : Stack) return Natural is
    if S.Top in S.Content'Range then S.Content (S.Top) else 0
  end Peek;

end Stacks;
```

Listing 27: stacks.adb

```ada
package body Stacks with SPARK_Mode is

  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      return;
    end if;
    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;

end Stacks;
```

This example isn't correct. The postcondition of Push is only satisfied if the stack isn't full when we call Push.
31.5.4 Example #4

We now change the behavior of Push so it raises an exception when the stack is full instead of returning.

Listing 28: stacks.ads

```ada
package Stacks with SPARK_Mode is

  type Stack is private;
  Is_Full_E : exception;

  function Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;

private

  Max : constant := 10;

  type Stack_Array is array (1 .. Max) of Natural;

  type Stack is record
    Top : Positive;
    Content : Stack_Array;
  end record;

  function Peek (S : Stack) return Natural is
    (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;
```

Listing 29: stacks.adb

```ada
package body Stacks with SPARK_Mode is

  procedure Push (S : in out Stack; E : Natural) is
    begin
      if S.Top >= Max then
        raise Is_Full_E;
      end if;
      S.Top := S.Top + 1;
      S.Content (S.Top) := E;
    end Push;
end Stacks;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_04
MD5: b573ebe93f85ea171166b6953cbb8956

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
gnatprove: medium: exception might be raised
The postcondition of Push is now proved because GNATprove only considers execution paths
leading to normal termination. But it issues a message warning that exception Is_Full_E may be raised at runtime.

### 31.5.5 Example #5

Let's add a precondition to Push stating that the stack shouldn't be full.

Listing 30: stacks.ads

```ada
package Stacks with SPARK_Mode is

  type Stack is private;
  Is_Full_E : exception;

  function Peek (S : Stack) return Natural;
  function Is_Full (S : Stack) return Boolean;
  procedure Push (S : in out Stack; E : Natural) with
    Pre => not Is_Full (S),
    Post => Peek (S) = E;

private

  Max : constant := 10;

  type Stack_Array is array (1 .. Max) of Natural;

  type Stack is record
    Top : Positive;
    Content : Stack_Array;
  end record;

  function Peek (S : Stack) return Natural is
    (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
  function Is_Full (S : Stack) return Boolean is (S.Top >= Max);

end Stacks;
```

Listing 31: stacks.adb

```ada
package body Stacks with SPARK_Mode is

  procedure Push (S : in out Stack; E : Natural) is
    begin
      if S.Top >= Max then
        raise Is_Full_E;
      end if;
      S.Top := S.Top + 1;
      S.Content (S.Top) := E;
    end Push;

end Stacks;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_05
MD5: 63c2dfd68dd5accd91d8d497286e7423e

**Prover output**
This example is correct. With the addition of the precondition, GNATprove can now verify that Is_Full_E can never be raised at runtime.

### 31.5.6 Example #6

The package **Memories** defines a type **Chunk** that models chunks of memory. Each element of the array, represented by its index, corresponds to one data element. The procedure **Read_Record** reads two pieces of data starting at index **From** out of the chunk represented by the value of **Memory**.

#### Listing 32: memories.ads

```ada
package Memories is

  type Chunk is array (Integer range <>) of Integer
  with Predicate => Chunk'Length >= 10;

  function Is_Too_Coarse (V: Integer) return Boolean;

  procedure Treat_Value (V: out Integer);

end Memories;
```

#### Listing 33: read_record.adb

```ada
with Memories; use Memories;

procedure Read_Record (Memory: Chunk; From: Integer)
  with SPARK_Mode => On,
  Pre => From in Memory'First .. Memory'Last - 2
is
  function Read_One (First: Integer; Offset: Integer) return Integer
  with Pre => First + Offset in Memory'Range
  is
    Value: Integer := Memory (First + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;

  Data1, Data2: Integer;

begin
  Data1 := Read_One (From, 1);
  Data2 := Read_One (From, 2);
end Read_Record;
```

*Code block metadata*
Learning Ada

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_06
MD5: aec8014dc291708999092fa123ee7416

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
read_record.adb:8:24: medium: overflow check might fail, cannot prove lower bound
for First + Offset [reason for check: result of addition must fit in a 32-bits
machine integer] [possible fix: use pragma Overflow_Mode or switch -gnato13 or
unit Ada.Numerics.Big_Numerics.Big_Integers]
gnatprove: unproved check messages considered as errors

This example is correct, but it can't be verified by GNATprove, which analyses Read_One on
its own and notices that an overflow may occur in its precondition in certain contexts.

31.5.7 Example #7

Let's rewrite the precondition of Read_One to avoid any possible overflow.

Listing 34: memories.ads

```ada
package Memories is
  type Chunk is array (Integer range <>) of Integer
    with Predicate => Chunk'Length >= 10;
  function Is_Too_Coarse (V : Integer) return Boolean;
  procedure Treat_Value (V : out Integer);
end Memories;
```

Listing 35: read_record.adb

```ada
with Memories; use Memories;
procedure Read_Record (Memory : Chunk; From : Integer)
  with SPARK Mode => On,
    Pre => From in Memory'First .. Memory'Last - 2 is
  function Read_One (First : Integer; Offset : Integer) return Integer
    with Pre => First >= Memory'First
    and then Offset in 0 .. Memory'Last - First
  is
    Value : Integer := Memory (First + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;

  Data1, Data2 : Integer;
begin
  Data1 := Read_One (From, 1);
  Data2 := Read_One (From, 2);
end Read_Record;
```
Learning Ada

31.5.8 Example #8

Let's completely remove the precondition of Read_One.

Listing 36: memories.ads

```ada
package Memories is
    type Chunk is array (Integer range <>) of Integer
        with Predicate => Chunk'Length >= 10;
    function Is_Too_Coarse (V : Integer) return Boolean;
    procedure Treat_Value (V : out Integer);
end Memories;
```

Listing 37: read_record.adb

```ada
with Memories; use Memories;

procedure Read_Record (Memory : Chunk; From : Integer)
    with SPARK_Mode => On,
        Pre => From in Memory'First .. Memory'Last - 2
is
    function Read_One (First : Integer; Offset : Integer) return Integer is
        Value : Integer := Memory (First + Offset);
    begin
        if Is_Too_Coarse (Value) then
            Treat_Value (Value);
        end if;
        return Value;
    end Read_One;

Data1, Data2 : Integer;
begin
    Data1 := Read_One (From, 1);
    Data2 := Read_One (From, 2);
end Read_Record;
```
31.5.9 Example #9

The procedure `Compute` performs various computations on its argument. The computation performed depends on its input range and is reflected in its contract, which we express using a `Contract_Cases` aspect.

Listing 38: compute.adb

```ada
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in -100 .. 100) => X = X'Old * 2,
                     (X in 0 .. 199) => X = X'Old + 1,
                     (X in -199 .. 0) => X = X'Old - 1,
                     X >= 200 => X = 200,
                     others => X = -200)

begin
  if X in -100 .. 100 then
    X := X * 2;
  elsif X in 0 .. 199 then
    X := X + 1;
  elsif X in -199 .. 0 then
    X := X - 1;
  elsif X >= 200 then
    X := 200;
  else
    X := -200;
  end if;
end Compute;
```

This example is correct and fully proved. We could have fixed the contract of `Read_One` to correctly handle both positive and negative values of `Memory`'s `Last`, but we found it simpler to let the function be inlined for proof by removing its precondition.
This example isn't correct. We duplicated the content of Compute's body in its contract. This is incorrect because the semantics of Contract_Cases require disjoint cases, just like a case statement. The counterexample returned by GNATprove shows that $X = 0$ is covered by two different case-guards (the first and the second).

### 31.5.10 Example #10

Let's rewrite the contract of Compute to avoid overlapping cases.

```ada
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in 0 .. 199) => X >= X\'Old,
  (X in 199 .. -1) => X <= X\'Old,
  X >= 200 => X = 200,
  X < -200 => X = -200)
is begin
  if X in -100 .. 100 then
    X := X * 2;
  elsif X in 0 .. 199 then
    X := X + 1;
  elsif X in 199 .. 0 then
    X := X - 1;
  elsif X >= 200 then
    X := 200;
  else
    X := -200;
  end if;
end Compute;
```

This example is still not correct. GNATprove can successfully prove the different cases are disjoint and also successfully verify each case individually. This isn't enough, though: a Contract_Cases must cover all cases. Here, we forgot the value -200, which is what GNATprove reports in its counterexample.
Abstraction is a key concept in programming that can drastically simplify both the implementation and maintenance of code. It's particularly well suited to SPARK and its modular analysis. This section explains what state abstraction is and how you use it in SPARK. We explain how it impacts GNATprove's analysis both in terms of information flow and proof of program properties.

State abstraction allows us to:

- express dependencies that wouldn't otherwise be expressible because some data that's read or written isn't visible at the point where a subprogram is declared — examples are dependencies on data, for which we use the Global contract, and on flow, for which we use the Depends contract.
- reduce the number of variables that need to be considered in flow analysis and proof, a reduction which may be critical in order to scale the analysis to programs with thousands of global variables.

### 32.1 What's an Abstraction?

Abstraction is an important part of programming language design. It provides two views of the same object: an abstract one and a refined one. The abstract one — usually called specification — describes what the object does in a coarse way. A subprogram's specification usually describes how it should be called (e.g., parameter information such as how many and of what types) as well as what it does (e.g., returns a result or modifies one or more of its parameters).

Contract-based programming, as supported in Ada, allows contracts to be added to a subprogram's specification. You use contracts to describe the subprogram's behavior in a more fine-grained manner, but all the details of how the subprogram actually works are left to its refined view, its implementation.

Take a look at the example code shown below.

Listing 1: increase.ads

``` ADA
procedure Increase (X : in out Integer) with
Global => null,
Pre => X <= 100,
Post => X'0ld < X;
```

Listing 2: increase.adb

``` ADA
procedure Increase (X : in out Integer) is
begin
X := X + 1;
end Increase;
```
We’ve written a specification of the subprogram Increase to say that it’s called with a single argument, a variable of type Integer whose initial value is less than 100. Our contract says that the only effect of the subprogram is to increase the value of its argument.

32.2 Why is Abstraction Useful?

A good abstraction of a subprogram’s implementation is one whose specification precisely and completely summarizes what its callers can rely on. In other words, a caller of that subprogram shouldn’t rely on any behavior of its implementation if that behavior isn’t documented in its specification.

For example, callers of the subprogram Increase can assume that it always strictly increases the value of its argument. In the code snippet shown below, this means the loop must terminate.

Listing 3: increase.ads

```ada
procedure Increase (X : in out Integer) with
  Global => null,
  Pre => X <= 100,
  Post => X'Old < X;
```

Listing 4: client.adb

```ada
with Increase;
procedure Client is
  X : Integer := 0;
begin
  while X <= 100 loop -- The loop will terminate
    Increase (X); -- Increase can be called safely
  end loop;
  pragma Assert (X = 101); -- Will this hold?
end Client;
```
Callers can also assume that the implementation of Increase won't cause any runtime errors when called in the loop. On the other hand, nothing in the specification guarantees that the assertion shown above is correct: it may fail if Increase's implementation is changed.

If you follow this basic principle, abstraction can bring you significant benefits. It simplifies both your program's implementation and verification. It also makes maintenance and code reuse much easier since changes to the implementation of an object shouldn't affect the code using this object. Your goal in using it is that it should be enough to understand the specification of an object in order to use that object, since understanding the specification is usually much simpler than understanding the implementation.

GNATprove relies on the abstraction defined by subprogram contracts and therefore doesn't prove the assertion after the loop in Client above.

### 32.3 Abstraction of a Package's State

Subprograms aren't the only objects that benefit from abstraction. The state of a package — the set of persistent variables defined in it — can also be hidden from external users. You achieve this form of abstraction — called state abstraction — by defining variables in the body or private part of a package so they can only be accessed through subprogram calls. For example, our Stack package shown below provides an abstraction for a Stack object which can only be modified using the Pop and Push procedures.

```ada
package Stack is
  procedure Pop (E : out Element);
  procedure Push (E : in Element);
end Stack;

package body Stack is
  Content : Element_Array (1 .. Max);
  Top : Natural;
  ...
end Stack;
```

The fact that we implemented it using an array is irrelevant to the caller. We could change that without impacting our callers' code.

### 32.4 Declaring a State Abstraction

Hidden state influences a program's behavior, so SPARK allows that state to be declared. You can use the Abstract_State aspect, an abstraction that names a state, to do this, but you aren't required to use it even for a package with hidden state. You can use several state abstractions to declare the hidden state of a single package or you can use it for a package with no hidden state at all. However, since SPARK doesn't allow aliasing, different state abstractions must always refer to disjoint sets of variables. A state abstraction isn't a variable: it doesn't have a type and can't be used inside expressions, either those in bodies or contracts.

As an example of the use of this aspect, we can optionally define a state abstraction for the entire hidden state of the Stack package like this:

```ada
package Stack with
  Abstract_State => The_Stack
is ...
```

Alternatively, we can define a state abstraction for each hidden variable:
package Stack with
  Abstract_State => (Top_State, Content_State)
is
...

Remember: a state abstraction isn't a variable (it has no type) and can't be used inside expressions. For example:

pragma Assert (Stack.Top_State = ...);
-- compilation error: Top_State is not a variable

32.5 Refining an Abstract State

Once you've declared an abstract state in a package, you must refine it into its constituents using a Refined_State aspect. You must place the Refined_State aspect on the package body even if the package wouldn't otherwise have required a body. For each state abstraction you've declared for the package, you list the set of variables represented by that state abstraction in its refined state.

If you specify an abstract state for a package, it must be complete, meaning you must have listed every hidden variable as part of some state abstraction. For example, we must add a Refined_State aspect on our Stack package's body linking the state abstraction (The_Stack) to the entire hidden state of the package, which consists of both Content and Top.

Listing 5: stack.ads

package Stack with
  Abstract_State => The_Stack
is
type Element is new Integer;

procedure Pop (E : out Element);
procedure Push (E : Element);
end Stack;

Listing 6: stack.adb

package body Stack with
  Refined_State => (The_Stack => (Content, Top))
is
  Max : constant := 100;
  type Element_Array is array (1 .. Max) of Element;
  Content : Element_Array := (others => 0);
  Top : Natural range 0 .. Max := 0;
  -- Both Content and Top must be listed in the list of
  -- constituents of The_Stack
procedure Pop (E : out Element) is
begin
  E := Content (Top);
  Top := Top - 1;
end Pop;
procedure Push (E : Element) is
(continues on next page)
Learning Ada

(continued from previous page)

```ada
begin
  Top := Top + 1;
  Content (Top) := E;
end Push;

end Stack;
```

### 32.6 Representing Private Variables

You can refine state abstractions in the package body, where all the variables are visible. When only the package's specification is available, you need a way to specify which state abstraction each private variable belongs to. You do this by adding the `Part_Of` aspect to the variable's declaration.

Part_Of annotations are mandatory: if you gave a package an abstract state annotation, you must link all the hidden variables defined in its private part to a state abstraction. For example:

```
package Stack with
  Abstract_State => The_Stack
is
  type Element is new Integer;
  procedure Pop (E : out Element);
  procedure Push (E : Element);

private
  Max : constant := 100;

  type Element_Array is array (1 .. Max) of Element;

  Content : Element_Array with Part_Of => The_Stack;
  Top : Natural range 0 .. Max with Part_Of => The_Stack;

end Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Private_Variables
MD5: 3b5f7edca8a4511071d2397197b01fda

Prover output
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

Since we chose to define Content and Top in Stack's private part instead of its body, we had to add a Part_Of aspect to both of their declarations, associating them with the state abstraction The_Stack, even though it's the only state abstraction. However, we still need to list them in the Refined_State aspect in Stack's body.

```ada
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
```

### 32.7 Additional State

#### 32.7.1 Nested Packages

So far, we've only discussed hidden variables. But variables aren't the only component of a package's state. If a package P contains a nested package, the nested package's state is also part of P's state. If the nested package is hidden, its state is part of P's hidden state and must be listed in P's state refinement.

We see this in the example below, where the package Hidden_Nested's hidden state is part of P's hidden state.

**Listing 8: p.ads**

```ada
package P with
  Abstract_State => State
is
  package Visible_Nested with
    Abstract_State => Visible_State
  is
    procedure Get (E : out Integer);
  end Visible_Nested;
end P;
```

**Listing 9: p.adb**

```ada
package body P with
  Refined_State => (State => Hidden_Nested.Hidden_State)
is
  package Hidden_Nested with
    Abstract_State => Hidden_State,
    Initializes => Hidden_State
  is
    function Get return Integer;
  end Hidden_Nested;

  package body Hidden_Nested with
    Refined_State => (Hidden_State => Cnt)
  is
    Cnt : Integer := 0;
    function Get return Integer is (Cnt);
  end Hidden_Nested;

  package body Visible_Nested with
    Refined_State => (Visible_State => Checked)
  end Visible_Nested;
```

(continues on next page)
is
    Checked : Boolean := False;

procedure Get (E : out Integer) is
begin
    Checked := True;
    E := Hidden_Nested.Get;
end Get;
end Visible_Nested;
end P;

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Nested_Packages
MD5: 8260089cbd651de296dd790506c76fd8

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
p.adb:6:07: info: flow dependencies proved
p.ads:7:22: info: initialization of "E" proved

Any visible state of Hidden_Nested would also have been part of P's hidden state. However, if P contains a visible nested package, that nested package's state isn't part of P's hidden state. Instead, you should declare that package's hidden state in a separate state abstraction on its own declaration, like we did above for Visible_Nested.

32.7.2 Constants that Depend on Variables

Some constants are also possible components of a state abstraction. These are constants whose value depends either on a variable or a subprogram parameter. They're handled as variables during flow analysis because they participate in the flow of information between variables throughout the program. Therefore, GNATprove considers these constants to be part of a package's state just like it does for variables.

If you've specified a state abstraction for a package, you must list such hidden constants declared in that package in the state abstraction refinement. However, constants that don't depend on variables don't participate in the flow of information and must not appear in a state refinement.

Let's look at this example.

Listing 10: stack.ads

package Stack with
    Abstract_State => The_Stack
is
    type Element is new Integer;

    procedure Pop (E : out Element);
    procedure Push (E : Element);
end Stack;

Listing 11: configuration.ads

package Configuration with
    Initializes => External_Variable
is

(continues on next page)
4 External_Variable : Positive with Volatile;
5 end Configuration;

Listing 12: stack.adb

1 with Configuration;
2 pragma Elaborate (Configuration);
3
4 package body Stack with
5   Refined_State => (The_Stack => (Content, Top, Max))
6   -- Max has variable inputs. It must appear as a
7   -- constituent of The_Stack
8 is
9   Max : constant Positive := Configuration.External_Variable;
10
11 type Element_Array is array (1 .. Max) of Element;
12
13 Content : Element_Array := (others => 0);
14 Top : Natural range 0 .. Max := 0;
15
16 procedure Pop (E : out Element) is
17 begin
18   E := Content (Top);
19   Top := Top - 1;
20 end Pop;
21
22 procedure Push (E : Element) is
23 begin
24   Top := Top + 1;
25   Content (Top) := E;
26 end Push;
27
28 end Stack;

Here, Max — the maximum number of elements that can be stored in the stack — is initialized from a variable in an external package. Because of this, we must include Max as part of the state abstraction The_Stack.

Note: For more details on state abstractions, see the SPARK User's Guide260.

32.8 Subprogram Contracts

32.8.1 Global and Depends

Hidden variables can only be accessed through subprogram calls, so you document how state abstractions are modified during the program’s execution via the contracts of those subprograms. You use Global and Depends contracts to specify which of the state abstractions are used by a subprogram and how values flow through the different variables. The Global and Depends contracts that you write when referring to state abstractions are often less precise than contracts referring to visible variables since the possibly different dependencies of the hidden variables contained within a state abstraction are collapsed into a single dependency.

Let’s add Global and Depends contracts to the Pop procedure in our stack.

Listing 13: stack.ads

```ada
package Stack with
  Abstract_State => (Top_State, Content_State)
is
  type Element is new Integer;

  procedure Pop (E : out Element) with
    Global => (Input => Content_State, In_Out => Top_State),
    Depends => (Top_State => Top_State, E => (Content_State, Top_State));
end Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Global_Depends
MD5: a7b383c35508d6a8294bf7cf0fe332ac

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

In this example, the Pop procedure only modifies the value of the hidden variable Top, while Content is unchanged. By using distinct state abstractions for the two variables, we’re able to preserve this semantic in the contract.

Let’s contrast this example with a different representation of Global and Depends contracts, this time using a single abstract state.

Listing 14: stack.ads

```ada
package Stack with
  Abstract_State => The_Stack
is
  type Element is new Integer;

  procedure Pop (E : out Element) with
    Global => (In_Out => The_Stack),
    Depends => ((The_Stack, E) => The_Stack);
end Stack;
```

Code block metadata

32.8. Subprogram Contracts
Here, Top_State and Content_State are merged into a single state abstraction, The_Stack. By doing so, we've hidden the fact that Content isn't modified (though we're still showing that Top may be modified). This loss in precision is reasonable here, since it's the whole point of the abstraction. However, you must be careful not to aggregate unrelated hidden state because this risks their annotations becoming meaningless.

Even though imprecise contracts that consider state abstractions as a whole are perfectly reasonable for users of a package, you should write Global and Depends contracts that are as precise as possible within the package body. To allow this, SPARK introduces the notion of *refined contracts*, which are precise contracts specified on the bodies of subprograms where state refinements are visible. These contracts are the same as normal Global and Depends contracts except they refer directly to the hidden state of the package.

When a subprogram is called inside the package body, you should write refined contracts instead of the general ones so that the verification can be as precise as possible. However, refined Global and Depends are optional: if you don't specify them, GNATprove will compute them to check the package's implementation.

For our Stack example, we could add refined contracts as shown below.

```
package Stack with
    Abstract_State => The_Stack
is
    type Element is new Integer;

    procedure Pop (E : out Element) with
        Global => (In_Out => The_Stack),
        Depends => ((The_Stack, E) => The_Stack);

    procedure Push (E : Element) with
        Global => (In_Out => The_Stack),
        Depends => (The_Stack => (The_Stack, E));
end Stack;
```

```
package body Stack with
    Refined_State => (The_Stack => (Content, Top))
is
    Max : constant := 100;

    type Element_Array is array (1 .. Max) of Element;

    Content : Element_Array := (others => 0);
    Top : Natural range 0 .. Max := 0;

    procedure Pop (E : out Element) with
        Refined_Global => (Input => Content,
                           In_Out => Top),
        Refined_Depends => (Top => Top),
```

(continues on next page)
Learning Ada

(continued from previous page)

is
begin
  E := Content (Top);
  Top := Top - 1;
end Pop;

procedure Push (E : Element) with
  Refined_Global => (In_Out => (Content, Top)),
  Refined_Depends => (Content => Content, Top, E),
  Top => Top) is
begin
  Top := Top + 1;
  Content (Top) := E;
end Push;

end Stack;

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Global_Refined
MD5: b7e700645885155ea7faf2f4170f0462

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

32.8.2 Preconditions and Postconditions

We mostly express functional properties of subprograms using preconditions and postconditions. These are standard Boolean expressions, so they can't directly refer to state abstractions. To work around this restriction, we can define functions to query the value of hidden variables. We then use these functions in place of the state abstraction in the contract of other subprograms.

For example, we can query the state of the stack with functions Is_Empty and Is_Full and call these in the contracts of procedures Pop and Push:

Listing 17: stack.ads

package Stack is
  type Element is new Integer;
  function Is_Empty return Boolean;
  function Is_Full return Boolean;
  procedure Pop (E : out Element) with
    Pre => not Is_Empty,
    Post => not Is_Full;
  procedure Push (E : Element) with
    Pre => not Is_Full,
    Post => not Is_Empty;
end Stack;

32.8. Subprogram Contracts
Learning Ada

Listing 18: stack.adb

```ada
package body Stack is

Max : constant := 100;

type Element_Array is array (1 .. Max) of Element;

Content : Element_Array := (others => 0);
Top : Natural range 0 .. Max := 0;

function Is_Empty return Boolean is (Top = 0);
function Is_Full return Boolean is (Top = Max);

procedure Pop (E : out Element) is
begin
  E := Content (Top);
  Top := Top - 1;
end Pop;

procedure Push (E : Element) is
begin
  Top := Top + 1;
  Content (Top) := E;
end Push;

end Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Pre_Postconditions_1
MD5: fe9d4b65babeabc7cf0feda29b8b3c

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.adb:15:23: info: index check proved
stack.adb:16:18: info: range check proved
stack.adb:21:28: info: range check proved
stack.adb:22:16: info: index check proved
stack.ads:7:19: info: initialization of "E" proved
stack.ads:9:14: info: postcondition proved
stack.ads:13:14: info: postcondition proved

Just like we saw for Global and Depends contracts, you may often find it useful to have a more precise view of functional contracts in the context where the hidden variables are visible. You do this using expression functions in the same way we did for the functions Is_Empty and Is_Full above. As expression function, bodies act as contracts for GNATprove, so they automatically give a more precise version of the contracts when their implementation is visible.

You may often need a more constraining contract to verify the package's implementation but want to be less strict outside the abstraction. You do this using the Refined_Post aspect. This aspect, when placed on a subprogram's body, provides stronger guarantees to internal callers of a subprogram. If you provide one, the refined postcondition must imply the subprogram's postcondition. This is checked by GNATprove, which reports a failing postcondition if the refined postcondition is too weak, even if it's actually implied by the subprogram's body. SPARK doesn't perform a similar verification for normal preconditions.

For example, we can refine the postconditions in the bodies of Pop and Push to be more detailed than what we wrote for them in their specification.
package Stack is
    type Element is new Integer;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;

    procedure Pop (E : out Element) with
        Pre => not Is_Empty,
        Post => not Is_Full;

    procedure Push (E : Element) with
        Pre => not Is_Full,
        Post => not Is_Empty;
end Stack;

package body Stack is
    Max : constant := 100;
    type Element_Array is array (1 .. Max) of Element;
    Content : Element_Array := (others => 0);
    Top : Natural range 0 .. Max := 0;

    function Is_Empty return Boolean is (Top = 0);
    function Is_Full return Boolean is (Top = Max);

    procedure Pop (E : out Element) with
        Refined_Post => not Is_Full and E = Content (Top)'Old
    begin
        E := Content (Top);
        Top := Top - 1;
    end Pop;

    procedure Push (E : Element) with
        Refined_Post => not Is_Empty and E = Content (Top)
    begin
        Top := Top + 1;
        Content (Top) := E;
    end Push;
end Stack;

Code block metadata
Project: Courses.Intro_To_Spark.State_Abstraction.Pre_Postconditions_2
MD5: 4691565d58ba039b3cbd06e65cecfa88
Prover output
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.adb:14:22: info: refined post proved
stack.adb:14:51: info: index check proved
(continues on next page)
32.9 Initialization of Local Variables

As part of flow analysis, GNATprove checks for the proper initialization of variables. Therefore, flow analysis needs to know which variables are initialized during the package's elaboration.

You can use the Initializes aspect to specify the set of visible variables and state abstractions that are initialized during the elaboration of a package. An Initializes aspect can't refer to a variable that isn't defined in the unit since, in SPARK, a package can only initialize variables declared immediately within the package.

Initializes aspects are optional. If you don't supply any, they'll be derived by GNATprove. For our Stack example, we could add an Initializes aspect.

Listing 21: stack.ads

```ada
package Stack with
  Abstract_State => The_Stack,
  Initializes    => The_Stack
is
  type Element is new Integer;
  procedure Pop (E : out Element);
end Stack;
```

Listing 22: stack.adb

```ada
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
is
  Max : constant := 100;
  type Element_Array is array (1 .. Max) of Element;
  Content : Element_Array := (others => 0);
  Top : Natural range 0 .. Max := 0;
  procedure Pop (E : out Element) is
  begin
    E := Content (Top);
  end Pop;
```

Note: For more details on refinement in contracts, see the SPARK User's Guide.\(^{261}\)

---

Flow analysis also checks for dependencies between variables, so it must be aware of how information flows through the code that performs the initialization of states. We discussed one use of the Initializes aspect above. But you also can use it to provide flow information. If the initial value of a variable or state abstraction is dependent on the value of another visible variable or state abstraction from another package, you must list this dependency in the Initializes contract. You specify the list of entities on which a variable's initial value depends using an arrow following that variable's name.

Let's look at this example:

Listing 23: q.ads

```
package Q is
   External_Variable : Integer := 2;
end Q;
```

Listing 24: p.ads

```
with Q;
package P with
   Initializes => (V1, V2 => Q.External_Variable)
is
   V1 : Integer := 0;
   V2 : Integer := Q.External_Variable;
end P;
```

Here we indicated that V2's initial value depends on the value of Q.External_Variable by including that dependency in the Initializes aspect of P. We didn't list any dependency for V1 because its initial value doesn't depend on any external variable. We could also have stated that lack of dependency explicitly by writing V1 => null.

GNATprove computes dependencies of initial values if you don't supply an Initializes aspect. However, if you do provide an Initializes aspect for a package, it must be com-
Learning Ada

plete: you must list every initialized state of the package, along with all its external dependencies.

Note: For more details on Initializes, see the SPARK User's Guide\textsuperscript{262}.

32.10 Code Examples / Pitfalls

This section contains some code examples to illustrate potential pitfalls.

32.10.1 Example #1

Package Communication defines a hidden local package, Ring_Buffer, whose capacity is initialized from an external configuration during elaboration.

Listing 25: configuration.ads

```ada
package Configuration is
  External_Variable : Natural := 1;
end Configuration;
```

Listing 26: communication.ads

```ada
with Configuration;

package Communication with
  Abstract_State => State,
  Initializes     => (State => Configuration.External_Variable)
is
  function Get_Capacity return Natural;

private

  package Ring_Buffer with
    Initializes => (Capacity => Configuration.External_Variable)
  is
    Capacity : constant Natural := Configuration.External_Variable;
  end Ring_Buffer;
end Communication;
```

Listing 27: communication.adb

```ada
package body Communication with
  Refined_State => (State => Ring_Buffer.Capacity)
is
  function Get_Capacity return Natural is
    begin
      return Ring_Buffer.Capacity;
    end Get_Capacity;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_01
MD5: 207e999f85a5b39fa2b9aebb836b479

Prover output

Phase 1 of 2: generation of Global contracts ...
communication.adb:2:41: error: "Capacity" cannot act as constituent of state "State"
communication.adb:2:41: error: missing Part_Of indicator at communication.ads:14,
should specify encapsulator "State"

This example isn't correct. Capacity is declared in the private part of Communication. Therefore, we should have linked it to State by using the Part_Of aspect in its declaration.

32.10.2 Example #2

Let's add Part_Of to the state of hidden local package Ring_Buffer, but this time we hide variable Capacity inside the private part of Ring_Buffer.

Listing 28: configuration.ads

```ada
package Configuration is
  External_Variable : Natural := 1;
end Configuration;
```

Listing 29: communication.ads

```ada
with Configuration;

package Communication with
  Abstract_State => State
is
  package Ring_Buffer with
    Abstract_State => (B_State with Part_Of => State),
    Initializes => (B_State => Configuration.External_Variable)
  is
    function Get_Capacity return Natural;
  private
    Capacity : constant Natural := Configuration.External_Variable
      with Part_Of => B_State;
    end Ring_Buffer;
end Communication;
```

Listing 30: communication.adb

```ada
package body Communication with
  Refined_State => (State => Ring_Buffer.B_State)
```

(continues on next page)
is

package body Ring_Buffer with
  Refined_State => (B_State => Capacity)
  is
  function Get_Capacity return Natural is (Capacity);
end Ring_Buffer;
end Communication;

---

**Code block metadata**

Project: Courses.Intro_To_Spark.State_Abstraction.Example_02
MD5: b8d31fcfbd11bf305646efe07baeb91b

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
communication.ads:10:06: info: flow dependencies proved

This program is correct and GNATprove is able to verify it.

### 32.10.3 Example #3

Package Counting defines two counters: Black_Counter and Red_Counter. It provides separate initialization procedures for each, both called from the main procedure.

Listing 31: counting.ads

```ada
package Counting with
  Abstract_State => State
  is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
end Counting;
```

Listing 32: counting.adb

```ada
package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
  is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
    begin
      Black_Counter := 0;
    end Reset_Black_Count;

  procedure Reset_Red_Count is
    begin
      Red_Counter := 0;
    end Reset_Red_Count;
end Counting;
```
Listing 33: main.adb

```ada
with Counting; use Counting;

procedure Main is
begin
  Reset_Black_Count;
  Reset_Red_Count;
end Main;
```

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
main.adb:5:04: medium: "Counting.State" might not be initialized after elaboration, ..of main program "Main"
counting.ads:2:21: warning: no procedure exists that can initialize abstract state "Counting.State"
gnatprove: unproved check messages considered as errors

This program doesn't read any uninitialized data, but GNATprove fails to verify that. This is because we provided a state abstraction for package Counting, so flow analysis computes the effects of subprograms in terms of this state abstraction and thus considers State to be an in-out global consisting of both Black.Counter and Red.Counter. So it issues the message requiring that State be initialized after elaboration as well as the warning that no procedure in package Counting can initialize its state.

### 32.10.4 Example #4

Let's remove the abstract state on package Counting.

Listing 34: counting.ads

```ada
package Counting is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
end Counting;
```

Listing 35: counting.adb

```ada
package body Counting is
  Black.Counter, Red.Counter : Natural;
  procedure Reset_Black_Count is
    begin
      Black.Counter := 0;
    end Reset_Black_Count;
  procedure Reset_Red_Count is
    begin
      Red.Counter := 0;
    end Reset_Red_Count;
end Counting;
```
Listing 36: main.adb

```ada
with Counting; use Counting;

procedure Main is
begin
  Reset_Black_Count;
  Reset_Red_Count;
end Main;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.State_Abstraction.Example_04
MD5: 3ddd934b6ede6df7b823e46828694d12

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

This example is correct. Because we didn't provide a state abstraction, GNATprove reasons in terms of variables, instead of states, and proves data initialization without any problem.

### 32.10.5 Example #5

Let's restore the abstract state to package Counting, but this time provide a procedure Reset_All that calls the initialization procedures Reset_Black.Counter and Reset_Red.Counter.

Listing 37: counting.ads

```ada
package Counting with
  Abstract_State => State
is
  procedure Reset_Black_Count with Global => (In_Out => State);
  procedure Reset_Red_Count with Global => (In_Out => State);
  procedure Reset_All with Global => (Output => State);
end Counting;
```

Listing 38: counting.adb

```ada
package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;

  procedure Reset_Red_Count is
  begin
    Red_Counter := 0;
  end Reset_Red_Count;

  procedure Reset_All is
  begin
    Reset_Black_Count;
  end Reset_All;
```

(continues on next page)
Learning Ada

(continued from previous page)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Reset_Red_Count;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>end Reset_All;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>end Counting;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Code block metadata**

Project: Courses.Intro_To_Spark.State_Abstraction.Example_05

MD5: d123ccc644fe6999699388708f2ecf89

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

counting.ads:4:37: info: data dependencies proved

counting.ads:5:37: info: data dependencies proved

counting.ads:6:14: info: initialization of "Black_Counter" constituent of "State"proved

counting.ads:6:14: info: initialization of "Red_Counter" constituent of "State"proved

counting.ads:6:37: info: data dependencies proved

This example is correct. Flow analysis computes refined versions of Global contracts for internal calls and uses these to verify that Reset_All indeed properly initializes State. The Refined_Global and Global annotations are not mandatory and can be computed by GNATprove.

### 32.10.6 Example #6

Let's consider yet another version of our abstract stack unit.

**Listing 39: stack.ads**

```ada
package Stack with
Abstract_State => The_Stack
is
pragma Unevaluated_Use_Of_Old (Allow);

type Element is new Integer;

type Element_Array is array (Positive range <>) of Element;

Max : constant Natural := 100;

subtype Length_Type is Natural range 0 .. Max;

procedure Push (E : Element) with
Post =>
    not Is_Empty and
    (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);

function Peek return Element with Pre => not Is_Empty;

function Is_Full return Boolean;

function Is_Empty return Boolean;

end Stack;
```

**Listing 40: stack.adb**

```ada
package body Stack with
Refined_State => (The_Stack => (Top, Content))
is

Top : Length_Type := 0;
```

(continues on next page)

### 32.10. Code Examples / Pitfalls
Learning Ada

(continued from previous page)

5 Content : Element_Array (1 .. Max) := (others => 0);
6
7 procedure Push (E : Element) is
8 begin
9 Top := Top + 1;
10 Content (Top) := E;
11 end Push;
12
13 function Peek return Element is (Content (Top));
14 function Is_Full return Boolean is (Top >= Max);
15 function Is_Empty return Boolean is (Top = 0);
16 end Stack;

Code block metadata

Project: Courses.Intro To Spark.State Abstraction.Example_06
MD5: 9da2b74da203a639dc66b2d33c8d500d

Build output

stack.ads:15:39: error: there is no applicable operator "=" for package or
procedure name
gprbuild: *** compilation phase failed

Prover output

Phase 1 of 2: generation of Global contracts ...
stack.ads:15:39: error: there is no applicable operator "=" for package or
procedure name
gnatprove: error during generation of Global contracts

This example isn't correct. There's a compilation error in Push's postcondition: The_Stack
is a state abstraction, not a variable, and therefore can't be used in an expression.

32.10.7 Example #7

In this version of our abstract stack unit, a copy of the stack is returned by function
Get_Stack, which we call in the postcondition of Push to specify that the stack shouldn't
be modified if it's full. We also assert that after we push an element on the stack, either
the stack is unchanged (if it was already full) or its top element is equal to the element just
pushed.

Listing 41: stack.ads

package Stack with
  Abstract_State => The_Stack
is
  pragma Unevaluated_Use_Of_Old (Allow);
  type Stack_Model is private;

    type Element is new Integer;
    type Element_Array is array (Positive range <>) of Element;
    Max : constant Natural := 100;
    subtype Length_Type is Natural range 0 .. Max;

    function Peek return Element with Pre => not Is_Empty;
    function Is_Full return Boolean;
    function Is_Empty return Boolean;

(continues on next page)
function Get_Stack return Stack_Model;

procedure Push (E : Element) with
  Post => not Is_Empty and
  (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

private

  type Stack_Model is record
    Top    : Length_Type := 0;
    Content : Element_Array (1 .. Max) := (others => 0);
  end record;
end Stack;

Listing 42: stack.adb

package body Stack with
  Refined_State => (The_Stack => (Top, Content))
is
  Top    : Length_Type := 0;
  Content : Element_Array (1 .. Max) := (others => 0);

procedure Push (E : Element) is
begin
  if Top = Max then
    return;
  end if;
  Top := Top + 1;
  Content (Top) := E;
end Push;

function Peek return Element is (Content (Top));
function Is_Full return Boolean is (Top = Max);
function Is_Empty return Boolean is (Top = 0);
function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));
end Stack;

Listing 43: use_stack.adb

with Stack; use Stack;

procedure Use_Stack (E : Element) with
  Pre => not Is_Empty
is
  F : Element := Peek;
begin
  Push (E);
  pragma Assert (Peek = E or Peek = F);
end Use_Stack;

Code block metadata

MD5: 4831aa7f018f2e2d4e6d102095f8f631

Prover output

32.10. Code Examples / Pitfalls 1067
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

use_stack.adb:9:19: medium: assertion might fail [possible fix: precondition of subprogram at line 3 should mention E]
gnatprove: unproved check messages considered as errors

This program is correct, but GNATprove can't prove the assertion in Use_Stack. Indeed, even if Get_Stack is an expression function, its body isn't visible outside of Stack's body, where it's defined.

32.10.8 Example #8

Let's move the definition of Get_Stack and other expression functions inside the private part of the spec of Stack.

Listing 44: stack.ads

```ada
package Stack with
    Abstract_State => The_Stack
is
    pragma Unevaluated_Use_Of_Old (Allow);

    type Stack_Model is private;

    type Element is new Integer;
    type Element_Array is array (Positive range <>) of Element;
    Max : constant Natural := 100;
    subtype Length_Type is Natural range 0 .. Max;

    function Peek return Element with Pre => not Is_Empty;
    function Is_Full return Boolean;
    function Is_Empty return Boolean;
    function Get_Stack return Stack_Model;

    procedure Push (E : Element) with
        Post => not Is_Empty and
            (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

private

    Top : Length_Type := 0 with Part_Of => The_Stack;
    Content : Element_Array (1 .. Max) := (others => 0) with
        Part_Of => The_Stack;

    type Stack_Model is record
        Top : Length_Type := 0;
        Content : Element_Array (1 .. Max) := (others => 0);
    end record;

    function Peek return Element is (Content (Top));
    function Is_Full return Boolean is (Top >= Max);
    function Is_Empty return Boolean is (Top = 0);

    function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));
end Stack;
```

1068 Chapter 32. State Abstraction
Listing 45: stack.adb

```
package body Stack with
  Refined_State => (The_Stack => (Top, Content))

is

  procedure Push (E : Element) is
  begin
    if Top >= Max then
      return;
    end if;
    Top := Top + 1;
    Content (Top) := E;
  end Push;
end Stack;
```

Listing 46: use_stack.adb

```
with Stack; use Stack;

procedure Use_Stack (E : Element) with
  Pre => not Is_Empty
is
  F : Element := Peek;
begin
  Push (E);
  pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_08
MD5: 7e5204d3f69e71c212e7263906a89da4

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
use_stack.adb:6:19: info: precondition proved
use_stack.adb:9:19: info: precondition proved
use_stack.adb:9:19: info: assertion proved
use_stack.adb:9:31: info: precondition proved
stack.adb:10:30: info: range check proved
stack.adb:11:16: info: index check proved
stack.ads:19:14: info: precondition proved
stack.ads:20:60: info: precondition proved
stack.ads:33:55: info: index check proved

This example is correct. GNATprove can verify the assertion in Use_Stack because it has visibility to Get_Stack's body.
32.10.9 Example #9

Package Data defines three variables, Data_1, Data_2 and Data_3, that are initialized at elaboration (in Data's package body) from an external interface that reads the file system.

Listing 47: external_interface.ads

```ada
package External_Interface with
  Abstract_State => File_System,
  Initializes    => File_System
is
  type Data_Type_1 is new Integer;
  type Data_Type_2 is new Integer;
  type Data_Type_3 is new Integer;

  type Data_Record is record
    Field_1 : Data_Type_1;
    Field_2 : Data_Type_2;
    Field_3 : Data_Type_3;
  end record;

  procedure Read_Data (File_Name : String; Data : out Data_Record)
    with Global => File_System;
end External_Interface;
```

Listing 48: data.ads

```ada
with External_Interface; use External_Interface;

package Data with
  Initializes => (Data_1, Data_2, Data_3)
is
  pragma Elaborate_Body;

  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  Data_3 : Data_Type_3;
end Data;
```

Listing 49: data.adb

```ada
with External_Interface;
pragma Elaborate_All (External_Interface);

package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    Data_2 := Data_Read.Field_2;
    Data_3 := Data_Read.Field_3;
  end;
end Data;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_09
MD5: 0ca44501f0c991865ea50d2ef663d992
**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
data.adb:9:07: high: "External Interface.File_System" must be mentioned as an input of the Initializes aspect of "Data" (SPARK RM 7.1.5(11))
gnatprove: unproved check messages considered as errors

This example isn't correct. The dependency between Data_1's, Data_2's, and Data_3's initial values and File_System must be listed in Data's Initializes aspect.

### 32.10.10 Example #10

Let's remove the Initializes contract on package Data.

**Listing 50: external_interface.ads**

```ada
package External_Interface with
  Abstract_State => File_System,
  Initializes     => File_System
is
  type Data_Type_1 is new Integer;
  type Data_Type_2 is new Integer;
  type Data_Type_3 is new Integer;

  type Data_Record is record
    Field_1 : Data_Type_1;
    Field_2 : Data_Type_2;
    Field_3 : Data_Type_3;
  end record;

  procedure Read_Data (File_Name : String; Data : out Data_Record)
    with Global => File_System;
end External_Interface;
```

**Listing 51: data.ads**

```ada
with External_Interface; use External_Interface;

package Data is
  pragma Elaborate Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  Data_3 : Data_Type_3;
end Data;
```

**Listing 52: data.adb**

```ada
with External_Interface;
pragma Elaborate All (External_Interface);

package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    Data_2 := Data_Read.Field_2;
    Data_3 := Data_Read.Field_3;
  end;
end Data;
```

(continues on next page)
Data_2 := Data_Read.Field_2;
Data_3 := Data_Read.Field_3;
end;
end Data;

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_10
MD5: 60cba2c920c7b1031d13c82a982ed0e9

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
data.adb:7:07: info: initialization of "Data_Read" proved
external_interface.ads:3:03: info: flow dependencies proved

This example is correct. Since Data has no Initializes aspect, GNATprove computes the set of variables initialized during its elaboration as well as their dependencies.
This section is dedicated to the functional correctness of programs. It presents advanced proof features that you may need to use for the specification and verification of your program's complex properties.

### 33.1 Beyond Program Integrity

When we speak about the *correctness* of a program or subprogram, we mean the extent to which it complies with its specification. Functional correctness is specifically concerned with properties that involve the relations between the subprogram's inputs and outputs, as opposed to other properties such as running time or memory consumption.

For functional correctness, we usually specify stronger properties than those required to just prove program integrity. When we're involved in a certification processes, we should derive these properties from the requirements of the system, but, especially in non-certification contexts, they can also come from more informal sources, such as the program's documentation, comments in its code, or test oracles.

For example, if one of our goals is to ensure that no runtime error is raised when using the result of the function `Find` below, it may be enough to know that the result is either 0 or in the range of `A`. We can express this as a postcondition of `Find`.

#### Listing 1: show_find.ads

```adl
package Show_Find is
  type Nat_Array is array (Positive range <>) of Natural;
  function Find (A : Nat_Array; E : Natural) return Natural with Post => Find’Result in 0 | A’Range;
end Show_Find;
```

#### Listing 2: show_find.adb

```adl
package body Show_Find is
  function Find (A : Nat_Array; E : Natural) return Natural is
    begin
      for I in A’Range loop
        if A (I) = E then
          return I;
        end if;
      end loop;
      return 0;
    end Find;
end Show_Find;
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
end Show_Find;
```

**Code block metadata**

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Find_1
MD5: d8f4ace6620fd46af170977c29947289

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:7:20: info: range check proved
show_find.ads:6:14: info: postcondition proved

In this case, it's automatically proved by GNATprove.
However, to be sure that Find performs the task we expect, we may want to verify more complex properties of that function. For example, we want to ensure it returns an index of A where E is stored and returns 0 only if E is nowhere in A. Again, we can express this as a postcondition of Find.

```ada
package Show_Find is

  type Nat_Array is array (Positive range <>) of Natural;

  function Find (A : Nat_Array; E : Natural) return Natural with
    Post =>
        (if (for all I in A'Range => A (I) /= E)
         then Find'Result = 0
         else Find'Result in A'Range and then A (Find'Result) = E);

end Show_Find;
```

Listing 3: show_find.ads

```ada
package body Show_Find is

  function Find (A : Nat_Array; E : Natural) return Natural is
    begin
      for I in A'Range loop
        if A (I) = E then
          return I;
        end if;
      end loop;
      return 0;
    end Find;

end Show_Find;
```

Listing 4: show_find.adb

**Code block metadata**

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Find_2
MD5: 8c12b9768228a3ea45ca02199f65057b

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

(continues on next page)
Writing at least part of your program's specification in the form of contracts has many advantages. You can execute those contracts during testing, which improves the maintainability of the code by detecting discrepancies between the program and its specification in earlier stages of development. If the contracts are precise enough, you can use them as oracles to decide whether a given test passed or failed. In that case, they can allow you to verify the outputs of specific subprograms while running a larger block of code. This may, in certain contexts, replace the need for you to perform unit testing, instead allowing you to run integration tests with assertions enabled. Finally, if the code is in SPARK, you can also use GNATprove to formally prove these contracts.

The advantage of a formal proof is that it verifies all possible execution paths, something which isn't always possible by running test cases. For example, during testing, the post-condition of the subprogram Find shown below is checked dynamically for the set of inputs for which Find is called in that test, but just for that set.

```
package Show_Find is
  type Nat_Array is array (Positive range <>) of Natural;
  function Find (A : Nat_Array; E : Natural) return Natural with
  Post =>
      (if (for all I in A'Range => A (I) /= E)
      then Find'Result = 0
      else Find'Result in A'Range and then A (Find'Result) = E);
end Show_Find;
```

```
package body Show_Find is
  function Find (A : Nat_Array; E : Natural) return Natural is
    begin
      for I in A'Range loop
        if A (I) = E then
          return I;
        end if;
      end loop;
      return 0;
    end Find;
end Show_Find;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Show_Find; use Show_Find;
procedure Use_Find with
  SPARK_Mode => Off
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
is
  Seq : constant Nat_Array (1 .. 3) := (1, 5, 3);
  Res : Natural;
begin
  Res := Find (Seq, 3);
  Put_Line ("Found 3 in index #" & Natural'Image (Res) & " of array");
end Use_Find;
```

33.2 Advanced Contracts

Contracts for functional correctness are usually more complex than contracts for program integrity, so they more often require you to use the new forms of expressions introduced by the Ada 2012 standard. In particular, quantified expressions, which allow you to specify properties that must hold for all or for at least one element of a range, come in handy when specifying properties of arrays.

As contracts become more complex, you may find it useful to introduce new abstractions to improve the readability of your contracts. Expression functions are a good means to this end because you can retain their bodies in your package’s specification.

Finally, some properties, especially those better described as invariants over data than as properties of subprograms, may be cumbersome to express as subprogram contracts. Type predicates, which must hold for every object of a given type, are usually a better match for this purpose. Here’s an example.

```ada
package Show_Sort is
  type Nat_Array is array (Positive range <>) of Natural;
  function Is_Sorted (A : Nat_Array) return Boolean is
    (for all I in A'Range =>
      (if I < A'Last then A (I) <= A (I + 1));
    --    Returns True if A is sorted in increasing order.
  end Is_Sorted;
end Show_Sort;
```
We can use the subtype `Sorted_Nat_Array` as the type of a variable that must remain sorted throughout the program’s execution. Specifying that an array is sorted requires a rather complex expression involving quantifiers, so we abstract away this property as an expression function to improve readability. `Is_Sorted`'s body remains in the package's specification and allows users of the package to retain a precise knowledge of its meaning when necessary. (You must use `Nat_Array` as the type of the operand of `Is_Sorted`. If you use `Sorted_Nat_Array`, you'll get infinite recursion at runtime when assertion checks are enabled since that function is called to check all operands of type `Sorted_Nat_Array`.)

### 33.2.1 Ghost Code

As the properties you need to specify grow more complex, you may have entities that are only needed because they are used in specifications (contracts). You may find it important to ensure that these entities can't affect the behavior of the program or that they're completely removed from production code. This concept, having entities that are only used for specifications, is usually called having *ghost* code and is supported in SPARK by the Ghost aspect.

You can use Ghost aspects to annotate any entity including variables, types, subprograms, and packages. If you mark an entity as Ghost, GNATprove ensures it can't affect the program's behavior. When the program is compiled with assertions enabled, ghost code is executed like normal code so it can execute the contracts using it. You can also instruct the compiler to not generate code for ghost entities.

Consider the procedure `Do_Something` below, which calls a complex function on its input, `X`, and wants to check that the initial and modified values of `X` are related in that complex way.

```ada
package Show_Ghost is

  type T is record
    A, B, C, D, E : Boolean;
  end record;

end Show_Ghost;
```

(continues on next page)
function Formula (X : T) return Boolean is
   ((X.A and X.B) or (X.C and (X.D or X.E)));

function Is_Correct (X, Y : T) return Boolean is
   (Formula (X) = Formula (Y));

procedure Do_Something (X : in out T);
end Show_Ghost;

---

code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Ghost_1
MD5: 0a6caaec950b3b043a53c18bab3cb39b

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_ghost.adb:12:22: info: assertion proved

Do_Something stores the initial value of X in a ghost constant, X_Init. We reference it in an assertion to check that the computation performed by the call to Do_Some_Complex_Stuff modified the value of X in the expected manner.

However, X_Init can't be used in normal code, for example to restore the initial value of X.

---

1078 Chapter 33. Proof of Functional Correctness
procedure Do_Something (X : in out T);

end Show_Ghost;

Listing 12: show_ghost.adb

package body Show_Ghost is

procedure Do_Some_Complex_Stuff (X : in out T) is
begin
X := T'(X.B, X.A, X.C, X.E, X.D);
end Do_Some_Complex_Stuff;

procedure Do_Something (X : in out T) is
X_Init : constant T := X with Ghost;
begin
Do_Some_Complex_Stuff (X);
pragma Assert (Is_Correct (X_Init, X));
X := X_Init; -- ERROR
end Do_Something;

end Show_Ghost;

Listing 13: use_ghost.adb

with Show_Ghost; use Show_Ghost;

procedure Use_Ghost is
X : T := (True, True, False, False, True);
begin
Do_Something (X);
end Use_Ghost;

Code block metadata

MD5: 464bb4bc355a648e2b92940ec80b4717

Build output

show_ghost.adb:14:12: error: ghost entity cannot appear in this context
go build: *** compilation phase failed

Prover output

Phase 1 of 2: generation of Global contracts ...
show_ghost.adb:14:12: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts

When compiling this example, the compiler flags the use of X_Init as illegal, but more complex cases of interference between ghost and normal code may sometimes only be detected when you run GNATprove.
33.2.2 Ghost Functions

Functions used only in specifications are a common occurrence when writing contracts for functional correctness. For example, expression functions used to simplify or factor out common patterns in contracts can usually be marked as ghost.

But ghost functions can do more than improve readability. In real-world programs, it's often the case that some information necessary for functional specification isn't accessible in the package's specification because of abstraction.

Making this information available to users of the packages is generally out of the question because that breaks the abstraction. Ghost functions come in handy in that case since they provide a way to give access to that information without making it available to normal client code.

Let's look at the following example.

Listing 14: stacks.ads

```ada
package Stacks is

  pragma Unevaluated_Use_Of_Old (Allow);

  type Stack is private;

  type Element is new Natural;
  type Element_Array is array (Positive range <>) of Element;

  Max : constant Natural := 100;

  function Get_Model (S : Stack) return Element_Array with Ghost;
    -- Returns an array as a model of a stack.

  procedure Push (S : in out Stack; E : Element) with
    Pre => Get_Model (S)'Length < Max,
    Post => Get_Model (S) = Get_Model (S)'Old & E;

private

  subtype Length_Type is Natural range 0 .. Max;

  type Stack is record
    Top : Length_Type := 0;
    Content : Element_Array (1 .. Max) := (others => 0);
  end record;
end Stacks;
```

Here, the type Stack is private. To specify the expected behavior of the Push procedure, we need to go inside this abstraction and access the values of the elements stored in S. For this, we introduce a function Get_Model that returns an array as a representation of the stack. However, we don't want code that uses the Stack package to use Get_Model in normal code since this breaks our stack's abstraction.
Here's an example of trying to break that abstraction in the subprogram Peek below.

```ada
package Stacks is
  pragma Unevaluated_Use_Of_Old (Allow);
  type Stack is private;
  type Element is new Natural;
  type Element_Array is array (Positive range <>) of Element;
  Max : constant Natural := 100;

  function Get_Model (S : Stack) return Element_Array with Ghost;
  -- Returns an array as a model of a stack.

  procedure Push (S : in out Stack; E : Element) with
    Pre => Get_Model (S)'Length < Max,
    Post => Get_Model (S) = Get_Model (S)'Old & E;

  function Peek (S : Stack; I : Positive) return Element is
    (Get_Model (S) (I)); -- ERROR

private

  subtype Length_Type is Natural range 0 .. Max;

  type Stack is record
    Top : Length_Type := 0;
    Content : Element_Array (1 .. Max) := (others => 0);
  end record;
end Stacks;
```

We see that marking the function as Ghost achieves this goal: it ensures that the subprogram Get_Model is never used in production code.

### 33.2.3 Global Ghost Variables

Though it happens less frequently, you may have specifications requiring you to store additional information in global variables that isn't needed in normal code. You should mark these global variables as ghost, allowing the compiler to remove them when assertions aren't enabled. You can use these variables for any purpose within the contracts that make up your specifications. A common scenario is writing specifications for subprograms that modify a complex or private global data structure: you can use these variables to provide a model for that structure that's updated by the ghost code as the program modifies the data structure itself.

You can also use ghost variables to store information about previous runs of subprograms.
Learning Ada

to specify temporal properties. In the following example, we have two procedures, one
that accesses a state A and the other that accesses a state B. We use the ghost variable
Last_Accessed_Is_A to specify that B can’t be accessed twice in a row without accessing
A in between.

Listing 16: call_sequence.ads

```ada
package Call_Sequence is
  type T is new Integer;
  Last_Accessed_Is_A : Boolean := False with Ghost;
  procedure Access_A with
    Post => Last_Accessed_Is_A;
  procedure Access_B with
    Pre => Last_Accessed_Is_A,
    Post => not Last_Accessed_Is_A;
-- B can only be accessed after A
end Call_Sequence;
```

Listing 17: call_sequence.adb

```ada
package body Call_Sequence is
  procedure Access_A is
    begin
      -- ...
      Last_Accessed_Is_A := True;
    end Access_A;
  procedure Access_B is
    begin
      -- ...
      Last_Accessed_Is_A := False;
    end Access_B;
end Call_Sequence;
```

Listing 18: main.adb

```ada
with Call_Sequence; use Call_Sequence;
procedure Main is
  begin
    Access_A;
    Access_B;
    Access_B; -- ERROR
end Main;
```

Code block metadata

MD5: f33fa2ad2bd31eb03d4400c78f22eb71

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

(continues on next page)
main.adb:7:04: medium: precondition might fail
gnatprove: unproved check messages considered as errors

Runtime output

raised ADA.ASSERTIONS ASSERTION_ERROR : failed precondition from call_sequence.
   ads:11

Let's look at another example. The specification of a subprogram's expected behavior is sometimes best expressed as a sequence of actions it must perform. You can use global ghost variables that store intermediate values of normal variables to write this sort of specification more easily.

For example, we specify the subprogram Do_Two_Things below in two steps, using the ghost variable V_Interm to store the intermediate value of V between those steps. We could also express this using an existential quantification on the variable V_Interm, but it would be impractical to iterate over all integers at runtime and this can't always be written in SPARK because quantification is restricted to for ... loop patterns.

Finally, supplying the value of the variable may help the prover verify the contracts.

Listing 19: action_sequence.ads

```ada
package Action_Sequence is
    type T is new Integer;
    V_Interm : T with Ghost;
    function First_Thing.Done (X, Y : T) return Boolean with Ghost;
    function Second_Thing.Done (X, Y : T) return Boolean with Ghost;
    procedure Do_Two_Things (V : in out T) with
        Post => First_Thing.Done (V'Old, V_Interm)
        and then Second_Thing.Done (V_Interm, V);
end Action_Sequence;
```

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

Note: For more details on ghost code, see the SPARK User's Guide\(^{263}\).

---

33.3 Guide Proof

Since properties of interest for functional correctness are more complex than those involved in proofs of program integrity, we expect GNATprove to initially be unable to verify them even though they're valid. You'll find the techniques we discussed in Debugging Failed Proof Attempts (page 1028) to come in handy here. We now go beyond those techniques and focus on more ways of improving results in the cases where the property is valid but GNATprove can't prove it in a reasonable amount of time.

In those cases, you may want to try and guide GNATprove to either complete the proof or strip it down to a small number of easily-reviewable assumptions. For this purpose, you can add assertions to break complex proofs into smaller steps.

```ada
pragma Assert (Assertion_Checked_By_The_Tool);
-- info: assertion proved
pragma Assert (Assumption_Validated_By_Other_Means);
-- medium: assertion might fail
pragma Assume (Assumption_Validated_By_Other_Means);
-- The tool does not attempt to check this expression.
-- It is recorded as an assumption.
```

One such intermediate step you may find useful is to try to prove a theoretically-equivalent version of the desired property, but one where you've simplified things for the prover, such as by splitting up different cases or inlining the definitions of functions.

Some intermediate assertions may not be proved by GNATprove either because it's missing some information or because the amount of information available is confusing. You can verify these remaining assertions by other means such as testing (since they’re executable) or by review. You can then choose to instruct GNATprove to ignore them, either by turning them into assumptions, as in our example, or by using a `pragma Annotate`. In both cases, the compiler generates code to check these assumptions at runtime when you enable assertions.

33.3.1 Local Ghost Variables

You can use ghost code to enhance what you can express inside intermediate assertions in the same way we did above to enhance our contracts in specifications. In particular, you'll commonly have local variables or constants whose only purpose is to be used in assertions. You'll mostly use these ghost variables to store previous values of variables or expressions you want to refer to in assertions. They're especially useful to refer to initial values of parameters and expressions since the 'Old attribute is only allowed in postconditions.

In the example below, we want to help GNATprove verify the postcondition of P. We do this by introducing a local ghost constant, X_Init, to represent this value and writing an assertion in both branches of an `if` statement that repeats the postcondition, but using X_Init.

```ada
package Show_Local_Ghost is
  type T is new Natural;
  function F (X, Y : T) return Boolean is (X > Y) with Ghost;
  function Condition (X : T) return Boolean is (X mod 2 = 0);
end Show_Local_Ghost;
```

Listing 20: show_local_ghost.ads
procedure P (X : in out T) with
  Pre => X < 1_000_000,
  Post => F (X, X'Old);
end Show_Local_Ghost;

Listing 21: show_local_ghost.adb

package body Show_Local_Ghost is
  procedure P (X : in out T) is
    X_Init : constant T := X with Ghost;
    begin
      if Condition (X) then
        X := X + 1;
        pragma Assert (F (X, X_Init));
      else
        X := X * 2;
        pragma Assert (F (X, X_Init));
      end if;
    end P;
  end Show_Local_Ghost;

You can also use local ghost variables for more complex purposes such as building a data structure that serves as witness for a complex property of a subprogram. In our example, we want to prove that the Sort procedure doesn't create new elements, that is, that all the elements present in A after the sort were in A before the sort. This property isn't enough to ensure that a call to Sort produces a value for A that's a permutation of its value before the call (or that the values are indeed sorted). However, it's already complex for a prover to verify because it involves a nesting of quantifiers. To help GNATprove, you may find it useful to store, for each index I, an index J that has the expected property.

procedure Sort (A : in out Nat_Array) with
  Post => (for all I in A'Range =>
    (for some J in A'Range => A (I) = A'Old (J)))
is
  Permutation : Index_Array := (1 => 1, 2 => 2, ...) with Ghost;
  begin
    ...
  end Sort;
33.3.2 Ghost Procedures

Ghost procedures can't affect the value of normal variables, so they're mostly used to perform operations on ghost variables or to group together a set of intermediate assertions.

Abstracting away the treatment of assertions and ghost variables inside a ghost procedure has several advantages. First, you're allowed to use these variables in any way you choose in code inside ghost procedures. This isn't the case outside ghost procedures, where the only ghost statements allowed are assignments to ghost variables and calls to ghost procedures.

As an example, the for loop contained in Increase_A couldn't appear by itself in normal code.

```
package Show_Ghost_Proc is
  type Nat_Array is array (Integer range <>) of Natural;
  A : Nat_Array (1 .. 100) with Ghost;
  procedure Increase_A with
    Ghost,
    Pre => (for all I in A'Range => A (I) < Natural'Last);
end Show_Ghost_Proc;
```

```
package body Show_Ghost_Proc is
  procedure Increase_A is
    begin
      for I in A'Range loop
        A (I) := A (I) + 1;
      end loop;
      end Increase_A;
end Show_Ghost_Proc;
```

Using the abstraction also improves readability by hiding complex code that isn't part of the functional behavior of the subprogram. Finally, it can help GNATprove by abstracting away assertions that would otherwise make its job more complex.

In the example below, calling Prove_P with X as an operand only adds P (X) to the proof context instead of the larger set of assertions required to verify it. In addition, the proof of P need only be done once and may be made easier not having any unnecessary information present in its context while verifying it. Also, if GNATprove can't fully verify Prove_P, you can review the remaining assumptions more easily since they're in a smaller context.
procedure Prove_P (X : T) with Ghost,
    Global => null,
    Post   => P (X);

33.3.3 Handling of Loops

When the program involves a loop, you're almost always required to provide additional
annotations to allow GNATprove to complete a proof because the verification techniques
used by GNATprove don't handle cycles in a subprogram's control flow. Instead, loops are
flattened by dividing them into several acyclic parts.

As an example, let's look at a simple loop with an exit condition.

    Stmt1;
    loop
        Stmt2;
        exit when Cond;
        Stmt3;
    end loop;
    Stmt4;

As shown below, the control flow is divided into three parts.

![Control Flow Diagram]

The first, shown in yellow, starts earlier in the subprogram and enters the loop statement.
The loop itself is divided into two parts. Red represents a complete execution of the loop's
body: an execution where the exit condition isn't satisfied. Blue represents the last execu-
tion of the loop, which includes some of the subprogram following it. For that path, the exit
condition is assumed to hold. The red and blue parts are always executed after the yellow
one.

GNATprove analyzes these parts independently since it doesn't have a way to track how
variables may have been updated by an iteration of the loop. It forgets everything it knows
about those variables from one part when entering another part. However, values of con-
stants and variables that aren't modified in the loop are not an issue.

In other words, handling loops in that way makes GNATprove imprecise when verifying a
subprogram involving a loop: it can't verify a property that relies on values of variables
modified inside the loop. It won't forget any information it had on the value of constants or
unmodified variables, but it nevertheless won't be able to deduce new information about
them from the loop.

For example, consider the function Find which iterates over the array A and searches for
an element where E is stored in A.

Listing 24: show_find.ads

```
package Show_Find is

(continues on next page)
```
type Nat_Array is array (Positive range <>) of Natural;

function Find (A : Nat_Array; E : Natural) return Natural;

end Show_Find;

Listing 25: show_find.adb

package body Show_Find is

function Find (A : Nat_Array; E : Natural) return Natural is
begin
  for I in A'Range loop
    pragma Assert (for all J in A'First .. I - 1 => A (J) /= E);  
    -- assertion is not proved
    if A (I) = E then
      return I;
    end if;
    pragma Assert (A (I) /= E); 
    -- assertion is proved
  end loop;
  return 0;
end Find;

end Show_Find;

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop
MD5: cb9cd0cb102c3baba3b21a788b6e4ae3

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:6:51: info: overflow check proved
show_find.adb:6:58: medium: assertion might fail, cannot prove A (J) /= E
  -- [possible fix: subprogram at show_find.ads:5 should mention A and E in a
  -- precondition]
show_find.adb:6:61: info: index check proved
show_find.adb:9:20: info: range check proved
show_find.adb:11:25: info: assertion proved
gnatprove: unproved check messages considered as errors

At the end of each loop iteration, GNATprove knows that the value stored at index I in A must not be E. (If it were, the loop wouldn't have reached the end of the iteration.) This proves the second assertion. But it's unable to aggregate this information over multiple loop iterations to deduce that it's true for all the indexes smaller than I, so it can't prove the first assertion.
33.3.4 Loop Invariants

To overcome these limitations, you can provide additional information to GNATprove in the form of a loop invariant. In SPARK, a loop invariant is a Boolean expression which holds true at every iteration of the loop. Like other assertions, you can have it checked at runtime by compiling the program with assertions enabled.

The major difference between loop invariants and other assertions is the way it's treated for proofs. GNATprove performs the proof of a loop invariant in two steps: first, it checks that it holds for the first iteration of the loop and then it checks that it holds in an arbitrary iteration assuming it held in the previous iteration. This is called proof by induction\(^\text{264}\).

As an example, let's add a loop invariant to the Find function stating that the first element of A is not E.

Listing 26: show_find.ads

```ada
package Show_Find is
  type Nat_Array is array (Positive range <>) of Natural;
  function Find (A : Nat_Array; E : Natural) return Natural;
end Show_Find;
```

Listing 27: show_find.adb

```ada
package body Show_Find is
  function Find (A : Nat_Array; E : Natural) return Natural is begin
    for I in A’Range loop
      pragma Loop_Invariant (A (A’First) /= E);
      -- loop invariant not proved in first iteration
      -- but preservation of loop invariant is proved
      if A (I) = E then
        return I;
      end if;
    end loop;
    return 0;
  end Find;
end Show_Find;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:6:33: info: loop invariant preservation proved
show_find.adb:6:33: medium: loop invariant might fail in first iteration [possible
  fix: subprogram at show_find.ads:5 should mention A and E in a precondition
show_find.adb:6:33: info: index check proved
show_find.adb:10:20: info: range check proved
gnatprove: unproved check messages considered as errors
```

\(^{264}\) https://en.wikipedia.org/wiki/Mathematical_induction
Learning Ada

To verify this invariant, GNATprove generates two checks. The first checks that the assertion holds in the first iteration of the loop. This isn't verified by GNATprove. And indeed there's no reason to expect the first element of A to always be different from E in this iteration. However, the second check is proved: it's easy to deduce that if the first element of A was not E in a given iteration it's still not E in the next. However, if we move the invariant to the end of the loop, then it is successfully verified by GNATprove.

Not only do loop invariants allow you to verify complex properties of loops, but GNATprove also uses them to verify other properties, such as the absence of runtime errors over both the loop's body and the statements following the loop. More precisely, when verifying a runtime check or other assertion there, GNATprove assumes that the last occurrence of the loop invariant preceding the check or assertion is true.

Let's look at a version of Find where we use a loop invariant instead of an assertion to state that none of the array elements seen so far are equal to E.

Listing 28: show_find.ads

```ada
package Show_Find is
  type Nat_Array is array (Positive range <>) of Natural;
  function Find (A : Nat_Array; E : Natural) return Natural;
end Show_Find;
```

Listing 29: show_find.adb

```ada
package body Show_Find is
  function Find (A : Nat_Array; E : Natural) return Natural is
  begin
    for I in A'Range loop
      pragma Loop_Invariant
      (for all J in A'First .. I - 1 => A (J) /= E);
      if A (I) = E then
        return I;
      end if;
    end loop;
    pragma Assert (for all I in A'Range => A (I) /= E);
    return 0;
  end Find;
end Show_Find;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop_Invariant_2
MD5: 21588161eaddb82f54c3cb3dcc14a6ac

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:7:13: info: loop invariant initialization proved
show_find.adb:7:13: info: loop invariant preservation proved
show_find.adb:7:39: info: overflow check proved
show_find.adb:7:49: info: index check proved
show_find.adb:9:20: info: range check proved
show_find.adb:12:22: info: assertion proved
show_find.adb:12:49: info: index check proved

This version is fully verified by GNATprove! This time, it proves that the loop invariant holds
in every iteration of the loop (separately proving this property for the first iteration and then for the following iterations). It also proves that none of the elements of A are equal to E after the loop exits by assuming that the loop invariant holds in the last iteration of the loop.

**Note:** For more details on loop invariants, see the SPARK User's Guide\textsuperscript{265}.

Finding a good loop invariant can turn out to be quite a challenge. To make this task easier, let's review the four good properties of a good loop invariant:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>It should be provable in the first iteration of the loop.</td>
</tr>
<tr>
<td>INSIDE</td>
<td>It should allow proving the absence of run-time errors and local assertions inside the loop.</td>
</tr>
<tr>
<td>AFTER</td>
<td>It should allow proving absence of run-time errors, local assertions, and the subprogram postcondition after the loop.</td>
</tr>
<tr>
<td>PRESERVE</td>
<td>It should be provable after the first iteration of the loop.</td>
</tr>
</tbody>
</table>

Let's look at each of these in turn. First, the loop invariant should be provable in the first iteration of the loop (INIT). If your invariant fails to achieve this property, you can debug the loop invariant's initialization like any failing proof attempt using strategies for *Debugging Failed Proof Attempts* (page 1028).

Second, the loop invariant should be precise enough to allow GNATprove to prove absence of runtime errors in both statements from the loop's body (INSIDE) and those following the loop (AFTER). To do this, you should remember that all information concerning a variable modified in the loop that's not included in the invariant is forgotten by GNATprove. In particular, you should take care to include in your invariant what's usually called the loop's *frame condition*, which lists properties of variables that are true throughout the execution of the loop even though those variables are modified by the loop.

Finally, the loop invariant should be precise enough to prove that it's preserved through successive iterations of the loop (PRESERVE). This is generally the trickiest part. To understand why GNATprove hasn't been able to verify the preservation of a loop invariant you provided, you may find it useful to repeat it as local assertions throughout the loop's body to determine at which point it can no longer be proved.

As an example, let's look at a loop that iterates through an array A and applies a function \( F \) to each of its elements.

```
package Show_Map is
  type Nat_Array is array (Positive range <>) of Natural;
  function F (V : Natural) return Natural is
    (if V /= Natural'Last then V + 1 else V);
  procedure Map (A : in out Nat_Array);
end Show_Map;
```

package body Show_Map is

procedure Map (A : in out Nat_Array) is
  A_I : constant Nat_Array := A with Ghost;
begin
  for K in A'Range loop
    A (K) := F (A (K));
    pragma Loop_Invariant
    (for all J in A'First .. K => A (J) = F (A'Loop_Entry (J)));
  end loop;
  pragma Assert (for all K in A'Range => A (K) = F (A_I (K)));
end Map;
end Show_Map;

Code block metadata
MD5: 1a4583c9b2b772f79b029c09ca96a

Prover output
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_map.adb:9:13: info: loop invariant initialization proved
show_map.adb:9:45: info: index check proved
show_map.adb:9:67: info: index check proved
show_map.adb:11:22: info: assertion proved
show_map.adb:11:49: info: index check proved
show_map.adb:11:62: info: index check proved
show_map.ads:6:35: info: overflow check proved

After the loop, each element of A should be the result of applying F to its previous value. We want to prove this. To specify this property, we copy the value of A before the loop into a ghost variable, A_I. Our loop invariant states that the element at each index less than K has been modified in the expected way. We use the Loop_Entry attribute to refer to the value of A on entry of the loop instead of using A_I.

Does our loop invariant have the four properties of a good loop-invariant? When launching GNATprove, we see that INIT is fulfilled: the invariant's initialization is proved. So are INSIDE and AFTER: no potential runtime errors are reported and the assertion following the loop is successfully verified.

The situation is slightly more complex for the PRESERVE property. GNATprove manages to prove that the invariant holds after the first iteration thanks to the automatic generation of frame conditions. It was able to do this because it completes the provided loop invariant with the following frame condition stating what part of the array hasn't been modified so far:

\begin{verbatim}
pragma Loop_Invariant
  (for all J in K .. A'Last => A (J) = (if J > K then A'Loop_Entry (J)));
\end{verbatim}

GNATprove then uses both our and the internally-generated loop invariants to prove PRESERVE. However, in more complex cases, the heuristics used by GNATprove to generate the frame condition may not be sufficient and you'll have to provide one as a loop invariant. For example, consider a version of Map where the result of applying F to an element at index K is stored at index K - 1:
package Show_Map is

  type Nat_Array is array (Positive range <>) of Natural;

  function F (V : Natural) return Natural is
    (if V /= Natural'Last then V + 1 else V);

procedure Map (A : in out Nat_Array);

end Show_Map;

package body Show_Map is

  procedure Map (A : in out Nat_Array) is
    A_I : constant Nat_Array := A with Ghost;
    begin
      for K in A'Range loop
        if K /= A'First then
          A (K - 1) := F (A (K));
        end if;
        pragma Loop Invariant
        (for all J in A'First .. K =>
          (if J /= A'First then A (J - 1) = F (A'Loop_Entry (J))));
        pragma Loop Invariant
        (for all J in K .. A'Last => A (J) = A'Loop_Entry (J));
      end loop;
      pragma Assert (for all K in A'Range =>
        (if K /= A'First then A (K - 1) = F (A_I (K))));
    end Map;

end Show_Map;

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop_Invariant_4
MD5: 6c51768547d3baa2c19d0e33959388fe

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_map.adb:8:18: info: overflow check proved
show_map.adb:8:18: info: index check proved
show_map.adb:11:13: info: loop invariant initialization proved
show_map.adb:12:36: medium: loop invariant might not be preserved by an arbitrary
  iteration, cannot prove A (J - 1) = F (A'Loop_Entry (J))
show_map.adb:12:41: info: overflow check proved
show_map.adb:12:41: info: index check proved
show_map.adb:12:65: info: index check proved
show_map.adb:16:22: info: assertion proved
show_map.adb:17:50: info: overflow check proved
show_map.adb:17:50: info: index check proved
show_map.adb:17:65: info: index check proved
show_map.ads:6:35: info: overflow check proved
gnatprove: unproved check messages considered as errors

You need to uncomment the second loop invariant containing the frame condition in order
prove the assertion after the loop.
33.4 Code Examples / Pitfalls

This section contains some code examples and pitfalls.

33.4.1 Example #1

We implement a ring buffer inside an array Content, where the contents of a ring buffer of length Length are obtained by starting at index First and possibly wrapping around the end of the buffer. We use a ghost function Get_Model to return the contents of the ring buffer for use in contracts.

Listing 34: ring_buffer.ads

```ada
package Ring_Buffer is

  Max_Size : constant := 100;

  type Nat_Array is array (Positive range <>) of Natural;

  function Get_Model return Nat_Array with Ghost;

  procedure Push_Last (E : Natural) with
  Pre => Get_Model'Length < Max_Size,
  Post => Get_Model'Length = Get_Model'Old'Length + 1;

end Ring_Buffer;
```

Listing 35: ring_buffer.adb

```ada
package body Ring_Buffer is

  subtype Length_Range is Natural range 0 .. Max_Size;

  subtype Index_Range is Natural range 1 .. Max_Size;

  Content : Nat_Array (1 .. Max_Size) := (others => 0);
  First : Index_Range := 1;
  Length : Length_Range := 0;

  function Get_Model return Nat_Array with
  Refined_Post => Get_Model'Result'Length = Length is
  Size : constant Length_Range := Length;
  Result : Nat_Array (1 .. Size) := (others => 0);
  begin
    if First + Length - 1 <= Max_Size then
      Result := Content (First .. First + Length - 1);
    else
      declare
        Len : constant Length_Range := Max_Size - First + 1;
      begin
        Result (1 .. Len) := Content (First .. Max_Size);
      end begin;
  end if;
```

(continues on next page)

Note: For more details on how to write a loop invariant, see the SPARK User’s Guide\(^\text{266}\).

Result (Len + 1 .. Length) := Content (1 .. Length - Len);

end if;
return Result;
end Get_Model;

procedure Push_Last (E : Natural) is
begin
if First + Length <= Max_Size then
Content (First + Length) := E;
else
Content (Length - Max_Size + First) := E;
end if;
Length := Length + 1;
end Push_Last;
end Ring_Buffer;

This is correct: Get_Model is used only in contracts. Calls to Get_Model make copies of the buffer's contents, which isn't efficient, but is fine because Get_Model is only used for verification, not in production code. We enforce this by making it a ghost function. We'll produce the final production code with appropriate compiler switches (i.e., not using -gnata) that ensure assertions are ignored.
33.4.2 Example #2

Instead of using a ghost function, Get_Model, to retrieve the contents of the ring buffer, we’re now using a global ghost variable, Model.

Listing 36: ring_buffer.ads

```ada
package Ring_Buffer is

   Max_Size : constant := 100;
   subtype Length_Range is Natural range 0 .. Max_Size;
   subtype Index_Range is Natural range 1 .. Max_Size;

   type Nat_Array is array (Positive range <>) of Natural;

   type Model_Type (Length : Length_Range := 0) is record
      Content : Nat_Array (1 .. Length);
      end record
      with Ghost;

   Model : Model_Type with Ghost;

   function Valid_Model return Boolean is
      (Model.Content'Length = Length);

   procedure Push_Last (E : Natural) with
      Pre => Valid_Model
      and then Model.Length < Max_Size,
      Post => Model.Length = Model.Length'Old + 1;
end Ring_Buffer;
```

Listing 37: ring_buffer.adb

```ada
package body Ring_Buffer is

   Content : Nat_Array (1 .. Max_Size) := (others => 0);
   First : Index_Range := 1;
   Length : Length_Range := 0;

   function Valid_Model return Boolean is
      (Model.Content'Length = Length);

   procedure Push_Last (E : Natural) is
      begin
         if First + Length <= Max_Size then
            Content (First + Length) := E;
         else
            Content (Length - Max_Size + First) := E;
         end if;
         Length := Length + 1;
      end Push_Last;

end Ring_Buffer;
```

Code block metadata

- Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_02
  - MD5: 144f58bd95cd460e4ed388d4f3351fe3
This example isn't correct. Model, which is a ghost variable, must not influence the return value of the normal function Valid_Model. Since Valid_Model is only used in specifications, we should have marked it as Ghost. Another problem is that Model needs to be updated inside Push_Last to reflect the changes to the ring buffer.

### 33.4.3 Example #3

Let's mark Valid_Model as Ghost and update Model inside Push_Last.

Listing 38: ring_buffer.ads

```ada
package Ring_Buffer is

   Max_Size : constant := 100;
   subtype Length_Range is Natural range 0 .. Max_Size;
   subtype Index_Range is Natural range 1 .. Max_Size;

   type Nat_Array is array (Positive range <>) of Natural;
   
   type Model_Type (Length : Length_Range := 0) is record
    Content : Nat_Array (1 .. Length);
   end record
   with Ghost;

   Model : Model_Type with Ghost;

   function Valid_Model return Boolean with Ghost;

   procedure Push_Last (E : Natural) with
    Pre => Valid_Model
    and then Model.Length < Max_Size,
    Post => Model.Length = Model.Length'Old + 1;

end Ring_Buffer;
```

Listing 39: ring_buffer.adb

```ada
package body Ring_Buffer is

   Content : Nat_Array (1 .. Max_Size) := (others => 0);
   First : Index_Range := 1;
   Length : Length_Range := 0;

   function Valid_Model return Boolean is
    (Model.Content'Length = Length);

   procedure Push_Last (E : Natural) is
    begin
     if First + Length <= Max_Size then
      Content (First + Length) := E;
```

(continues on next page)
else
    Content (Length - Max_Size + First) := E;
end if;
Length := Length + 1;
Model := (Length => Model.Length + 1,
         Content => Model.Content & E);
end Push_Last;
end Ring_Buffer;

33.4.4 Example #4

We're now modifying Push_Last to share the computation of the new length between the
operational and ghost code.

Listing 40: ring_buffer.ads

package Ring_Buffer is
  Max_Size : constant := 100;
  subtype Length_Range is Natural range 0 .. Max_Size;
  subtype Index_Range is Natural range 1 .. Max_Size;

  type Nat_Array is array (Positive range <>) of Natural;
  type Model_Type (Length : Length_Range := 0) is record
    Content : Nat_Array (1 .. Length);
  end record
  with Ghost;

  Model : Model_Type with Ghost;

  function Valid_Model return Boolean with Ghost;

  procedure Push_Last (E : Natural) with

Learning Ada

(continued from previous page)

Pre => Valid_Model

and then Model.Length < Max_Size,

Post => Model.Length = Model.Length'Old + 1;

end Ring_Buffer;

Listing 41: ring_buffer.adb

package body Ring_Buffer is

Content : Nat_Array (1 .. Max_Size) := (others => 0);
First : Index_Range := 1;
Length : Length_Range := 0;

function Valid_Model return Boolean is
  (Model.Content'Length = Length);

procedure Push_Last (E : Natural) is
  New_Length : constant Length_Range := Model.Length + 1;
begin
  if First + Length <= Max_Size then
    Content (First + Length) := E;
  else
    Content (Length - Max_Size + First) := E;
  end if;
  Length := New_Length;
  Model := (Length => New_Length,
            Content => Model.Content & E);
end Push_Last;

end Ring_Buffer;

Code block metadata

MD5: e27f0b4729be72d83f2cb981b1d00412

Build output

ring_buffer.adb:11:45: error: ghost entity cannot appear in this context
gprbuild: *** compilation phase failed

Prover output

Phase 1 of 2: generation of Global contracts ...
ring_buffer.adb:11:45: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts

This example isn’t correct. We didn’t mark local constant New_Length as Ghost, so it can’t
be computed from the value of ghost variable Model. If we made New_Length a ghost
constant, the compiler would report the problem on the assignment from New_Length to
Length. The correct solution here is to compute New_Length from the value of the non-
ghost variable Length.

33.4. Code Examples / Pitfalls 1099
33.4.5 Example #5

Let’s move the code updating Model inside a local ghost procedure, Update_Model, but still using a local variable, New_Length, to compute the length.

Listing 42: ring_buffer.ads

```ada
package Ring_Buffer is

  Max_Size : constant := 100;
  subtype Length_Range is Natural range 0 .. Max_Size;
  subtype Index_Range is Natural range 1 .. Max_Size;

  type Nat_Array is array (Positive range <>) of Natural;

  type Model_Type (Length : Length_Range := 0) is record
    Content : Nat_Array (1 .. Length);
  end record
    with Ghost;

  Model : Model_Type with Ghost;

  function Valid_Model return Boolean with Ghost;

procedure Push_Last (E : Natural) with
  Pre => Valid_Model
  and then Model.Length < Max_Size,
  Post => Model.Length = Model.Length'Old + 1;
end Ring_Buffer;
```

Listing 43: ring_buffer.adb

```ada
package body Ring_Buffer is

  Content : Nat_Array (1 .. Max_Size) := (others => 0);
  First : Index_Range := 1;
  Length : Length_Range := 0;

  function Valid_Model return Boolean is
    (Model.Content'Length = Length);

procedure Push_Last (E : Natural) is

  procedure Update_Model with Ghost is
    New_Length : constant Length_Range := Model.Length + 1;
    begin
      Model := (Length => New_Length,
        Content => Model.Content & E);
    end Update_Model;

    begin
      if First + Length <= Max_Size then
        Content (First + Length) := E;
      else
        Content (Length - Max_Size + First) := E;
      end if;
      Length := Length + 1;
    end Push_Last;
end Ring_Buffer;
```
33.4.6 Example #6

The function Max_Array takes two arrays of the same length (but not necessarily with the same bounds) as arguments and returns an array with each entry being the maximum values of both arguments at that index.

Listing 44: array_util.ads

```ada
package Array_Util is

  type Nat_Array is array (Positive range <>) of Natural;

  function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

end Array_Util;
```

Listing 45: array_util.adb

```ada
package body Array_Util is

  function Max_Array (A, B : Nat_Array) return Nat_Array is
    R : Nat_Array (A'Range);
    J : Integer := B'First;
  begin
    for I in A'Range loop
      if A (I) > B (J) then
        R (I) := A (I);
      else
        R (I) := B (J);
      end if;
    end loop;
  end function Max_Array;
```

(continues on next page)
Learning Ada

13
J := J + 1;
14
end loop;
15
return R;
16
end Max_Array;
17
end Array_Util;

33.4.7 Example #7

Let's add a loop invariant that states that J stays in the index range of B and let's protect the increment to J by checking that it's not already the maximal integer value.

Listing 46: array_util.ads

package Array_Util is

type Nat_Array is array (Positive range <>) of Natural;

function Max_Array (A, B : Nat_Array) return Nat_Array with
Pre => A'Length = B'Length;

end Array_Util;

Listing 47: array_util.adb

package body Array_Util is

function Max_Array (A, B : Nat_Array) return Nat_Array is
R : Nat_Array (A'Range);
J : Integer := B'First;
begin
for I in A'Range loop
pragma Loop Invariant (J in B'Range);
if A (I) > B (J) then
R (I) := A (I);

This program is correct, but GNATprove can't prove that J is always in the index range of B (the unproved index check) or even that it's always within the bounds of its type (the unproved overflow check). Indeed, when checking the body of the loop, GNATprove forgets everything about the current value of J because it's been modified by previous loop iterations. To get more precise results, we need to provide a loop invariant.
else
    R (I) := B (J);
end if;
if J < Integer'Last then
    J := J + 1;
end if;
end loop;
return R;
end Max_Array;
end Array_Util;

**33.4.8 Example #8**

We now consider a version of Max_Array which takes arguments that have the same bounds. We want to prove that Max_Array returns an array of the maximum values of both its arguments at each index.

**Listing 48: array_util.ads**

```ada
package Array_Util is
  type Nat_Array is array (Positive range <>) of Natural;
  function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
              Max_Array'Result (K) = Natural'Max (A (K), B (K)));
end Array_Util;
```

**Listing 49: array_util.adb**

```ada
package body Array_Util is
  function Max_Array (A, B : Nat_Array) return Nat_Array is
  begin : Nat_Array (A'Range) := (others => 0);
end
```

The loop invariant now allows verifying that no runtime error can occur in the loop's body (property INSIDE seen in section *Loop Invariants* (page 1089)). Unfortunately, GNATprove fails to verify that the invariant stays valid after the first iteration of the loop (property PRESERVE). Indeed, knowing that J is in B'Range in a given iteration isn't enough to prove it'll remain so in the next iteration. We need a more precise invariant, linking J to the value of the loop index I, like J = I - A'First + B'First.
Listing 50: main.adb

```ada
with Array_Util; use Array_Util;

procedure Main is
   A : Nat_Array := (1, 1, 2);
   B : Nat_Array := (2, 1, 0);
   R : Nat_Array (1 .. 3);
begin
   R := Max_Array (A, B);
end Main;
```

Code block metadata

MD5: d0a04c214a632a66a4fe4ec6cb7f8842

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
main.adb:8:09: medium: length check might fail [reason for check: array must be of the appropriate length]
array_util.adb:8:35: medium: loop invariant might not be preserved by an arbitrary iteration, cannot prove R (K) = Natural’max
array_util.adb:8:35: medium: loop invariant might fail in first iteration, cannot prove R (K) = Natural’max
gnatprove: unproved check messages considered as errors

Runtime output

raised ADA.ASSERTIONS.ASSERTION_ERROR : Loop_Invariant failed at array_util.adb:7

Here, GNATprove doesn't manage to prove the loop invariant even for the first loop iteration (property INIT seen in section Loop Invariants (page 1089)). In fact, the loop invariant is incorrect, as you can see by executing the function Max_Array with assertions enabled: at each loop iteration, R contains the maximum of A and B only until I - 1 because the I'th index wasn't yet handled.
33.4.9 Example #9

We now consider a procedural version of `Max_Array` which updates its first argument instead of returning a new array. We want to prove that `Max_Array` sets the maximum values of both its arguments into each index in its first argument.

Listing 51: array_util.ads

```ada
package Array_Util is
  type Nat_Array is array (Positive range <>) of Natural;
  procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
    Pre => A'First = B'First and A'Last = B'Last,
    Post => (for all K in A'Range =>
      A (K) = Natural'Max (A'Old (K), B (K)));
end Array_Util;
```

Listing 52: array_util.adb

```ada
package body Array_Util is
  procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
  begin
    for I in A'Range loop
      pragma Loop Invariant
      (for all K in A'First .. I - 1 =>
        A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
      pragma Loop Invariant
      (for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
      if A (I) <= B (I) then
        A (I) := B (I);
      end if;
    end loop;
  end Max_Array;
end Array_Util;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_09
MD5: 2de4bdd9c59d7d1eccc6259067ffdfc3

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
array_util.adb:7:13: info: loop invariant preservation proved
array_util.adb:7:13: info: loop invariant initialization proved
array_util.adb:7:39: info: overflow check proved
array_util.adb:8:18: info: index check proved
array_util.adb:8:50: info: index check proved
array_util.adb:8:57: info: index check proved
array_util.adb:10:13: info: loop invariant initialization proved
array_util.adb:10:13: info: loop invariant preservation proved
array_util.adb:10:44: info: index check proved
array_util.adb:10:63: info: index check proved
array_util.adb:11:25: info: index check proved
array_util.adb:12:25: info: index check proved
array_util.adb:7:14: info: postcondition proved
array_util.ads:8:20: info: index check proved

(continues on next page)
Everything is proved. The first loop invariant states that the values of \( A \) before the loop index contain the maximum values of the arguments of \( \text{Max}_{\text{Array}} \) (referring to the input value of \( A \) with \( A'\text{Loop\_Entry} \)). The second loop invariant states that the values of \( A \) beyond and including the loop index are the same as they were on entry. This is the frame condition of the loop.

### 33.4.10 Example #10

Let's remove the frame condition from the previous example.

**Listing 53: array_util.ads**

```ada
package Array_Util is

  type Nat_Array is array (Positive range <>) of Natural;

  procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
  Pre => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
            A (K) = Natural'Max (A'Old (K), B (K)));
end Array_Util;
```

**Listing 54: array_util.adb**

```ada
package body Array_Util is

  procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
  begin
    for I in A'Range loop
      pragma Loop Invariant
      (for all K in A'First .. I - 1 =>
        A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
      if A (I) <= B (I) then
        A (I) := B (I);
      end if;
    end loop;
  end Max_Array;
end Array_Util;
```

**Code block metadata**

MD5: 8bdca84023cb3f26f58f6345740e7172

**Prover output**

Phase 1 of 2: generation of Global contracts ...  
Phase 2 of 2: flow analysis and proof ...  
array_util.adb:7:13: info: loop invariant initialization proved  
array_util.adb:7:13: info: loop invariant preservation proved  
array_util.adb:7:39: info: overflow check proved  
array_util.adb:8:18: info: index check proved  
array_util.adb:8:50: info: index check proved
(continues on next page)
Everything is still proved. GNATprove internally generates the frame condition for the loop, so it's sufficient here to state that A before the loop index contains the maximum values of the arguments of Max_Array.
Part IV

Introduction to Embedded Systems Programming
Copyright © 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details on this page.

This course will teach you the basics of the Embedded Systems Programming using Ada.

This document was written by Patrick Rogers, with review by Stephen Baird, Tucker Taft, Filip Gajowniczek, and Gustavo A. Hoffmann.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

---

[267](http://creativecommons.org/licenses/by-sa/4.0)
INTRODUCTION

This is a course about embedded systems programming. Embedded systems are everywhere today, including — just to name a few — the thermostats that control a building's temperature, the power-steering controller in modern automobiles, and the control systems in charge of jet engines.

Clearly, much can depend on these systems operating correctly. It might be only a matter of comfort if the thermostat fails. But imagine what might happen if one of the critical control systems in your car failed when you're out on the freeway. When a jet engine controller is designed to have absolute control, it is known as a Full Authority Digital Engine Controller, or FADEC for short. If a FADEC fails, the result can make international news.

Using Ada can help you get it right, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal is to get you started in using it well.

The course is based on the assumption that you know some of the Ada language already, preferably even some of the more advanced concepts. You don't need to know how to use Ada constructs for embedded systems, of course, but you do need to know at least the language basics. If you need that introduction, see the course Introduction to Ada (page 5).

We also assume that you already have some programming experience so we won't cover CS-101.

Ideally, you also have some experience with low-level programming, because we will focus on "how to do it in Ada." If you do, feel free to gloss over the introductory material. If not, don't worry. We will cover enough for the course to be of value in any case.

34.1 So, what will we actually cover?

We will introduce you to using Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data, e.g., data from sensors that are occasionally faulty.

We will discuss development using more than Ada alone, nowadays a quite common approach. Specifically, how to interface with code and data written in other languages, and how (and why) to work with assembly language.

Embedded systems interact with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. Frequently these devices are mapped into the target memory address space. We will cover how to define and interact with these memory-mapped devices.

Finally, we will show how to handle interrupts in Ada, using portable constructs.
34.2 Definitions

Before we go any further, what do we mean by "embedded system" anyway? It's time to be specific. We're talking about a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system: the enclosing thermostat controlling the temperature, the power steering controller in the enclosing automobile, and the FADEC embedded in the enclosing aircraft. So these are not stand-alone computers for general purpose application execution.

As such, embedded systems typically have reduced resources available, especially power, which means reduced processor speed and reduced memory on-board. For an example at the small end of the spectrum, consider the computer embedded in a wearable device: it must run for a long time on a very little battery, with comparatively little memory available. But that's often true of bigger systems too, such as systems on aircraft where power (and heat) are directly limiting factors.

As a result, developing embedded systems software can be more difficult than general application development, not to mention that this software is potentially safety-critical. Ada is known for use in very large, very long-lived projects (e.g., deployed for decades), but it can also be used for very small systems with tight resource constraints. We'll show you how.

We used the term "computer" above. You already know what that means, but you may be thinking of your laptop or something like that, where the processor, memory, and devices are all distinct, separate components. That can be the case for embedded systems too, albeit in a different form-factor such as rack-mounted boards. However, be sure to expand your definition to include the notion of a system-on-chip (SoC), in which the processor, memory, and various useful devices are all on a single chip. Embedded systems don't necessarily involve SoC computers but they frequently do. The techniques and information in this course work on any of these kinds of computer.

34.3 Down To The Bare Metal

Ada has always had facilities designed specifically for embedded systems. The language includes constructs for directly manipulating hardware, for example, and direct interaction with assembly language. These constructs are as effective as those of any high-level programming language (yes, including C). These constructs are expressively powerful, well-specified (so there are few surprises), efficient, and portable (within reason).

We say "within reason" because portability is a difficult goal for embedded systems. That's because the hardware is so much a part of the application itself, rather than being abstracted away as in a general-purpose application. That said, the hardware details can be managed in Ada so that portability is maximized to the extent possible for the application.

But strictly speaking, not all software can or should be absolutely portable! If a specific device is required, well, the program won't work with some other device. But to the extent possible portability is obviously a good thing.
34.4 The Ada Drivers Library

Speaking of SoC computers, there is a library of freely-available device drivers in Ada. Known as the Ada Driver Library (ADL), it supports many devices on a number of vendors' products. Device drivers for timers, I2C, SPI, A/D and D/A converters, DMA, General Purpose I/O, LCD displays, sensors, and other devices are included. The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.

An extensive description of a project using the ADL is available here: https://blog.adacore.com/making-an-rc-car-with-ada-and-spark

We will refer to components of this library and use some of them as examples.
LOW LEVEL PROGRAMMING

This section introduces a number of topics in low-level programming, in which the hardware and the compiler's representation choices are much more in view at the source code level. In comparatively high level code these topics are "abstracted away" in that the programmer can assume that the compiler does whatever is necessary on the current target machine so that their code executes as intended. That approach is not sufficient in low-level programming.

Note that we do not cover every possibility or language feature. Instead, we cover the necessary concepts, and also potential surprises or pitfalls, so that the parts not covered can be learned on your own.

35.1 Separation Principle

There is a language design principle underlying the Ada facilities intended for implementing embedded software. This design principle directly affects how the language is used, and therefore, the portability and readability of the resulting application code.

This language design principle is known as the "separation principle." What's being separated? The low-level, less portable aspects of some piece of code are separated from the usage of that piece of code.

Don't confuse this with hiding unnecessary implementation details via compile-time visibility control (i.e., information hiding and encapsulation). That certainly should be done too. Instead, because of the separation principle, we specify the low-level properties of something once, when we declare it. From then on, we can use regular Ada code to interact with it. That way the bulk of the code — the usage — is like any other Ada code, and doesn't propagate the low-level details all over the client code. This greatly simplifies usage and understandability as well as easing porting to new hardware-specific aspects. You change things in one place, rather than everywhere.

For example, consider a device mapped to the memory address space of the processor. To interact with the device we interact with one or more memory cells. Reading input from the device amounts to reading the value at the associated memory location. Likewise, sending output to the device amounts to writing to that location.

To represent this device mapping we declare a variable of an appropriate type and specify the starting address the object should occupy. (There are other ways too, but for a single, statically mapped object this is the simplest approach.) We'd want to specify some other characteristics as well, but let's focus on the address.
If the hardware presents an interface consisting of multiple fields within individual memory cells, we can use a record type instead of a single unsigned type representing a single word. Ada allows us to specify the exact record layout, down to the individual bit level, for any types we may need to use for the record components. When we declare the object we use that record type, again specifying the starting address. Then we can just refer to the object's record components as usual, having the compiler compute the address offsets required to access the components representing the individual hardware fields.

Note that we aren't saying that other languages cannot do this too. Many can, using good programming practices. What we're saying is that those practices are designed into the Ada way of doing it.

### 35.2 Guaranteed Level of Support

The Ada reference manual has an entire section dedicated to low-level programming. That's section 13, "Representation Issues," which provides facilities for developers to query and control aspects of various entities in their code, and for interfacing to hardware. Want to specify the exact layout for a record type's components? Easy, and the compiler will check your layout too. Want to specify the alignment of a type? That's easy too. And that's just the beginning. We'll talk about these facilities as we go, but there's another point to make about this section.

In particular, section 13 includes recommended levels of support to be provided by language implementations, i.e., compilers and other associated tools. Although the word "recommended" is used, the recommendations are meant to be followed.

For example, section 13.3 says that, for some entity named X, "X'Address should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose Address has been specified." So, for example, if the programmer specifies the address for a memory-mapped variable, the compiler cannot ignore that specification and instead, for the sake of performance, represent that variable using a register. The object must be represented as an addressable entity, as requested by the programmer. (Registers are not addressable.)

We mention this because, although the recommended levels of support are intended to be followed, those recommendations become requirements if the Systems Programming (SP) Annex is implemented by the vendor. In that case the vendor’s implementation of section 13 must support at least the recommended levels. The SP Annex defines additional, optional functionality oriented toward this programming domain; you want it anyway. (Like all the annexes it adds no new syntax.) Almost all vendors, if not literally all, implement the Annex so you can rely on the recommended levels of support.
35.3 Querying Implementation Limits and Characteristics

Sometimes you need to know more about the underlying machine than is typical for general purpose applications. For example, your numerical analysis algorithm might need to know the maximum number of digits of precision that a floating-point number can have on this specific machine. For networking code, you will need to know the "endianness" of the machine so you can know whether to swap the bytes in an Ethernet packet. You'd go look in the limits.h file in C implementations, but in Ada we go to a package named System to get this information.

Clearly, these implementation values will vary with the hardware, so the package declares constants with implementation-defined values. The names of the constants are what's portable, you can count on them being the same in any Ada implementation.

However, vendors can add implementation-defined declarations to the language-defined content in package System. You might require some of those additions, but portability could then suffer when moving to a new vendor's compiler. Try not to use them unless it is unavoidable. Ideally these additions will appear in the private part of the package, so the implementation can use them but application code cannot.

For examples of the useful, language-defined constants, here are those for the numeric limits of an Ada compiler for an Arm 32-bit SoC:

```ada
Min_Int  :  constant := Long_Long_Integer'First;
Max_Int  :  constant := Long_Long_Integer'Last;
Max_Binary_Modulus : constant := 2 ** Long_Long_Integer'Size;
Max_Nonbinary_Modulus : constant := 2 ** Integer'Size - 1;
Max_Base_Digits : constant := Long_Long_Float'Digits;
Max_Digits : constant := Long_Long_Float'Digits;
Max_Mantissa : constant := 63;
Fine_Delta : constant := 2.0 ** (-Max_Mantissa);
```

Min_Int and Max_Int supply the most-negative and most-positive integer values supported by the machine.

Max_Binary_Modulus is the largest power of two allowed as the modulus of a modular type definition.

But a modular type need not be defined in terms of powers of two. An arbitrary modulus is allowed, as long as it is not bigger than the machine can handle. That's specified by Max_Nonbinary_Modulus, the largest non-power-of-two value allowed as the modulus of a modular type definition.

Max_Base_Digits is the largest value allowed for the requested decimal precision in a floating-point type's definition.

We won't go over all of the above, you get the idea. Let's examine the more important contents.

Two of the most frequently referenced constants in System are the following, especially the first. (The values here are again for the Arm 32-bit SoC):

```ada
Storage_Unit : constant := 8;
Word_Size : constant := 32;
```

Storage_Unit is the number of bits per memory storage element. Storage elements are the components of memory cells, and typically correspond to the individually addressable
memory elements. A "byte" would correspond to a storage element with the above constant value.

Consider a typical idiom for determining the number of whole storage elements an object named \texttt{X} occupies:

\begin{verbatim}
Units : constant Integer := (X'Size + Storage_Unit - 1) / Storage_Unit;
\end{verbatim}

Remember that \texttt{Size} returns a value in terms of bits. There are more direct ways to determine that size information but this will serve as an example of the sort of thing you might do with that constant.

A machine "word" is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of some number of storage elements, maybe one but typically more than one. As the unit the machine natively manipulates, words are expected to be independently addressable. (On some machines only words are independently addressable.)

\texttt{Word_Size} is the number of bits in the machine word. On a 32-bit machine we'd expect \texttt{Word_Size} to have a value of 32; on a 64-bit machine it would probably be 64, and so on.

\texttt{Storage_Unit} and \texttt{Word_Size} are obviously related.

Another frequently referenced declaration in package \texttt{System} is that of the type representing memory addresses, along with a constant for the null address designating no storage element.

\begin{verbatim}
type Address is private;
Null_Address : constant Address;
\end{verbatim}

You may be wondering why type \texttt{Address} is a private type, since that choice means that we programmers cannot treat it like an ordinary (unsigned) integer value. Portability is of course the issue, because addressing, and thus address representation, varies among computer architectures. Not all architectures have a flat address space directly referenced by numeric values, although that is common. Some are represented by a base address plus an offset, for example. Therefore, the representation for type \texttt{Address} is hidden from us, the clients. Consequently we cannot simply treat address values as numeric values. Don't worry, though. The operations we need are provided.

Package \texttt{System} declares these comparison functions, for example:

\begin{verbatim}
function "<" (Left, Right : Address) return Boolean;
function "<=" (Left, Right : Address) return Boolean;
function ">" (Left, Right : Address) return Boolean;
function ">=" (Left, Right : Address) return Boolean;
function ":=" (Left, Right : Address) return Boolean;
\end{verbatim}

These functions are intrinsic, i.e., built-in, meaning that the compiler generates the code for them directly at the point of calls. There is no actual function body for any of them so there is no performance penalty.

Any private type directly supports the equality function, and consequently the inequality function, as well as assignment. What we don't get here is address arithmetic, again because we don't have a compile-time view of the actual representation. That functionality is provided by package \texttt{System.Storage_Elements}, a child package we will cover later. We should say though, that the need for address arithmetic in Ada is rare, especially compared to C.

Having type \texttt{Address} presented as a private type is not, strictly speaking, required by the language. Doing so is a good idea for the reasons given above, and is common among vendors. Not all vendors do, though.

Note that \texttt{Address} is the type of the result of the query attribute \texttt{Address}. 
We mentioned potentially needing to swap bytes in networking communications software, due to the differences in the "endianness" of the machines communicating. That characteristic can be determined via a constant declared in package System as follows:

```ada
type Bit_Order is (High_Order_First, Low_Order_First);
Default_Bit_Order : constant Bit_Order := implementation-defined;
```

High_Order_First corresponds to "Big Endian" and Low_Order_First to "Little Endian." On a Big Endian machine, bit 0 is the most significant bit. On a Little Endian machine, bit 0 is the least significant bit.

Strictly speaking, this constant gives us the default order for bits within storage elements in record representation clauses, not the order of bytes within words. However, we can usually use it for the byte order too. In particular, if Word_Size is greater than Storage_Unit, a word necessarily consists of multiple storage elements, so the default bit ordering is the same as the ordering of storage elements in a word.

Let's take that example of swapping the bytes in a received Ethernet packet. The "wire" format is Big Endian so if we are running on a Little Endian machine we must swap the bytes received.

Suppose we want to retrieve typed values from a given buffer or bytes. We get the bytes from the buffer into a variable named Value, of the type of interest, and then swap those bytes within Value if necessary.

```ada
begin
  Value := ...
  if Default_Bit_Order /= High_Order_First then
    -- we're not on a Big Endian machine
    Value := Byte_Swapped (Value);
  end if;
end Retrieve_4_Bytes;
```

We have elided the code that gets the bytes into Value, for the sake of simplicity. How the bytes are actually swapped by function Byte_Swapped is also irrelevant. The point here is the if-statement: the expression compares the Default_Bit_Order constant to High_Order_First to see if this execution is on a Big Endian machine. If not, it swaps the bytes because the incoming bytes are always received in "wire-order," i.e., Big Endian order.

Another important set of declarations in package System define the values for priorities, including interrupt priorities. We will ignore them until we get to the section on interrupt handling.

Finally, and perhaps surprisingly, a few declarations in package System are almost always (if not actually always) ignored.

```ada
type Name is implementation-defined enumeration type;
System_Name : constant Name := implementation-defined;
```

Values of type Name are the names of alternative machine configurations supported by the implementation. System_Name represents the current machine configuration. We've never seen any actual use of this.

Memory_Size is an implementation-defined value that is intended to reflect the memory size of the configuration, in units of storage elements. What the value actually refers to is not specified. Is it the size of the address space, i.e., the amount possible, or is it the amount of physical memory actually on the machine, or what? In any case, the amount of memory available to a given computer is neither dependent upon, nor reflected by, this constant. Consequently, Memory_Size is not useful either.
Why have something defined in the language that nobody uses? In short, it seemed like a good idea at the time when Ada was first defined. Upward-compatibility concerns propagate these declarations forward as the language evolves, just in case somebody does use them.

### 35.4 Querying Representation Choices

As we mentioned in the introduction, in low-level programming the hardware and the compiler's representation choices can come to the forefront. You can, therefore, query many such choices.

For example, let's say we want to query the addresses of some objects because we are calling the imported C `memcpy` function. That function requires two addresses to be passed to the call: one for the source, and one for the destination. We can use the `Address` attribute to get those values.

We will explore importing routines and objects implemented in other languages elsewhere. For now, just understand that we will have an Ada declaration for the imported routine that tells the compiler how it should be called. Let's assume we have an Ada function declared like so:

```ada
function MemCopy(Destination : System.Address; Source : System.Address; Length : Natural) return System.Address with Import, Convention => C, Link_Name => "memcpy", Pre => Source /= Null_Address and then Destination /= Null_Address and then not Overlapping (Destination, Source, Length), Post => MemCopy'Result = Destination;
-- Copies Length bytes from the object designated by Source to the object designated by Destination.
```

The three aspects that do the importing are specified after the reserved word `with` but can be ignored for this discussion. We'll talk about them later. The preconditions make explicit the otherwise implicit requirements for the arguments passed to `memcpy`, and the postcondition specifies the expected result returned from a successful call. Neither the preconditions nor the postconditions are required for importing external entities but they are good "guard-rails" for using those entities. If we call it incorrectly the precondition will inform us, and likewise, if we misunderstand the result the postcondition will let us know (at least to the extent that the return value does that).

For a sample call to our imported routine, imagine that we have a procedure that copies the bytes of a `String` parameter into a `Buffer` parameter, which is just a contiguous array of bytes. We need to tell `MemCopy` the addresses of the arguments passed so we apply the `Address` attribute accordingly:

```ada
procedure Put (This : in out Buffer; Start : Index; Value : String) is
  Result : System.Address with Unreferenced;
begin
  Result := MemCopy (Destination => This (Start)'Address,
                     Source => Value'Address,
                     Length => Value'Length);
end Put;
```

The order of the address parameters is easily confused so we use the named association format for specifying the actual parameters in the call.
Although we assign Result we don't otherwise use it, so we tell the compiler this is not a mistake via the Unreferenced aspect. And if we do turn around and reference it the compiler will complain, as it should. Note that Unreferenced is defined by GNAT, so usage is not necessarily portable. Other vendors may or may not implement something like it, perhaps with a different name.

(We don't show the preconditions for Put, but they would have specified that Start must be a valid index into this particular buffer, and that there must be room in the Buffer argument for the number of bytes in Value when starting at the Start index, so that we don't copy past the end of the Buffer argument.)

There are other characteristics we might want to query too.

We might want to ask the compiler what alignment it chose for a given object (or type, for all such objects).

For a type, when Alignment returns a non-zero value we can be sure that the compiler will allocate storage for objects of the type at correspondingly aligned addresses (unless we force it to do otherwise). Similarly, references to dynamically allocated objects of the type will be to properly aligned locations. Otherwise, an Alignment of zero means that the guarantee does not hold. That could happen if the type is packed down into a composite object, such as an array of Booleans. We'll discuss "packing" soon. More commonly, the smallest likely value is 1, meaning that any storage element's address will suffice. If the machine has no particular natural alignments, then all type alignments will probably be 1 by default. That would be somewhat rare today, though, because modern processors usually have comparatively strict alignment requirements.

We can ask for the amount of storage associated with various entities. For example, when applied to a task, 'Storage_Size tells us the number of storage elements reserved for the task's execution. The value includes the size of the task's stack, if it has one. We aren't told if other required storage, used internally in the implementation, is also included in this number. Often that other storage is not included in this number, but it could be.

Storage_Size is also defined for access types. The meaning is a little complicated. Access types can be classified into those that designate only variables and constants ("access-to-object") and those that can designate subprograms. Each access-to-object type has an associated storage pool. The storage allocated by new comes from the pool, and instances of Unchecked_Deallocation return storage to the pool.

When applied to an access-to-object type, Storage_Size gives us the number of storage elements reserved for the corresponding pool.

Note that Storage_Size doesn't tell us how much available, unallocated space remains in a pool. It includes both allocated and unallocated space. Note, too, that although each access-to-object type has an associated pool, that doesn't mean that each one has a distinct, dedicated pool. They might all share one, by default. On an operating system, such as Linux, the default shared pool might even be implicit, consisting merely of calls to the OS routines in C.

As a result, querying Storage_Size for access types and tasks is not necessarily all that useful. Specifying the sizes, on the other hand, definitely can be useful.

That said, we can create our own pool types and define precisely how they are sized and how allocation and deallocation work, so in that case querying the size for access types could be more useful.

For an array type or object, 'Component_Size provides the size in bits of the individual components.

More useful are the following two attributes that query a degree of memory sharing between objects.

Applied to an object, 'Has_Same_Storage is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations occupy exactly the same bits.
Applied to an object, 'Overlaps_Storage is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations share at least one bit.

Generally, though, we specify representation characteristics far more often than we query them. Rather than describe all the possibilities, we can just say that all the representation characteristics that can be specified can also be queried. We cover specifying representation characteristics next, so just assume the corresponding queries are available.

That said, there is one particular representation query we need to talk about explicitly, now, because there is a lot of confusion about it: the 'Size attribute. The confusion stems from the fact that there are multiple contexts for applying the attribute, and multiple reasonable interpretations possible. We can apply the 'Size attribute to a type, in an attempt to get information about all objects of the type, or we can apply it to individual objects to get specific information. In both cases, what actual information do we get? In the original version of Ada these questions weren't really answered so vendors did what they thought was correct. But they did not agree with each other, and portability became a problem.

For example, suppose you want to convert some value to a series of bytes in order to send the value over the wire. To do that you need to know how many bytes are required to represent the value. Many applications queried the size of the type to determine that, and then, when porting to a new vendor's compiler, found that their code no longer worked correctly. The new vendor's implementation wasn't wrong, it was just different.

Later versions of Ada answered these questions, where possible, so let's examine the contexts and meaning. Above all, though, remember that 'Size returns values in terms of bits.

If we apply 'Size to a type, the resulting value depends on the kind of type.

For scalar types, the attribute returns the minimum number of bits required to represent all the values of the type. Here's a diagram showing what the category "scalar types" includes:

Consider type Boolean, which has two possible values. One bit will suffice, and indeed the language standard requires Boolean 'Size to be the value 1.

This meaning also applies to subtypes, which can constrain the number of values for a scalar type. Consider subtype Natural. That's a subtype defined by the language to be type Integer but with a range of 0..Integer'Last. On a 32-bit machine we would expect Integer to be a native type, and thus 32-bits. On such a machine if we say Integer'Size
we will indeed get 32. But if we say `Natural'Size` we will get 31, not 32, because only 31 bits are needed to represent that range on that machine.

The size of objects, on the other hand, cannot be just a matter of the possible values. Consider type `Boolean` again, where `Boolean'Size` is required to be 1. No compiler is likely to allocate one bit to a `Boolean` variable, because typical machines don't support individually-addressable bits. Instead, addresses refer to storage elements, of a size indicated by the `Storage_Unit` constant. The compiler will allocate the smallest number of storage elements necessary, consistent with other considerations such as alignment. Therefore, for a machine that has `Storage_Unit` set to a value of eight, we can assume that a compiler for that machine will allocate an entire eight-bit storage element to a stand-alone `Boolean` variable. The other seven bits are simply not used by that variable. Moreover, those seven bits are not used by any other stand-alone object either, because access would be far less efficient, and such sharing would require some kind of locking to prevent tasks from interfering with each other when accessing those stand-alone objects. (Stand-alone objects are independently addressable; they wouldn't stand alone otherwise.)

By the same token (and still assuming a 32-bit machine), a compiler will allocate more than 31 bits to a variable of subtype `Natural` because there is no 31-bit addressable unit. The variable will get all 32-bits.

Note that we're talking about individual, stand-alone variables. Components of composite types, on the other hand, might indeed share bytes if the individual components don't require all the bits of their storage elements. You'd have to request that representation, though, with most implementations, because accessing the components at run-time would require more machine instructions. We'll go into the details of that later.

Let's talk further about sizes of types.

For record types, `'Size` gives the minimum number of bits required to represent the whole composite value. But again, that's not necessarily the number of bits required for the objects' in-memory representation. The order of the components within the record can make a difference, as well as their alignments. The compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. As a result the overall size could be larger.

Note that Ada compilers are allowed to reorder the components; the order in memory might not match the order in the source code.

For example, consider this record type and its components:

```ada
type My_Int is range 1..10;
subtype S is Integer range 1..10;

type R is record
    M : My_Int;
    X : S;
    B : Boolean;
    C : Character;
end record;
```

In the figure, we see a record type with some components, and a sample layout for that record type assuming the compiler does not reorder the components. Observe that some bytes allocated to objects of type R are unused (the darkly shaded ones). In this case that's because the alignment of subtype S happens to be 4 on this machine. The component X of that subtype S cannot start at byte offset 1, or 2, or 3, because those addresses would not satisfy the alignment constraint of S. (We're assuming byte 0 is at a word-aligned address.) Therefore, X starts at the object's starting address plus 4. Components B and C are of types

---

35.4. Querying Representation Choices 1125
that have an alignment of 1, so they can start at any storage element. They immediately follow the bytes allocated to component X. Therefore, R'\textit{Size} is 80, or 10 bytes. The three bytes following component M are simply not used.

But what about the two bytes following the last component C? They could be allocated to stand-alone objects if they would fit. More likely, though, the compiler will allocate those two bytes to objects of type R, that is, 12 bytes instead of 10 are allocated. As a result, 96 bits are actually used in memory. The extra, unused 16 bits are "padding."

Why add unused padding? It simplifies the memory allocation of objects of type R. Suppose some array type has components of record type R. Assuming the first component is aligned properly, every following component will also be aligned properly, automatically, because the two padding bytes are considered parts of the components.

To make that work, the compiler takes the most stringent alignment of all the record type's components and uses that for the alignment of the overall record type. That way, any address that satisfies the record object's alignment will satisfy the components' alignment requirements. The alignment is component X, of subtype S, is 4. The other components have an alignment of 1, therefore R'\textit{Alignment} is 4. An aligned address plus 12 will also be an aligned address.

This rounding up based on alignment is recommended behavior for the compiler, not a requirement, but is reasonable and typical among vendors. Although it can result in unused storage, that's the price paid for speed of access (or even correctness for machines that would fault on misaligned component accesses).

As you can see, alignment is a critical factor in the sizes of composite objects. If you care about the layout of the type you very likely need to care about the alignment of the components and overall record type.

Ada compilers are allowed to reorder the components of record types in order to minimize these gaps or satisfy the alignment requirements of the components. Some compilers do, some don't. Consider the type R again, this time with the first two components switched in the component declaration order:

```ada
type My_Int is range 1..10;
subtype S is My_Int.

type R is record
  X : S;
  M : My_Int;
  B : Boolean;
  C : Character;
end record;
```

Now R'\textit{Size} will report 56 bits instead of 80. The one trailing byte will still be padding, but only that one.

What about unbounded types, for example type \textit{String}? Querying the \textit{Size} in that case would provide an implementation-defined result. A somewhat silly thing to do, really, since the type — by definition — doesn't specify how many components are involved.

Usually, though, you don't want to query the size of a type. Most of the time what you want is the size of objects of the type. Going back to sending values over the wire, the code should query the size of the parameter holding the value to be sent. That will tell you how many bits are really needed.

One last point: GNAT, and now Ada 202x, define an attribute named \texttt{Object_Size}. It does just what the name suggests: what \textit{Size} does when applied to objects rather than types.
GNAT also defines another attribute, named Value_Size, that does what 'Size does when applied to types. The former is far more useful so Ada has standardized it.

### 35.5 Specifying Representation

Recall that we said Boolean 'Size is required to be 1, and that stand-alone objects of type Boolean are very likely allocated some integral number of storage elements (e.g., bytes) in memory, typically one. What about arrays of Booleans? Suppose we have an array of 16 Boolean components. How big are objects of the array type? It depends on the machine. Continuing with our hypothetical (but typical) byte-addressable machine, for the sake of efficient access each component is almost certainly allocated an individual byte rather than a single bit, just like stand-alone objects. Consequently, our array of 16 Booleans will be reported by 'Size to be 128 bits, i.e., 16 bytes. If you wanted a bit-mask, in which each Boolean component is allocated a single bit and the total array size is 16 bits, you'd have a problem. The compiler assumes you want speed of access rather than storage minimization, and normally that would be the right assumption.

Naturally there is a solution. Ada allows us to specify the representation characteristics of types, and thus objects of those types, including their bit-wise layouts. It also allows us to specify the representation of individual objects. You should understand, though, that the compiler is not required to do what you ask, because you might ask the impossible. For example, if you specify that the array of 16 Booleans is to be represented completely in 15 bits, what can the compiler do? Rejecting that specification is the only reasonable response. But if you specify something possible, the compiler must do what you ask, absent some compelling reason to the contrary.

With that in mind, let's examine setting the size for types.

So, how do we specify that we want our array of 16 Boolean components to be allocated one bit per component, for a total allocation of 16 bits? There are a couple of ways, one somewhat better than the other.

First, you can ask that the compiler "pack" the components into as small a number of bits as it can:

```ada
type Bits16 is array (0 .. 15) of Boolean with Pack;
```

That likely does what you want: Bits16'Size will probably be 16.

But realize that the Pack aspect (and corresponding pragma) is merely a request that the compiler do its best to minimize the number of bits allocated, not necessarily that it do exactly what you expected or required.

We could set the size of the entire array type:

```ada
type Bits16 is array (0 .. 15) of Boolean with Size => 16;
```

But the language standard says that a Size clause on array and record types should not affect the internal layout of their components. That's Implementation Advice, so not normative, but implementations are really expected to follow the advice, absent some compelling reason. That's what the Pack aspect, record representation clauses, and Component_Size clauses are for. (We'll talk about record representation clauses momentarily.) That said, at least one other vendor's compiler would have changed the size of the array type because of the Size clause, so GNAT defines a configuration pragma named Implicit_Packing that overrides the default behavior. With that pragma applied, the Size clause would compile and suffice to make the overall size be 16. That's a vendor-defined pragma though, so not portable.
Therefore, the best way to set the size for the array type is to set the size of the individual components, via the Component_Size aspect as the Implementation Advice indicates. That will say what we really want, rather than a "best effort" request for the compiler, and is portable:

\[
\text{type Bits16 is array (0 .. 15) of Boolean with Component_Size => 1;}
\]

With this approach the compiler must either use the specified size for each component or refuse to compile the code. If it compiles, objects of the array type will be 16 bits total (plus any padding bits required to make objects have a size that is a multiple of Storage_Unit, typically zero on modern machines).

Now that we have a bit-mask array type, let's put it to use.

Let's say that you have an object that is represented as a simple signed integer because, for most usage, that's the appropriate representation. Sometimes, though, let's say you need to access individual bits of the object instead of the whole numeric value. Signed integer types don't provide bit-level access. In Ada we'd say that the "view" presented by the object's type doesn't include bit-oriented operations. Therefore, we need to add a view to the object that does provide them. A different view will require an additional type for the same object.

Applying different types, and thus their operations, to the same object is known as \textit{type punning}\textsuperscript{268} in computer programming. Realize that doing so circumvents the static strong typing we harness to protect us from ourselves and from others. Use it with care! (For example, limit the compile-time visibility to such code.)

One way to add a view is to express an "overlay," in which an object of one type is placed at the same memory location as a distinct object of a different type, thus "overlaying" one object over the other in memory. The different types present different views, therefore different operations available for the shared memory cells. Our hypothetical example uses two views, but you can overlay as many different views as needed. (That said, requiring a large number of different views of the same object would be suspect.)

There are other ways in Ada to apply different views, some more flexible than others, but an overlay is a simple one that will often suffice.

Here is an implementation of the overlay approach, using our bit-mask array type:

\[
\text{type Bits32 is array (0 .. 31) of Boolean with Component_Size => 1;}
\]

\[
X : \text{Integer};
\]

\[
Y : \text{Bits32 with Address => X'Address;}
\]

We can query the addresses of objects, and other things too, but objects, especially vari-ables, are the most common case. In the above, we say X'Address to query the starting address of object X. With that information we know what address to specify for our bit-mask overlay object Y. Now X and Y are aliases for the same memory cells, and therefore we can manipulate and query that memory as either a signed integer or as an array of bits. Reading or updating individual array components accesses the individual bits of the overlaid object.

Instead of the Bits32 array type, we could have specified a modular type for the overlay Y to get a view providing bit-oriented operations. Overlaying such an array was a common idiom prior to the introduction of modular "unsigned" types in Ada, and remains useful for accessing individual bits. In other words, using a modular type for Y, you could indeed access an individual bit by passing a mask value to the \texttt{and} operator defined in any modular type's view. Using a bit array representation lets the compiler do that work for you, in the generated code. The source code will be both easier to read and more explicit about what it is doing when using the bit array overlay.

\textsuperscript{268} https://en.wikipedia.org/wiki/Type_punning
One final issue remains: in our specific overlay example the compiler would likely generate code that works. But strictly speaking it might not.

The Ada language rules say that for such an overlaid object — Y in the example above — the compiler should not perform optimizations regarding Y that it would otherwise apply in the absence of aliases. That's necessary, functionally, but may imply degraded performance regarding Y, so keep it in mind. Aliasing precludes some desirable optimizations.

But what about X in the example above? We're querying that object's address, not specifying it, so the RM rule precluding optimizations doesn't apply to X. That can be problematic.

The compiler might very well place X in a register, for example, for the sake of the significant performance increase (another way of being friendly). But in that case System. Null_ADDRESS will be returned by the X'Address query and, consequently, the declaration for Y will not result in the desired overlaying.

Therefore, we should mark X as explicitly aliased to ensure that X'Address is well-defined:

```ada
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;
X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

The only difference in the version above is the addition of aliased in the declaration of X. Now we can be certain that the optimizer will not represent X in some way incompatible with the idiom, and X'Address will be well-defined.

In our example X and Y are clearly declared in the same compilation unit. Most compilers will be friendly in this scenario, representing X in such a way that querying the address will return a non-null address value even if aliased is not applied. Indeed, aliased is relatively new to Ada, and earlier compilers typically emitted code that would handle the overlay as intended.

But suppose, instead of being declared in the same declarative part, that X was declared in some other compilation unit. Let's say it is in the visible part of a package declaration. (Assume X is visible to clients for some good reason.) That package declaration can be, and usually will be, compiled independently of clients, with the result that X might be represented in some way that cannot supporting querying the address meaningfully.

Therefore, the declaration of X in the package spec should be marked as aliased, explicitly:

```ada
package P is
  X : aliased Integer;
end P;
```

Then, in the client code declaring the overlay, we only declare Y, assuming a with-clause for P:

```ada
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;
Y : Bits32 with Address => P.X'Address;
```

All well and good, but how did the developer of the package know that some other unit, a client of the package, would query the address of X, such that it needed to be marked as aliased? Indeed, the package developer might not know. Yet the programmer is responsible for ensuring a valid and appropriate Address value is used in the declaration of Y. Execution is erroneous otherwise, so we can't say what would happen in that case. Maybe an exception is raised or a machine trap, maybe not.

Worse, the switches that were applied when compiling the spec for package P can make a difference: P.X might not be placed in a register unless the optimizer is enabled. Hence the client code using Y might work as expected when built for debugging, with the optimizer
disabled, and then not do so when re-built for the final release. You'd probably have to solve this issue by debugging the application.

On a related note, you may be asking yourself how to know that type `Integer` is 32 bits wide, so that we know what size array to use for the bit-mask. The answer is that you just have to know the target well when doing low-level programming. The hardware becomes much more visible, as we mentioned.

That said, you could at least verify the assumption:

```ada
pragma Compile_Time_Error (Integer'Object_Size /= 32,
                          "Integers expected to be 32 bits");
X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

That's a vendor-defined pragma so this is not fully portable. It isn't an unusual pragma, though, so at least you can probably get the same functionality even if the pragma name varies.

Overlays aren't always structured like our example above, i.e., with two objects declared at the same time. We might apply a different type to the same memory locations at different times. Here's an example from the ADL to illustrate the idea. We'll elaborate on this example later, in another section.

First, a package declaration, with two functions that provide a device-specific unique identifier located in shared memory. Each function provides the same Id value in a distinct format. One format is a string of 12 characters, the other is a sequence of three 32-bit values. Hence both representations are the same size.

```ada
package STM32.Device_Id is

  subtype Device_Id_Image is String (1 .. 12);

  function Unique_Id return Device_Id_Image;

  type Device_Id_Tuple is array (1 .. 3) of UInt32
    with Component_Size => 32;

  function Unique_Id return Device_Id_Tuple;

end STM32.Device_Id;
```

In the package body we implement the functions as two ways to access the same shared memory, specified by `ID_Address`:

```ada
with System;
package body STM32.Device_Id is

  ID_Address : constant System.Address := System'To_Address (16#1FFF_7A10#);

  function Unique_Id return Device_Id Image is
    Result : Device_Id_Image with Address => ID_Address, Import;
  begin
    return Result;
  end Unique_Id;

  function Unique_Id return Device_Id_Tuple is
    Result : Device_Id_Tuple with Address => ID_Address, Import;
  begin
    return Result;
  end Unique_Id;
```

(continues on next page)
System'To_Address is just a convenient way to convert a numeric value into an Address value. The primary benefit is that the call is a static expression, but we can ignore that here. Using Import is a good idea to ensure that the Ada code does no initialization of the object, since the value is coming from the hardware via the shared memory. Doing so may not be necessary, depending on the type used, but is a good habit to develop.

The point of this example is that we have one object declaration per function, of a type corresponding to the intended function result type. Because each function places their local object at the same address, they are still overlaying the shared memory.

Now let's return, momentarily, to setting the size of entities, but now let's focus on setting the size of objects.

We've said that the size of an object is not necessarily the same as the size of the object's type. The object size won't be smaller, but it could be larger. Why? For a stand-alone object or a parameter, most implementations will round the size up to a storage element boundary, or more, so the object size might be greater than that of the type. Think back to Boolean, where Size is required to be 1, but stand-alone objects are probably allocated 8 bits, i.e., an entire storage element (on our hypothetical byte-addressed machine).

Likewise, recall that numeric type declarations are mapped to underlying hardware numeric types. These underlying numeric types provide at least the capabilities we request with our type declarations, e.g., the range or number of digits, perhaps more. But the mapped numeric hardware type cannot provide less than requested. If there is no underlying hardware type with at least our requested capabilities, our declarations won't compile. That mapping means that specifying the size of a numeric type doesn't necessarily affect the size of objects of the type. That numeric hardware type is the size that it is, and is fixed by the hardware.

For example, let's say we have this declaration:

```ada
type Device_Register is range 0 .. 2**5 - 1 with Size => 5;
```

That will compile successfully, because there will be a signed integer hardware type with at least that range. (Not necessarily, legally speaking, but realistically speaking, there will be such a hardware type.) Indeed, it may be an 8-bit signed integer, in which case Device_Register'Size will give us 5, but objects of the type will have a size of 8, unavoidably, even though we set Size to 5.

The difference between the type and object sizes can lead to potentially problematic code:

```ada
type Device_Register is range 0 .. 2**8 - 1 with Size => 8;
My_Device : Device_Register
  with Address => To_Address (...);
```

The code compiles successfully, and tries to map a byte to a hardware device that is physically connected to one storage element in the processor memory space. The actual address is elided as it is not important here.

That code might work too, but it might not. We might think that My_Device'Size is 8, and that My_Device'Address points at an 8-bit location. However, this isn't necessarily so, as we saw with the supposedly 5-bit example earlier. Maybe the smallest signed integer the hardware has is 16-bits wide. The code would compile because a 16-bit signed numeric type can certainly handle the 8-bit range requested. My_Device'Size would be then 16, and because 'Address gives us the starting storage element, My_Device'Address might designate the high-order byte of the overall 16-bit object. When the compiler reads the two bytes for My_Device what will happen? One of the bytes will be the data presented by
the hardware device mapped to the memory. The other byte will contain undefined junk, whatever happens to be in the memory cell at the time. We might have to debug the code a long time to identify that as the problem. More likely we'll conclude we have a failed device.

The correct way to write the code is to specify the size of the object instead of the type:

```ada
type Device_Register is range 0 .. 2**8 - 1;

My_Device : Device_Register with
  Size => 8,
  Address => To_Address (...);
```

If the compiler cannot support stand-alone 8-bit objects for the type, the code won't compile.

Alternatively, we could change the earlier Size clause on the type to apply Object_Size instead:

```ada
type Device_Register is range 0 .. 2**8 - 1 with Object_Size => 8;

My_Device : Device_Register with
  Address => To_Address (...);
```

The choice between the two approaches comes down to personal preference, at least if only a small number of stand-alone objects of the type are going to be declared. With either approach, if the implementation cannot support 8-bit stand-alone objects, we find out that there is a problem at compile-time. That's always cheaper than debugging.

You might conclude that setting the Size for a type serves no purpose. That's not an unreasonable conclusion, given what you've seen, but in fact there are reasons to do so. However, there are only a few specific cases so we will save the reasons for the discussions of the specific cases.

There is one general case, though, for setting the 'Size of a type. Specifically, you may want to specify the size that you think is the minimum possible, and you want the compiler to confirm that belief. This would be one of the so-called "confirming" representation clauses, in which the representation detail is what the compiler would have chosen anyway, absent the specification. You're not actually changing anything, you're just getting confirmation via Size whether or not the compiler accepts the clause. Suppose, for example, that you have an enumeration type with 256 values. For enumeration types, the compiler allocates the smallest number of bits required to represent all the values, rounded up to the nearest storage element. (It's not like C, where enums are just named int values.) For 256 values, an eight-bit byte would suffice, so setting the size to 8 would be confirming. But suppose we actually had 257 enumerals, accidentally? Our size clause set to 8 would not compile, and we'd be told that something is amiss.

However, note that if your supposedly "confirming" size clause actually specifies a size larger than what the compiler would have chosen, you won't know, because the compiler will silently accept sizes larger than necessary. It just won't accept sizes that are too small.

There are other confirming representation clauses as well. Thinking again of enumeration types, the underlying numeric values are integers, starting with zero and consecutively increasing from there up to N-1, where N is the total number of enumeral values.

For example:

```ada
type Commands is (Off, On);

for Commands use (Off => 0, On => 1);
```

As a result, Off is encoded as 0 and On as 1. That specific underlying encoding is guaranteed by the language, as of Ada 95, so this is just a confirming representation clause nowadays.
But it was not guaranteed in the original version of the language, so if you wanted to be sure of the encoding values you would have specified the above. It wasn't necessarily confirming before Ada 95, in other words.

But let's also say that the underlying numeric values are not what you want because you're interacting with some device and the commands are encoded with values other than 0 and 1. Maybe you want to use an enumeration type because you want to specify all the possible values actually used by clients. If you just used some numeric type instead and made up constants for 0n and 0ff, there's nothing to keep clients from using other numeric values in place of the two constants (absent some comparatively heavy code to prevent that from happening). Better to use the compiler to make that impossible in the first place, rather than debug the code to find the incorrect values used. Therefore, we could specify different encodings:

```ada
for Commands use (Off => 2, On => 4);
```

Now the compiler will use those encoding values instead of 0 and 1, transparently to client code.

The encoding values specified must maintain the relative ordering, otherwise the relational operators won't work correctly. For example, for type Commands above, Off is less than On, so the specified encoding value for Off must be less than that of On.

Note that the values given in the example no longer increase consecutively, i.e., there's a gap. That gap is OK, in itself. As long as we use the two numerals the same way we'd use named constants, all is well. Otherwise, there is both a storage issue and a performance issue possible. Let's say that we use that enumeration type as the index for an array type. Perfectly legal, but how much storage is allocated to objects of this array type? Enough for exactly two components? Four, with two unused? The answer depends on the compiler, and is therefore not portable. The bigger the gaps, the bigger the overall storage difference possible. Likewise, imagine we have a for-loop iterating over the index values of one of these array objects. The for-loop parameter cannot be coded by the compiler to start at 0, clearly, because there is no index (enumeration) value corresponding to 0. Similarly, to get the next index, the compiler cannot have the code simply increment the current value. Working around that takes some extra code, and takes some extra time that would not be required if we did not have the gaps.

The performance degradation can be significant compared to the usual code generated for a for-loop. Some coding guidelines say that you shouldn't use an enumeration representation clause for this reason, with or without gaps. Now that Ada has type predicates we could limit the values used by clients for a numeric type, so an enumeration type is not the only way to get a restricted set of named, encoded values.

```ada
type Commands is new Integer with
  Static_Predicate => Commands in 2 | 4;
On    : constant Commands := 2;
Off   : constant Commands := 4;
```

The storage and performance issues bring us back to confirming clauses. We want the compiler to recognize them as such, so that it can generate the usual code, thereby avoiding the unnecessary portability and performance issues. Why would we have such a confirming clause now? It might be left over from the original version of the language, written before the Ada 95 change. Some projects have lifetimes of several decades, after all, and changing the code can be expensive (certified code, for example). Whether the compiler does recognize confirming clauses is a feature of the compiler implementation. We can expect a mature compiler to do so, but there's no guarantee.

Now let's turn to what is arguably the most common representation specification, that of record type layouts.

Recall from the discussion above that Ada compilers are allowed to reorder record com-
ponents in physical memory. In other words, the textual order in the source code is not necessarily the physical order in memory. That's different from, say, C, where what you write is what you get, and you better know what you're doing. On some targets a misaligned struct component access will perform very poorly, or even trap and halt, but that's not the C compiler's fault. In Ada you'd have to explicitly specify the problematic layout. Otherwise, if compilation is successful, the Ada compiler must find a representation that will work, either by reordering the components or by some other means. Otherwise it won't compile.

GNAT did not reorder components until relatively recently but does now, at least for the more egregious performance cases. It does this reordering silently, too, although there is a switch to have it warn you when it does. To prevent reordering, GNAT defines a pragma named No_Component_Reorder that does what the name suggests. You can apply it to individual record types, or globally, as a configuration pragma. But of course because the pragma is vendor defined it is not portable.

Therefore, if you care about the record components' layout in memory, the best approach is to specify the layout explicitly. For example, perhaps you are passing data to code written in C. In that case, you need the component order in memory to match the order given in the corresponding C struct declaration. That order in memory is not necessarily guaranteed from the order in the Ada source code. The Ada compiler is allowed to chose the representation unless you specify it, and it might chose a different layout from the one given. (Ordinarily, letting the compiler chose the layout is the most desirable approach, but in this case we have an external layout requirement.)

Fortunately, specifying a record type's layout is straightforward. The record layout specification consists of the storage places for some or all components, specified with a record representation clause. This clause specifies the order, position, and size of components (including discriminants, if any).

The approach is to first define the record type, as usual, using any component order you like — you're about to specify the physical layout explicitly, in the next step.

Let's reuse that record type from the earlier discussion:

```ada
type My_Int is range 1 .. 10;
subtype S is Integer range 1 .. 10;
type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```

The resulting layout might be like so, assuming the compiler doesn't reorder the components:

```ada
type My_Int is range 1..10;
subtype S is Integer range 1..10;
type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```

Sample layout for a given compiler

If compiler allocates in declaration order

R'Size will be 80 bits (10 bytes)
but all 12 are allocated to objects
As a result, R'\textit{Size} will be 80 bits (10 bytes), but those last two bytes will be allocated to objects, for an \textit{Object Size} of 96 bits (12 bytes). We'll change that with an explicit layout specification.

Having declared the record type, the second step consists of defining the corresponding record representation clause giving the components' layout. The clause uses syntax that somewhat mirrors that of a record type declaration. The components' names appear, as in a record type declaration. But now, we don't repeat the components' types, instead we give their relative positions within the record, in terms of a relative offset that starts at zero. We also specify the bits we want them to occupy within the storage elements starting at that offset.

```ada
for R use record
  X at 0 range 0 .. 31; -- note the order swap,
  M at 4 range 0 .. 7; -- with this component
  B at 5 range 0 .. 7;
  C at 6 range 0 .. 7;
end record;
```

Now we'll get the optimized order, and we'll always get that order, or the layout specification won't compile in the first place. In the following diagram, both layouts, the default, and the one resulting from the record representation clause, are depicted for comparison:

```
\begin{figure}[h]
  \centering
  \includegraphics[width=0.7\textwidth]{example_diagram.png}
  \caption{Comparison of default and specified layouts.}
  \end{figure}
```

\textbf{type} My\_Int is range 1..10;

\textbf{subtype} S is Integer range 1..10;

\textbf{type} R is record
  M : My\_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;

```ada
for R use record
  X at 0 range 0 .. 31;
  M at 4 range 0 .. 7;
  B at 5 range 0 .. 7;
  C at 6 range 0 .. 7;
end record;
```

R'\textit{Size} will be 56 bits (7 bytes), but that last padding byte will also be allocated to objects, so the \textit{Object Size} will be 64 bits (8 bytes).

Notice how we gave each component an offset, after the reserved word \texttt{at}. These offsets are in terms of storage elements, and specify their positions within the record object as a whole. They are relative to the beginning of the memory allocated to the record object so they are numbered starting at zero. We want the X component to be the very first component in the allocated memory so the offset for that one is zero. The M component, in comparison, starts at an offset of 4 because we are allocating 4 bytes to the prior component X: bytes 0 through 3 specifically. M just occupies one storage element so the next
component, B, starts at offset 5. Likewise, component C starts at offset 6.

Note that there is no requirement for the components in the record representation clause to be in any particular textual order. The offsets alone specify the components' order in memory. A good style, though, is to order the components in the representation clause so that their textual order corresponds to their order in memory. Doing so facilitates our verifying that the layout is correct because the offsets will be increasing as we read the specification.

An individual component may occupy part of a single storage element, all of a single storage element, multiple contiguous storage elements, or a combination of those (i.e., some number of whole storage elements but also part of another). The bit "range" specifies this bit-specific layout, per component, by specifying the first and last bits occupied. The X component occupies 4 complete 8-bit storage elements, so the bit range is 0 through 31, for a total of 32 bits. All the other components each occupy an entire single storage element so their bit ranges are 0 through 7, for a total of 8 bits.

The text specifying the offset and bit range is known as a "component_clause" in the syntax productions. Not all components need be specified by component_clauses, but (not surprisingly) at most one clause is allowed per component. Really none are required but it would be strange not to have some. Typically, all the components are given positions. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation.

Components not otherwise given an explicit placement are given positions chosen by the compiler. We don't say that they "follow" those explicitly positioned because there's no requirement that the explicit positions start at offset 0, although it would be unusual not to start there.

Placements must not make components overlap, except for components of variant parts, a topic covered elsewhere. You can also specify the placement of implementation-defined components, as long as you have a name to refer to them. (In addition to the components listed in the source code, the implementation can add components to help implement what you wrote explicitly.) Such names are always attribute references but the specific attributes, if any, are implementation-defined. It would be a mistake for the compiler to define such implicit components without giving you a way to refer to them. Otherwise they might go exactly where you want some other component to be placed, or overlap that place.

The positions (offsets) and the bit numbers must be static, informally meaning that they are known at compile-time. They don't have to be numeric literals, though. Numeric constants would work, but literals are the most common by far.

Note that the language does not limit support for component clauses to specific component types. They need not be one of the integer types, in particular. For example, a position can be given for components that are themselves record types, or array types. Even task types are allowed as far as the language goes, although the implementation might require a specific representation, such as the component taking no bits whatsoever (0 .. 1). There are restrictions that keep things sane, for example rules about how a component name can be used within the overall record layout construct, but not restrictions on the types allowed for individual components. For example, here is a record layout containing a String component, arbitrarily set to contain 11 characters:

```ada
type R is record
  S : String (1 .. 11);
  B : Boolean;
end record;

for R use record
  S at 0 range 0 .. 87;
  B at 11 range 0 .. 7;
end record;
```
Component $S$ is to be the first component in memory in this example, hence the position offset is 0, for the first byte of $S$. Next, $S$ is 11 characters long, or 88 bits, so the bit range is 0 .. 87. That's 11 bytes of course, so $S$ occupies storage elements 0 .. 10. Therefore, the next component position must be at least 11, unless there is to be a gap, in which case it would be greater than 11. We'll place $B$ immediately after the last character of $S$, so $B$ is at storage element offset 11 and occupying all that one byte's bits.

We'll have more to say about record type layouts but first we need to talk about alignment.

Modern target architectures are comparatively strict about the address alignments for some of their types. If the alignment is off, an access to the memory for objects of the type can have highly undesirable consequences. Some targets will experience seriously degraded performance. On others, the target will halt altogether. As you can see, getting the alignment correct is a low-level, but vital, part of correct code on these machines.

Normally the compiler does this work for us, choosing an alignment that is both possible for the target and also optimal for speed of access. You can, however, override the compiler's alignment choice using an attribute definition clause or the `Alignment` aspect. You can do so on types other than record types, but specifying it on record types is typical. Here's our example record type with the alignment specified via the aspect:

```ada
type My_Int is range 1 .. 10;
subtype S is Integer range 1 .. 10;

type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record
  with
    Alignment => 1;
```

Alignment values are in terms of storage elements. The effect of the aspect or attribute clause is to ensure that the starting address of the memory allocated to objects of the type will be a multiple of the specified value.

In fact, whenever we specify a record type layout we really should also specify the record type's alignment, even though doing so is optional. Why? The alignment makes a difference in the overall record object's size. We've seen that already, with the padding bytes: the compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. The alignment also affects the size allocated to the record type even when the components are already aligned. As a result the overall size could be larger than we want for the sake of space. Additionally, when we pass such objects to code written in other languages, we want to ensure that the starting address of these objects is aligned as the external code expects. The compiler might not choose that required alignment by default.

Specifying alignment for record types is so useful that in the first version of Ada there was no syntax to specify alignment for anything other than record types (via the obsolete `at mod` clause on record representation clauses).

For that reason GNAT provides a pragma named `Optimize_Alignment`. This is a configuration pragma that affects the compiler's choice of default alignments where no alignment is explicitly specified. There is a time/space trade-off in the selection of these values, as we've seen. The normal choice tries to balance these two characteristics, but with an argument to the pragma you can give more weight to one or the other. The best approach is to specify the alignments explicitly, per type, for those that require specific alignment values. The pragma has the nice property of giving general guidance to the compiler for what should be done for the other types and objects not explicitly specified.
Now let's look into the details. We'll use a case study for this purpose, including specifying sizes as well as alignments.

The code for the case study is as follows. It uses Size clauses to specify the Sizes, instead of the Size aspect, just to emphasize that the Size clause approach is not obsolete.

```ada
package Some_Types is
  type Temperature is range -275 .. 1_000;
  type Identity is range 1 .. 127;
  type Info is record
    T : Temperature;
    Id : Identity;
  end record;

  for Info use record
    T at 0 range 0 .. 15;
    Id at 2 range 0 .. 7;
  end record;

  for Info'Size use 24;

  type List is array (1 .. 3) of Info;
  for List'Size use 24 * 3;
end Some_Types;
```

When we compile this, the compiler will complain that the size for List is too small, i.e., that the minimum allowed is 96 bits instead of the 72 we specified. We specified 24 * 3 because we said the record size should be 24 bits, and we want our array to contain 3 record components of that size, so 72 seems right.

What's wrong? As we've shown earlier, specifying the record type size doesn't necessarily mean that objects (in this case array components) are that size. The object size could be bigger than we specified for the type. In this case, the compiler says we need 96 total bits for the array type, meaning that each of the 3 array components is 32 bits wide instead of 24.

Why is it 32 bits? Because the alignment for Info is 2 (on this machine). The record alignment is a multiple of the largest alignment of the enclosed components. The alignment for type Temperature (2), is larger than the alignment for type Identity (1), therefore the alignment for the whole record type is 2. We need to go from that number of storage elements to a number of bits for the size.

Here's where it gets subtle. The alignment is in terms of storage elements. Each storage element is of a size in bits given by System.Storage_Unit. We've said that on our hypothetical machine Storage_Unit is 8, so storage elements are 8 bits wide on this machine. Bytes, in other words. Therefore, to get the required size in bits, we have to find a multiple of the two 8-bit bytes (specified by the alignment) that has at least the number of bits we gave in the Size clause. Two bytes only provides 16 bits, so that's not big enough, we need at least 24 bits. The next multiple of 2 bytes is 4 bytes, providing 32 bits, which is indeed larger than 24. Therefore, the overall size of the record type, consistent with the alignment, is 4 bytes, or 32 bits. That's why the compiler says each array component is 32 bits wide.

But for our example let's say that we really want to use only 72 total bits for the array type (and that we want three array components). That's the size we specified, after all. So how do we get the record type to be 24 bits instead of 32? Yes, you guessed it, we change the alignment for the record type. If we change it from 2 to 1, the size of 24 bits will work. Adding this Alignment clause line will do that:
An alignment of 1 means that any address will work, assuming that addresses refer to entire storage elements. (An alignment of 0 would mean that the address need not start on a storage element boundary, but we know of no such machines.)

We can even entirely replace the Size clause with the Alignment clause, because the Size clause specifying 24 bits is just confirming: it's the value that 'Size would return anyway. The problem is the object size.

Now, you may be wondering why an alignment of 1 would work, given that the alignment of the Temperature component is 2. Wouldn't it slow down the code, or even trap? Well, maybe. It depends on the machine. If it doesn't work we would just have to use 32 bits for the record type, with the original alignment of 2, for a larger total array size. Of course, if the compiler recognizes that a representation cannot be supported it must reject the code, but the compiler might not recognize the problem.

We said earlier that there are only a small number of reasons to specify 'Size for a type. We can mention one of them now. Setting 'Size can be useful to give the minimum number of bits to use for a component of a packed composite type, that is, within either a record type or an array type that is explicitly packed via the aspect or pragma Pack. It says that the compiler, when giving its best effort, shouldn't compress components of the type any smaller than the number of bits specified. No, it isn't earth-shattering, but other uses are more valuable, to be discussed soon.

One thing we will leave unaddressed (pun intended) is the question of bit ordering and byte ordering within our record layouts. In other words, the "endian-ness". That's a subject beyond the scope of this course. Suffice it to say that GNAT provides a way to specify record layouts that are independent of the endian-ness of the machine, within some implementation-oriented limits. That's obviously useful when the code might be compiled for a different ISA in the future. On the other hand, if your code is specifically for a single ISA, e.g., Arm, even if different boards and hardware vendors are involved, there's no need to be independent of the endian-ness. It will always be the same in that case. (Those are "famous last words" though.) For an overview of the GNAT facility, an attribute named attribute Scalar_Storage_Order see https://www.adacore.com/papers/lady-ada-mediates-peace-treaty-in-endianness-war.

Although specifying record type layouts and alignments are perhaps the most common representation characteristics expressed, there are a couple of other useful cases. Both involve storage allocation.

One useful scenario concerns tasking. We can specify the number of storage elements reserved for the execution of a task object, or all objects of a task type. You use the Storage_Size aspect to do so:

```ada
for Info'Alignment use 1;
```

```ada
task Servo with
    Storage_Size => 1 * 1024,
...
```

Or the corresponding pragma:

```ada
task Servo is
    pragma Storage_Size (1 * 1024);
end Servo;
```

The aspect seems textually cleaner and lighter unless you have task entries to declare as well. In that case the line for the pragma wouldn't add all that much. That's a matter of personal aesthetics anyway.

The specified number of storage elements includes the size of the task's stack (GNAT does have one, per task). The language does not specify whether or not it includes other storage associated with the task used for implementing and managing the task execution. With
GNAT, the extent of the primary stack size is the value returned, ignoring any other storage used internally in the run-time library for managing the task.

The GNAT run-time library allocates a default stack amount to each task, with different defaults depending on the underlying O.S., or lack thereof, and the target. You need to read the documentation to find the actual amount, or, with GNAT, read the code.

You would need to specify this amount in order to either increase or decrease the allocated storage. If the task won't run properly, perhaps crashing at strange and seemingly random places, there's a decent chance it is running out of stack space. That might also be the reason if you have a really deep series of subprogram calls that fails. The correction is to increase the allocation, as shown above. How much? Depends on the application code. The quick-and-dirty approach is to iteratively increase the allocation until the task runs properly. Then, reverse the approach until it starts to fail again. Add a little back until it runs, and leave it there. We'll mention a much better approach momentarily (GNATstack).

Even if the task doesn't seem to run out of task stack, you might want to reduce it anyway, to the extent possible, because the total amount of storage on your target might be limited. Some of the GNAT bare-metal embedded targets have very small amounts of memory available, so much so that the default task stack allocations would exhaust the memory available quickly. That's what the example above does: empirical data showed that the Servo task could run with just 1K bytes allocated, so we reduced it from the default accordingly. (We specified the size with that expression for the sake of readability, relative to using literals directly.)

Notice we said "empirical data" above. How do we know that we exercised the task's thread of control exhaustively, such that the arrived-at allocation value covers the worst case? We don't, not with certainty. If we really must know the allocation will suffice for all cases, say because this is a high-integrity application, we would use GNATstack. GNATstack is an offline tool that exploits data generated by the compiler to compute worst-case stack requirements per subprogram and per task. As a static analysis tool, its computation is based on information known at compile time. It does not rely on empirical run-time information.

The other useful scenario for allocating storage concerns access types, specifically access types whose values designate objects, as opposed to designating subprograms. (Remember, objects are either variables or constants.) There is no notion of dynamically allocating procedures and functions in Ada so access-to-subprogram types are not relevant here. But objects can be of protected types (or task types), and protected objects can "contain" entries and protected subprograms, so there's a lot of expressive power available. You just don't dynamically allocate procedures or functions as such.

First, a little background on access types, to supplement what we said earlier.

By default, the implementation chooses a standard storage pool for each named access-to-object type. The storage allocated by an allocator (i.e., new) for such a type comes from the associated pool.

Several access types can share the same pool. By default, the implementation might choose to have a single global storage pool, used by all such access types. This global pool might consist merely of calls to operating system routines (e.g., malloc), or it might be a vendor-defined pool instead. Alternatively, the implementation might choose to create a new pool for each access-to-object type, reclaiming the pool's memory when the access type goes out of scope (if ever). Other schemes are possible.

Finally, users may define new pool types, and may override the choice of pool for an access-to-object type by specifying Storage_Pool for the type. In this case, allocation (via new) takes memory from the user-defined pool and deallocation puts it back into that pool, transparently.

With that said, here's how to specify the storage to be used for an access-to-object type. There are two ways to do it.

If you specify Storage_Pool for an access type, you indicate a specific pool object to be used (user-defined or vendor-defined). The pool object determines how much storage is
available for allocation via new for that access type. Alternatively, you can specify Storage_Size for the access type. In this case, an implementation-defined pool is used for the access type, and the storage available is at least the amount requested, maybe more (it might round up to some advantageous block size, for example). If the implementation cannot satisfy the request, Storage_Error is raised.

It should be clear that that the two alternatives are mutually exclusive. Therefore the compiler will not allow you to specify both.

Each alternative has advantages. If your only concern is the total number of allocations possible, use Storage_Size and let the implementation do the rest. However, maybe you also care about the behavior of the allocation and deallocation routines themselves, beyond just providing and reclaiming the storage. In that case, use Storage_Pool and specify a pool object of the appropriate type. For example, you (or the vendor, or someone else) might create a pool type in which the allocation routine performs in constant time, because you want to do new in a real-time application where predictability is essential.

Lastly, an idiom: when using Storage_Size you may want to specify a value of zero. That means you intend to do no allocations whatsoever, and want the compiler to reject the code if you try. Why would you want an access type that doesn't allow dynamically allocating objects? It isn't as unreasonable as it might sound. If you plan to use the access type strictly with aliased objects, never doing any allocations, you can have the compiler enforce your intent. There are application domains that prohibit dynamic allocations due to the difficulties in analyzing their behavior, including issues of fragmentation and exhaustion. Access types themselves are allowed in these domains. You'd simply use them to designate aliased objects alone. In addition, in this usage scenario, if the implementation associates an actual pool with each access type, the pool's storage would be wasted since you never intend to allocate any storage from it. Specifying a size of 0 tells the implementation not to waste that storage.

Before we end this section, there is a GNAT compiler switch you should know about. The -gnatR? switch instructs the compiler to list the representation details for the types, objects and subprograms in the compiled file(s). Both implementation-defined and user-defined representation details are presented. The '?' is just a placeholder and can be one of the following characters:

[0|1|2|3|4][e][j][m][s]

Increasing numeric values provide increasing amounts of information. The default is '1' and usually will suffice. See the GNAT User's Guide for Native Platforms for the details of the switch in section 4.3.15 Debugging Control. You'll have to scroll down some to find that specific switch but it is worth finding and remembering. When you cannot understand what the compiler is telling you about the representation of something, this switch is your best friend.

https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with__gnat.html#debugging-control
35.6 Unchecked Programming

Ada is designed to be a reliable language by default, based as it is on static strong typing and high-level semantics. Many of the pitfalls that a developer must keep in the back of their mind with other languages do not apply in Ada, and are typically impossible. That protection extends to low-level programming as well, e.g., the Separation Principle. Nevertheless, low-level programming occasionally does require mechanisms that allow us to go beyond the safety net provided by the type rules and high-level language constructs.

One such mechanism (unchecked conversion) provides a way to circumvent the type system, a system otherwise firmly enforced by the compiler on our behalf. Note that by "circumventing the type system" we do not include so-called "checked" conversions. These conversions have meaningful semantics, and are, therefore, allowed by the language using a specific syntax. This conversion syntax is known as "functional" syntax because it looks like a function call, except that the "function" name is a type name, and the parameter is the object or value being converted to that type. These conversions are said to be "checked" because only specific kinds of types are allowed, and the compiler checks that such conversions are indeed between these allowed types.

Instead, this section discusses "unchecked" programming, so-called because the compiler does not check for meaningful semantics. There are multiple mechanisms for unchecked programming in Ada: in addition to circumventing the type system, we can also deallocate a previously-allocated object, and can create an access value without the usual checks. In all cases the responsibility for correct meaning and behavior rests on the developer. Very few, if any, checks are done by the compiler. If we convert a value to another type that generally makes no sense, for example a task object converted to a record type, we are on our own. If we deallocate an allocated object more than once, it is our fault and Bad Things inevitably result.

Likened to "escape hatches," the facilities for unchecked programming are explicit in Ada. Their use is very clear in the source code, and is relatively heavy: each mechanism is provided by the language in the form of a generic library subprogram that must be specified in a context clause ("with-clause") at the top of the file, and then instantiated prior to use, like any generic. For an introduction to generic units in Ada, see that section in the introductory Ada course: Introduction to Ada (page 123)

You should understand that the explicitly unchecked facilities in Ada are no more unsafe than the implicitly unchecked facilities in other languages. There's no safety-oriented reason to "drop down" to C, for example, to do low-level programming. For that matter, the low-level programming facilities in Ada are at least as powerful as those in other languages, and probably more so.

We will explore unchecked storage deallocation in a separate book so let's focus on unchecked type conversions.

Unchecked type conversions are achieved by instantiating this language-defined generic library function, a "child" of the root package named "Ada":

```ada
generic
  type Source(<>) is limited private;
  type Target(<>) is limited private;
function Ada.Unchecked_Conversion (S : Source) return Target
  with Pure, Nonblocking, Convention => Intrinsic;
```

The function, once instantiated and eventually invoked, returns the caller's value passed to S (of type Source) as if it is a value of type Target. That value can then be used in any way consistent with the Target type.

The two generic parameters, Source and Target, are defined in a manner that makes them very permissive in terms of the types they will accept when instantiated. To understand how, you need to understand a little bit of Ada's terminology and design for generic
unit parameters. (If you are already familiar with generic formal types and how they are matched, feel free to skip this material.)

First, the terminology. The type parameters defined by a generic unit are known as "generic formal types," or "generic formals" for short. Types Source and Target are the generic formals in the unit above. When instantiating such a generic, clients must specify a type for each generic formal type. The types specified by the client are known as "generic actual types," or "generic actuals" for short. You can remember that by the fact that the actuals are the types "actually" given to the generic unit to work with when instantiated. (You may laugh, but that mnemonic works.)

Now we're ready to discuss the language design concept. The idea is that the syntax of a generic formal type indicates what kind of generic actual is required for a legal instantiation. This is known as the "Contract Model" because we can think of the formal parameters as expressing a contract between the generic unit's implementation and the client code that instantiates the generic. The contract is enforced by the compiler, in that it will reject any instantiation that attempts to specify some actual type that does not match the formal's requirements.

For example, if the generic computes some value for any floating point type, that floating-point type would be declared as a generic formal type, and would be defined so that only some floating-point type could be used for the corresponding actual type:

```ada
generic
  type Real is digits <>;
```

The formal parameter syntax reflects the syntax of a floating-point type declaration, except that the <> (the "box") indicates that the generic does not care how many digits are available. The generic actual will be some floating point type and it will specify the number of decimal digits.

If instead we try to match that formal with some actual that is anything other than a floating-point type the compiler will reject the instantiation. Therefore, within the generic body, the implementation code can be written with the assurance that the characteristics and capabilities required of a floating point type will be available. That's the Contract Model in full: the requirements are a matter of the generic unit's purpose and implementation, so the formal parameters reflect those requirements and the compiler ensures they will be met.

Some generic units, though, do not require specifically numeric actual types. These generics can use less specific syntax for their formal types, and as a result, more kinds of actual types are permitted in the instantiations. Remember the Contract Model and this will make sense. The contract between the generic and the clients is, in this case, more permissive: it does not require a numeric type in order to implement whatever it does.

For illustration, suppose we want a generic procedure that will exchange two values of some type. What operations does the generic unit require in the implementation in order to swap two values? There are two: assignment, as you might expect, but also the ability to declare objects of the type (the "temporary" used to hold one of the values during the swap steps). As long as the body can do that, any type will suffice, so the generic formals are written to be that permissive. What is the syntax that expresses that permissiveness, you ask? To answer that, first consider simple, non-generic private types from the user's point of view. For example:

```ada
package P is
  type Foo is private;
  procedure Do_Something (This : Foo);
private
  type Foo is ... -- whatever
end P;
```

There are two "views" associated with the package: one for the "visible" part of the package...
Learning Ada

spec (declaration), known as the "partial" view, and one for the "private" part of the package spec and the package body, known as the "full" view. The differences between the two views are a function of compile-time visibility.

The partial view is what clients (i.e., users) of the package have: the ability to do things that a type name provides, such as declarations of objects, as well as some basic operations such as assignment, some functions for equality and inequality, some conversions, and whatever subprograms work on the type (the procedure Do_Something above). Practically speaking, that's about all that the partial view provides. That's quite a lot, in fact, and corresponds to the classic definition of an "abstract data type."

The code within the package private part and package body has the full view. This code has compile-time visibility to the full definition for type Foo, so there are additional capabilities available to this code. For example, if the full definition for Foo is as an array type, indexing will be available with the private part and body. If Foo is fully defined as some numeric type, arithmetic operations will be possible within the package, and so on.

Therefore, the full view provides capabilities for type Foo that users of the type cannot access via the partial view. Only the implementation for type Foo and procedure Do_Something have the potential to access them.

Now, back to the generic formal parameter. If the generic unit doesn't care what the actual type is, and just needs to be able do assignment and object declaration, a "generic formal private type" expresses exactly that:

```ada
generic
  type Item is private;
procedure Exchange( Left, Right : in out Item );

procedure Exchange( Left, Right : in out Item ) is
  Old_Left : Item;
begin
  Old_Left := Left;
  Left := Right;
  Right := Old_Left;
end Exchange;
```

Inside generic procedure Exchange, the view of type Item is as if Item were some private type declared in a package, with only the partial view available. But the operations provided by a partial view are sufficient to implement the body of Exchange: only assignment and object declaration are required. Any additional capabilities that the generic actual type may have — array indexing, arithmetic operators, whatever — are immaterial because they are not required. That's the Contract Model: only the specified view's required capabilities are important. Anything else the type can also do is not relevant.

But consider limited types. Those types don't allow assignment, by definition. Therefore, an instantiation that specified a limited actual type for the generic formal type Item above would be rejected by the compiler. The contract specifies the ability to do assignment so a limited type would violate the contract.

Finally, as mentioned, our Exchange generic needs to declare the "temporary" object Old_Left. A partial view of a private type allows that. But not all types are sufficient, by their name alone, to declare objects. Unconstrained array types, such as type String, are a familiar example: they require the bounds to be specified when declaring objects; the name String alone is insufficient. Therefore, such types would also violate the contract and, therefore, would be rejected by the compiler when attempting to instantiate generic procedure Exchange.

Suppose, however, that we have some other generic unit whose implementation does not need to declare objects of the formal type. In that case, a generic actual type that did not support object declaration (by the name alone) would be acceptable for an instantiation. The generic formal syntax for expressing that contract uses these tokens: (<> ) in addition to the other syntax mentioned earlier.
In the above, the generic formal type Foo expresses the fact that it can allow unconstrained types — known as "indefinite types" — when instantiated because it will not attempt to use that type name to declare objects. Of course, the compiler will also allow constrained types (e.g., Integer, Boolean, etc.) in instantiations because it doesn't matter one way or the other inside the generic implementation. The Contract Model says that additional capabilities, declaring objects in this case, are allowed but not required. (There is a way to declare objects of indefinite types, but not using the type name alone. The unchecked facilities don't need to declare objects so we will not show how to do it.)

Now that you understand the Contract Model (perhaps more than you cared), we are ready to examine the generic formal type parameters for Ada.Unchecked_Conversion. Here's the declaration again:

```
generic
  type Foo<> is private;
```

The two generic formal types, Source, and Target, are the types used for the incoming value and the returned value, respectively. Both formals are "indefinite, limited private types" in the jargon, but now you know what that means. Inside the implementation of the generic function, neither Source nor Target will be used to declare objects (the <> syntax). Likewise, neither type will be used in an assignment statement (the "limited" reserved word). And finally, no particular kind of type is required for Source or Target (the private reserved word). That's a fairly restricted usage within the generic implementation, but as a result the contract can be very permissive: the generic can be instantiated with almost any type. It doesn't matter if the actual is limited or not, private or not, and indefinite or not. The generic implementation doesn't need those capabilities to implement a conversion so they are not part of the contract expressed by the generic formal types.

What sort of type would be disallowed? Abstract types, and incomplete types. However, it is impossible to declare objects of those types, for good reasons, so unchecked conversion is never needed for them.

Note that the result value is returned by-reference whenever possible, in which case it is just a view of the Source bits in the formal parameter S and not a copy. For a Source type that is not a by-copy type, the result of an unchecked conversion will typically be returned by-reference (so that the result and the parameter S share the same storage); for a by-copy Source type, a copy is made.

The compiler can restrict instantiations but implementers are advised by the language standard to avoid them unless they are required by the target environment. For example, an instantiation for types for which unchecked conversion can't possibly make sense might be disallowed.

Clients can apply language- and vendor-defined restrictions as well, via pragma Restrictions. In particular, the language defines the No_Dependence restriction, meaning that no client's context clause can specify the unit specified. As a result no client can instantiate the generic for unchecked conversion:

```
pragma Restrictions (No_Dependence => Ada.Unchecked_Conversion);
```

hence there would be no use of unchecked conversion.

From the Contract Model's point of view most any type can be converted to some other type via this generic function. But practically speaking, some limitations are necessary. The following must all be true for the conversion effect to be defined by the language:
Learning Ada

- $S'\text{Size} = \text{Target}'\text{Size}$
- $S'\text{Alignment}$ is a multiple of $\text{Target}'\text{Alignment}$, or $\text{Target}'\text{Alignment}$ is 0 (meaning no alignment required whatsoever)
- Target is not an unconstrained composite type
- $S$ and Target both have a contiguous representation
- The representation of $S$ is a representation of an object of the target subtype

We will examine these requirements in turn, but realize that they are not a matter of legality. Compilers can allow instantiations that violate these requirements. Rather, they are requirements for conversions to have the defined effect.

The first requirement is that the size (in bits) for the parameter $S$, of type Source, is the same as the size of the Target type. That's reasonable if you consider it. What would it mean to convert, for example, a 32-bit value to an 8-bit value? Which 8 bits should be used?

As a result, one of the few reasons for setting the size of a type (as opposed to the size of an object) is for the sake of well-defined unchecked conversions. We might make the size larger than it would need to be because we want to convert a value of that type to what would otherwise be a larger Target type.

Because converting between types that are not the same size is so open to interpretation, most compilers will issue a warning when the sizes are not the same. Some will even reject the instantiation. GNAT will issue a warning for these cases when the warnings are enabled, but will allow the instantiation. We're supposed to know what we are doing, after all. The warning is enabled via the specific `-gnatwz` switch or the more general `-gnatwa` switch. GNAT tries to be permissive. For example, in the case of discrete types, a shorter source is first zero or sign extended as necessary, and a shorter target is simply truncated on the left. See the GNAT RM for the other details.

The next requirement concerns alignment. As we mentioned earlier, modern architectures tend to have strict alignment requirements. We can meaningfully convert to a type with a stricter alignment, or to a type with no alignment requirement, but converting in the other direction would require a copy.

Next, recall that objects of unconstrained types, such as unconstrained array types or discriminated record types, must have their constraints specified when the objects are declared. We cannot just declare a String object, for example, we must also specify the lower and upper bounds. Those bounds are stored in memory, logically as part of the String object, since each object could have different bounds (that's the point, after all). What, then, would it mean to convert some value of a type that has no bounds to a type that requires bounds? The third requirement says that it is not meaningful to do so.

The next requirement is that the argument for $S$, and the conversion target type Target, have a contiguous representation in memory. In other words, each storage unit must be immediately adjacent, physically, to the next logical storage unit in the value. Such a representation for any given type is not required by the language, although on typical modern architectures it is common. (The type `System.Storage_Elements.Storage_Array` is an exception, in that a contiguous representation is guaranteed.) An instance of Ada.Unchecked_Conversion just takes the bits of $S$ and treats them as if they are bits for a value of type Target (more or less), and does not handle issues of segmentation.

The last requirement merely states that the bits of the argument $S$, when treated as a value of type Target, must actually be a bit-pattern representing a value of type Target (strictly, the subtype). For example, with signed integers, any bit pattern (of the right size) represents a valid value for those types. In contrast, consider an enumeration type. By default, the underlying representational values are the same as the position values, i.e., starting at zero and increasing by one. But users can override that representation: they can start with any value and, although the values must increase, they need not increase by one:
If we covert an unsigned integer (of the right size) to a Toggle_Switch value, what would it mean if the Source value was neither 0 nor 4?

We’ve said that the instantiations are likely allowed, hence callable functions are created. If the above requirements are not met, what happens?

What happens depends on the Target type, that is, the result type for the conversion. Specifically, it depends on whether the target type is a "scalar" type. As we mentioned earlier, a scalar type is either a "discrete" type or a "real" type, which are themselves further defined, as the figure below indicates. Any other type is a non-scalar type, e.g., record types, access types, task types, and so on.

When the requirements for meaningful instantiations are not respected and the Target type is a scalar type, the result returned from the call is implementation defined and is potentially an invalid representation. For example, type Toggle_Switch is an enumeration type, hence it is a scalar type. Therefore, if we covert an unsigned integer (of the right size) to a Toggle_Switch value, and the Source value is neither 0 nor 4, the resulting value is an invalid representation. That's the same as an object of type Toggle_Switch that is never assigned a value. The random junk in the bits may or may not be a valid Toggle_Switch value. That's not a good situation, clearly, but it is well-defined: if it is detected, either Constraint_Error or Program_Error is raised. If the situation is not detected, execution continues using the invalid representation. In that case it may or may not be detected, near the call or later. For example:

```ada
with Ada.Unchecked_Conversion;
with Ada.Text_IO;  use Ada.Text_IO;
with Interfaces;   use Interfaces;

procedure Demo is
  type Toggle_Switch is (Off, On) with Size => 8;
  for Toggle_Switch use (Off => 0, On => 4);

  function As_Toggle_Switch is new Ada.Unchecked_Conversion
    (Source => Unsigned_8, Target => Toggle_Switch);

35.6. Unchecked Programming 1147
```
T1 : Toggle_Switch;
T2 : Toggle_Switch;
begin
  T1 := As_Toggle_Switch (12);  -- neither 1 nor 4
  if T1 = Off then
    Put_Line ("T1's off");
  else
    Put_Line ("T1's on");
  end if;
  T2 := T1;
  if T2 = Off then
    Put_Line ("T2's off");
  else
    Put_Line ("T2's on");
  end if;
  Put_Line (T2'Image);
end Demo;

In the execution of the code above, the invalid representation value in T1 is not detected, except that it is copied into T2, where it is eventually detected when 'Image is applied to T2. The invalid representation is not detected in the assignment statement or the comparison because we want the optimizer to be able to avoid emitting a check prior to every use of the value. Otherwise the generated code would be too slow. (The language explicitly allows this optimization.)

The evaluation of an object having an invalid representation value due to unchecked conversion is a so-called "bounded error" because the results at run-time are predictable and limited to one of those three possibilities: the two possible exceptions, or continued execution.

Continued execution might even work as hoped, but such code is not portable and should be avoided. A new vendor's compiler, or even a new version of a given vendor's compiler, might detect the situation and raise an exception. That happens, and it ends up costing developer time to make the required application code changes.

The possibilities get much worse when the result type is not a scalar type. In this case, the effect of the call — not the value returned by the call — is implementation defined. As a result, the possible run-time behavior is unpredictable and, consequently, from the language rules point of view anything is possible. Such execution is said to be "erroneous."

Why the difference based on scalar versus non-scalar types? Scalar types have a simple representation: their bits directly represent their values. Non-scalar types don't always have a simple representation that can be verified by examining their bits.

For example, we can have record types with discriminants that control the size of the corresponding objects because the record type contains an array component that uses the discriminant to set the upper bound. These record types might have multiple discriminants, and multiple dependent components. As a result, an implementation could have hidden, internal record components. These internal components might be used to store the starting address of the dependent components, for example, or might use pointers to provide a level of indirection. If an unchecked conversion did not provide correct values for these internal components, the effect of referencing the record object would be unpredictable.

Even a comparatively simple record type with one such dependent component is sufficient to illustrate the problem. There are no internal, hidden components involved:

```ada
with Ada.Unchecked_Conversion;
with Ada.Text_IO; use Ada.Text_IO;
with System; use System; -- for Storage_Unit
with System.Storage_Elements; use System.Storage_Elements;
(continues on next page)```
procedure Demo_Erroneous is

subtype Buffer_Size is Storage_Offset range 1 .. Storage_Offset'Last;

type Bounded_Buffer (Capacity : Buffer_Size) is record
  Content : Storage_Array (1 .. Capacity);
  Length : Storage_Offset := 0;
end record;

procedure Show_Capacity (This : Bounded_Buffer);

subtype OneK_Bounded_Buffer is Bounded_Buffer (Capacity => 1 * 1024);

function As_OneK_Bounded_Buffer is new Ada.Unchecked_Conversion
  (Source => Storage_Array, Target => OneK_Bounded_Buffer);

Buffer : OneK_Bounded_Buffer;
Sequence : Storage_Array (1 .. Buffer'Size / Storage_Unit);

procedure Show_Capacity (This : Bounded_Buffer) is
begin
  Put_line ("This.Capacity is" & This.Capacity'Image);
end Show_Capacity;

begin
  Buffer := As_OneK_Bounded_Buffer (Sequence);
  Put_line ("Buffer capacity is" & Buffer.Capacity'Image);
  Show_Capacity (Buffer);
  Put_line ("Done");
end Demo_Erroneous;

In the above, the type Bounded_Buffer has an array component Content that depends on the discriminant Capacity for the number of array components. This is an extremely common idiom. However, unchecked conversion is only meaningful, as defined earlier, when converting to constrained target types. Bounded_Buffer is not constrained, so we define a constrained subtype (OneK_Bounded_Buffer) for the sake of the conversion.

The specific Buffer object is 8320 bits (1024 * 8, plus 2 * 64), as is the Sequence object, so the sizes are the same.

The alignment of OneK_Bounded_Buffer is 8, and Storage_Array's alignment is 1, so the Target type is a multiple of the Source type, as required.

Both types have a contiguous representation, and the sequence of bytes can be a valid representation for the record type, although it certainly might not be valid. For example, if we change the discriminant from what the subtype specifies, we would have an invalid representation for that subtype.

So we can reasonably invoke an unchecked conversion between the array of bytes and the record type. However, as you can see in the code and as the compiler warns, we never assigned a value to the Sequence array object. The unchecked conversion from that Sequence of bytes includes the discriminant value, so it is very possible that we will get a discriminant value that is not 1K.

We can test that possibility by running the program. In the first call to Put_Line, the program prints the Capacity discriminant for the Buffer object. The compiler knew it was 1024, so it doesn't get the discriminant component from memory, it just directly prints 1024. However, we can force the compiler to query the discriminant in memory. We can pass Buffer to procedure Show_Capacity, which takes any Bounded_Buffer, and there query (print) the Capacity component under that different view. That works because the view inside the procedure Show_Capacity is as of Bounded_Buffer, in which the discriminant value is unknown at compile-time.
In the above examples, we are responsible for ensuring that the enumeration representation encoding and the record discriminant value are correct when converted from some other type. That's not too hard to recognize because we can literally see in the source code that there is something to be maintained by the conversions. However, there might be hidden implementation artifacts that we cannot see in the source code but that must be maintained nevertheless.

For example, the compiler’s implementation for some record type might use dynamic memory allocations instead of directly representing some components. That would not appear in the source code. As a simpler example of invisible implementation issues, consider again our earlier record type:

```ada
type My_Int is range 1..10;
subtype S is Integer range 1..10;

type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```

As we discussed earlier, between the bytes that are allocated to the record components are some other bytes that are not used at all. As usual, the compiler must implement the language-defined equality operator for the record type. One way to implement that function would be to generate code that checks the equality for each component individually, ignoring any unused bytes. But suppose you have a large record type with many components. The code for checking record level equality will be extensive and inefficient. An alternative implementation for the compiler would be to use a "block compare" machine instruction to check the equality of the entire record at once, rather than component-by-component. That will be considerably more efficient because the block-compare instruction just compares the bits from one starting address to another ending address. But in that case the "unused" bytes are not skipped so the values within those bytes become significant. Comparison of those unused bytes will only work if their values are defined and assigned in each record object. Compilers that may use a block-comparison approach will, therefore, always set those unused bytes to a known value (typically zero). That is part of the valid representation for values of the type, and consequently must be maintained by our unchecked conversions. This being a non-scalar target type, failure to do so results in erroneous execution, i.e., undefined behavior. "There be dragons" as ancient maps of the unknown world once said.

As you can see, you should use unchecked conversions with considerable care and thought. Moreover, because unchecked programming is such a low-level activity, and has vendor-defined implementation issues, it is not only less portable than high-level coding, it is also less portable than other low-level programming. You will be well served if you limit the use of unchecked conversions overall. If your application code is performing unchecked conversions all over the code, something is very likely wrong, or at least very questionable. A well-designed Ada program should not need ubiquitous unchecked conversions.

That said, of course sometimes unchecked conversions are reasonable. But even then, it is better to isolate and hide their use via compile-time visibility controls. For example, instead of having clients invoke unchecked conversion instances many times, have a procedure that is invoked many times, and let the procedure body do the conversion. That way, the clients see a high-level specification of functionality, and, if the conversion needs to be changed later, there is only that one conversion usage (the procedure body) to change. This approach is really just another example of isolating and hiding code that might need to change in the future.
### 35.7 Data Validity

Our earlier demo program assigned an incorrect value via unchecked conversion into an object of an enumeration type that had non-standard representation values. The value assigned was not one of those representation values so the object had an invalid representation. Certain uses of an invalid representation value will be erroneous, and we saw that the effect of erroneous execution was unpredictable and unbounded.

That example was somewhat artificial, for the sake of illustration. But we might get an invalid value in a real-world application. For example, we could get an invalid value from a sensor. Hardware sensors are frequently unreliable and noisy. We might get an invalid value from a call to an imported function implemented in some other language. Whenever an assignment is aborted, the target of the assignment might not be fully assigned, leading to so-called "abnormal" values. Other causes are also possible. The problem is not unusual in low-level programming.

How do we avoid the resulting bounded errors and erroneous execution?

In addition to assignment statements, we can safely apply the Valid attribute to the object. This language-defined attribute returns a Boolean value indicating whether or not the object's value is a valid representation for the object's subtype. (More details in a moment.) There is no portable alternative to check an object's validity. Here's an example:

```ada
with Ada.Unchecked_Conversion;
with Ada.Text_IO;  use Ada.Text_IO;
with Interfaces;   use Interfaces;
with System;

procedure Demo_Viability_Check is

    type Toggle_Switch is (Off, On) with Size => 8;
    for Toggle_Switch use (Off => 1, On => 4);

    T1 : Toggle_Switch;

    function Sensor_Reading (Default : Toggle_Switch) return Toggle_Switch is

        function As_Toggle_Switch is new Ada.Unchecked_Conversion
            (Source => Unsigned_8, Target => Toggle_Switch);

        Result : Toggle_Switch;
        Sensor : Unsigned_8;
        -- for Sensor'Address use System'To_Address (...);

        begin
            Result := As_Toggle_Switch (Sensor);
            return (if Result'Valid then Result else Default);
        end Sensor_Reading;

    begin
        T1 := Sensor_Reading (Default => Off); -- arbitrary
        Put_Line (T1'Image);
    end Demo_Viability_Check;
```

In the above, Sensor_Reading is the high-level, functional API provided to clients. The function hides the use of the unchecked conversion, and also hides the memory-mapped hardware interface named Sensor. We've commented out the address clause since we don't really have a memory mapped device available. You can experiment with this program by changing the code to assign a value to Sensor (e.g., when it is declared). It is an unsigned 8-bit quantity so any value in the corresponding range would be allowed.

In addition to checking for a valid representation, thus preventing the bounded error, Valid
also checks that the object is not abnormal, so erroneous execution can be prevented too. (It also checks that any subtype predicate defined for the Target type is also satisfied, but that's a lesson for another day.)

However, the Valid attribute can be applied only to scalar objects. There is no language-defined attribute for checking objects of composite types. That's because it would be very hard to implement for some types, if not impossible. For example, given a typical run-time model, it is impossible to check the validity of an access value component. Therefore, you must individually check the validity of scalar record or array components.

At least, you would have to check them individually in standard Ada. GNAT defines another Boolean attribute, named Valid_Scalars, to check them all for us. This attribute returns True if the evaluation of Valid returns True for every scalar subcomponent of the enclosing composite type. It also returns True when there are no scalar subcomponents. See the GNAT RM for more information.
Software projects often involve more than one programming language. Typically that's because there is existing code that already does something we need done and, for that specific code, it doesn't make economic sense to redevelop it in some other language. Consider the rotor blade model in a high-fidelity helicopter simulation. Nobody touches the code for that model except for a few specialists, because the code is extraordinarily complex. (This complexity is unavoidable because a rotor blade's dynamic behavior is so complex. You can't even model it as one physical piece because the tip is traveling so much faster than the other end.) Complex and expensive models like that are a simulator company's crown jewels; their cost is meant to be amortized over as many projects as possible. Nobody would imagine redeveloping it simply because a new project is to be written in a different language.

Therefore, Ada includes extensive facilities to "import" foreign entities into Ada code, and to "export" Ada entities to code in foreign languages. The facilities are so useful that Ada has been used purely as "glue code" to allow code written in two other programming languages to be used together.

You've already seen an introduction to Ada and C code working together in the "Interfacing" section of the Ada introductory course (page 175). If you have not seen that material, be sure to see it first. We will cover some further details not already discussed there, and then go into the details of the facilities not covered elsewhere, but we assume you're familiar with it.

The Ada foreign language interfacing facilities include both "general" and "language-specific" capabilities. The "general" facilities are known as such because they are not tied to any specific language. These pragmas and aspects work with any of the supported foreign languages. In contrast, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. For example, as you saw in that "Interfacing" section, there is a package with a number of declarations for C types, such as `int`, `float`, and `double`, as well as C "strings", with subprograms to convert back and forth between them and Ada's string type. Other languages are also supported, both by the Ada Standard and by vendor additions. You will frequently use both the "general" and the "language-specific" facilities together.

All these interfacing capabilities are defined in Annex B of the language standard. Note that Annex B is not a "Specialized Needs" annex, unlike some of the other annexes. The Specialized Needs annexes are wholly optional, whereas all Ada implementations must implement Annex B. However, some parts of Annex B are optional, so more precisely we should say that every implementation must support all the required features of Annex B. That comes down mainly to the package Interfaces (more on that package in a moment). However, if an implementation does implement any optional part of Annex B, it must be implemented as described by the standard, or with less functionality. An implementation cannot use the same name for some facility (aspect, etc.) but with different semantics. That's true of the Specialized Needs annexes too: not every part need be implemented, but any part that is implemented must conform to the standard. In practice, for Annex B, all implementations provide the required parts, but not all provide support for all the "language-specific" foreign
languages' interfaces. The vendors make a business decision for the optional parts, just as they do regarding the Specialized Needs annexes.

## 36.1 General Interfacing

In the "Interfacing" section of the Ada introductory course you saw that Ada defines aspects and pragmas for working with foreign languages. These aspects and pragmas are functionally interchangeable, and we will use whichever one of the two that is most convenient in our discussion. The pragmas are officially "obsolescent," but that merely means that a newer approach is available, in this case the corresponding aspects. You can use either one without concern for future support because language constructs that are obsolescent are not removed from the language. Any compiler that supports such constructs will almost certainly support them forever, for the sake of not invalidating existing customers' code. The pragmas have been in the language since Ada 95 so there's a lot of existing code using them. Changing the compiler isn't cost-free, after all, so why spend the money to potentially lose a customer? Likewise, a brand new compiler will also probably support them, for the sake of potentially gaining a customer.

The general interfacing facility consists of these aspects and pragmas, specifically **Import**, **Export**, and **Convention**. As you saw in the Ada Introduction course, **Import** brings a foreign entity into Ada code, **Export** does the opposite, and **Convention** supplies additional information and directives to the compiler. We will go into the details of each.

Regardless of whether the Ada code is importing or exporting some entity, there will be an Ada declaration for that entity. That declaration tells the compiler how the entity can be used, as usual. The interfacing aspects and pragmas are then applied to these Ada declarations.

If we are exporting, then the entity is implemented in Ada. For a subprogram that means there will also be a subprogram body matching the declaration, and the compiler will enforce that requirement as usual. In contrast, if we are importing a subprogram, then it is not implemented in Ada, and therefore there will be no corresponding subprogram body for the Ada declaration. The compiler would not allow it if we tried. In that case the **Import** is the subprogram's completion.

Subprograms often have a separate declaration. Sometimes that's required, for example when we want to include a subprogram as part of a package's API, but at other times it is optional. Remember that a subprogram body acts as a corresponding declaration when there is no separate declaration defined. Thus, either way, we have a subprogram declaration available for the interfacing aspects and/or pragmas.

For data that are imported or exported, we'll have the declaration of the object in Ada to which we can apply the necessary interfacing aspects/pragmas. But we will also have the types for these objects, and as you will see, the types can be part of interfacing too.

### 36.1.1 Aspect/Pragma Convention

As you saw in the "Interfacing" section of the Ada introductory course (page 175), when importing and exporting you'll also specify the "convention" for the entity in question. The pragmas for importing and exporting include a parameter for this purpose. When using the aspects, you'll specify the Convention aspect too.

For types, though, you will specify the Convention aspect/pragma alone, without **Import** or **Export**. In this case the convention specifies the layout for objects of that type, presumably a layout different than the Ada compiler would normally use. You would need to specify this other layout either because you're going to later declare and export an object of the type, or because you are going to declare an object of the type and pass it as a argument to an imported subprogram.
For example, Ada specifies that multi-dimensional arrays are represented in memory in row-major order. In contrast, the Fortran standard specifies column-major order. If we want to define a type in Ada that can be used for passing parameters to Fortran routines, we need to specify that convention for the type. For example:

```ada
type Matrix is array (Rows, Columns) of Float
  with Convention => Fortran;
```

(Rows and Columns are user-defined discrete subtypes.)

As a result when we declare Matrix objects the Ada compiler will use the column-major layout. That makes it possible to pass objects of the type to imported Fortran subprograms because the formal parameter will also be of type Matrix. The imported Fortran routine will then see the parameter in memory as it expects to see it. So although you wouldn't need to import or export a type itself, you might very well import or export an object of the type, or pass it as a argument.

When Convention is applied to subprograms, a natural mistake is to think that we are specifying the programming language used to implement the subprogram. In reality, the convention indicates the subprogram calling convention, not the implementation language. The calling convention specifies how parameters are passed to and from subprogram calls, how result values for functions are returned, the order that parameters are pushed on the call stack, how dynamically-sized parameters are passed, and so on. Ordinarily these are matters you don't need to consider because you're working within a single convention automatically, in other words the one used by the Ada compiler you're using.

To illustrate that the convention is not the implementation language, consider a subprogram that we intend to import and call from Ada. This imported routine is implemented in assembly language, but, in addition, let's say it is written to use the same calling convention as the Ada compiler we are using for Ada code. Therefore, the calling convention would be Ada even though the implementation is in assembler.

```ada
procedure P (X : Integer) with
  Convention => Ada,
  ...
```

In the example above, Ada is known as a convention identifier, as is Fortran in the earlier example. Convention identifiers are defined by the Ada language standard, but also by Ada vendors.

The Ada standard defines two convention identifiers: Ada (the default), and Intrinsic. In addition, Annex B defines convention identifiers C, COBOL, and Fortran. Support for these Annex B conventions is optional.

GNAT supports the standard and Annex B conventions, as well as the following: Assembler, "C_PLUS_PLUS" (or CPP), Stdcall, WIN32, and a few others. C_PLUS_PLUS is the convention identifier required by the standard when C++ is supported. (Convention identifiers are actual identifiers, not strings, so they must obey the syntax rules for identifiers. "C++" would not be a valid identifier.) See the GNAT User Guide for those other GNAT-specific conventions.

Stdcall and WIN32 actually do specify a particular calling convention, but for those convention identifiers that are language names, how do we get from the name to a calling convention?

The ultimate requirement for any calling convention is compatibility with the Ada compiler we are using. Specifically, the Ada compiler must recognize what the calling convention specifies, and support importing and exporting subprograms with that convention applied.

For the Ada convention that's simple. There is no standard calling convention for Ada. Convention Ada simply means the calling convention applied by the Ada compiler we happen to be using. (We'll talk about Intrinsic shortly.)
So far, so good. But how to we get from those other language names to corresponding calling conventions? There is no standard calling convention for, say, C, any more than there is a standard calling convention for Ada.

In fact we don't get to the calling convention, at least not directly. What the language name in the convention identifier actually tells us is that, when that convention is supported, there is a compiler for that foreign language that uses a calling convention known to, and supported by, the Ada compiler we are using. The Ada compiler vendor defines which languages it supports, after all. For example, when supported, convention C means that there is a compatible C compiler known to the Ada compiler vendor. For GNAT you can guess which C compiler that might be.

It's actually pretty straightforward once you have the big picture. If the convention is supported, the Ada compiler in use knows of a compiler for that language with which it can work. Annex B just defines some convention identifiers for the sake of portability.

But suppose a given Ada compiler supports more than one vendor for a given programming language? In that case the Ada compiler would define and support multiple convention identifiers for the same programming language. Presumably these identifiers would be differentiated by the compiler vendors' names. Thus we might have available conventions GNU_Fortran and Intel_Fortran if both were supported. The Fortran convention identifier would then indicate the default vendor's compiler.

The Intrinsic calling convention represents subprograms that are "built in" to the compiler. When such a subprogram is called the compiler doesn't actually generate the code for an out-of-line call. Instead, the compiler emits the assembly code — often just a single instruction — corresponding to the intrinsic subprogram's name. There will be a separate declaration for the subprogram, but no actual subprogram body containing a sequence of statements. The compiler just knows what to emit in place of the call.

For example:

```ada
function Shift_Left
  (Value   : Unsigned 16;
   Amount : Natural)
return Unsigned_16
with ..., Convention => Intrinsic;
```

The effect is much like a subprogram call that is always in-lined, except that there's no body for the subprogram. In this example the compiler simply issues a shift-left instruction in assembly language.

You'll see the Intrinsic convention applied to many language-defined subprograms. For example:

```ada
generic
  type Source(<>), Target(<>) is limited private;
function Ada.Unchecked_Conversion(Source : Source) return Target
with ..., Convention => Intrinsic;
```

Thus when we call an instantiation of Ada.Unchecked_Conversion there is no actual call made to some subprogram. The compiler just treats the bits of S as a value of type Target.

Intrinsic subprograms are a good way to access interesting capabilities of the target hardware, without having to write the assembly language yourself (although we will show how to do that, later, directly in Ada). For example, some targets provide an instruction that atomically compares and swaps a value in memory. Ada 2022 just added a standard package for this, but before that we could use the following to access a gcc built-in:

```ada
-- Perform an atomic compare and swap: if the current value of
-- Destination.all is Comparand, then write New_Value into Destination.all.
```

(continues on next page)
function Sync_Val_Compare_And_Swap_Bool_8  
(Destination : access Unsigned_8;  
Comparand   : Unsigned_8;  
New Value   : Unsigned_8)  
return Boolean  
with Convention => Intrinsic,
...

We would specify additional aspects beyond that of Convention but these have not yet been discussed. That's what the ellipses indicate in the various examples above.

36.1.2 Aspect/Pragma Import and Export

You've already seen these aspects in the Ada Introduction course, but for completeness: Import brings a foreign entity into Ada code, and Export makes an Ada entity available to foreign code. In practice, these entities consist of objects and subprograms, but the language doesn't impose many restrictions. It is up to the vendor to decide what makes sense for their specific target.

The aspects Import and Export are so-called Boolean aspects because their value is either True or False. For example:

Obj : Matrix with  
   Export => True,  
   ...

For any Boolean-valued aspect the default is True so you only need to give the value explicitly if that value is False. There would be no point in doing that in these two cases, of course. Hence we just give the aspect name:

Obj : Matrix with  
   Export,  
   ...

Recall that objects of some types are initialized automatically during the objects' elaboration, unless they are explicitly initialized as part of their declarations. Access types are like that, for example. Objects of these types are default initialized to null as part of ensuring that their values are always meaningful (absent unchecked conversion).

type Reference is access Integer;

Obj : Reference;

In the above the value of Obj is null, just as if we had explicitly set it that way.

But that initialization is a problem if we are importing an object of an access type. Presumably the value is set by the foreign code, so automatic initialization to null would overwrite the incoming value. Therefore, the language guarantees that implicit initialization won't be applied to imported objects.

type Reference is access Integer;

Obj : Reference with Import;

Now the value of Obj is whatever the foreign code sets it to, and is not, in other words, overwritten during elaboration of the declaration.
36.1.3 Aspect/Pragma External_Name and Link_Name

For an entity with a `True` Import or Export aspect, we can also specify a so-called external name or link name. These names are specified via aspects `External_Name` and `Link_Name` respectively.

An external name is a string value indicating the name for some entity as known by foreign language code. For an entity that Ada code imports, this is the name that the foreign code declares it to be. For an entity that Ada code exports, this is the name that the foreign code is told to use. This string value is exactly the name to be used, so if you misspell the name the link will fail. For example:

```ada
function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand  : Unsigned_8;
   New Value  : Unsigned_8)
return Boolean with
Import, Convention => Intrinsic,
External_Name => "__sync_bool_compare_and_swap_1"
```

The `External_Name` and `Link_Name` values are strings because the foreign unit names don't necessarily follow the Ada rules for identifiers (the leading underscores in this case). Note that the ending digit in the name above is different from the declared Ada name.

Usually, the name of the imported or exported entity is precisely known and hence exactly specified by `External_Name`. Sometimes, however, a compilation system may have a linker "preprocessor" that augments the name actually used by the linkage step. For example, an implementation might always prepend "." and then pass the result to the system linker. In that case we don't want to specify the exact name. Instead, we want to provide the "starting point" for the name modification. That's the purpose of the aspect `Link_Name`.

If you don't specify either `External_Name` or `Link_Name` the compilation system will choose one in some implementation-defined manner. Typically this would be the entity's defining name in the Ada declaration, or some simple transformation thereof. But usually we know the name exactly and so we use `External_Name` to give it.

As you can see, it really wouldn't make sense to specify both `External_Name` and `Link_Name` since the semantics of the two conflict. But if both are specified for some reason, the `External_Name` value is ignored.

Note that `Link_Name` cannot be specified for Intrinsic subprograms because there is no actual unit being linked into the executable, because intrinsics are built-in. In this case you must specify the `External_Name`.

Finally, because you will see a lot the pragma usage we should go into enough detail so that you know what you're looking at when you see them.

Pragma `Import` and pragma `Export` work almost like a subprogram call. Parameters cannot be omitted unless named notation is used. Reordering the parameters is not permitted, however, unlike subprogram calls.

The BNF syntax as follows. We show `Import`, but `Export` has identical parameters:

```ada
pragma Import(
  [Convention =>] convention_identifier,
  [Entity =>] local_name
[, [External_Name =>] external_name_string_expression]
[, [Link_Name =>] link_name_string_expression])
```

As you can see, the parameters correspond to the individual aspects `Convention`, `External_Name`, and `Link_Name`. When using aspects you don't need to say which Ada entity
Learning Ada

you're applying the aspects to, because the aspects are part of the entity declaration syntax. In contrast, the pragma is distinct from the declaration so we must specify what's being imported or exported via the Entity parameter. That's the declared Ada name, in other words. Note that both the External_Name and Link_Name parameters are optional.

Here's that same built-in function, using the pragma to import it:

```ada
function Sync_Val_Compare_And_Swap_Bool_8
(Destination : access Unsigned_8;
 Comparand   : Unsigned_8;
 New_Value   : Unsigned_8)
return Boolean;
pragma Import (Intrinsic,
 Sync_Val_Compare_And_Swap_Bool_8,
 "__sync_bool_compare_and_swap_1");
```

The first pragma parameter is for the convention. The next parameter, the Entity, is the Ada unit's declared name. The last parameter is the external name. The compiler either knows what we are referencing by that external name or it will reject the pragma. As we mentioned before, the string value for the name is not required to match the Ada unit name.

You will see later that there are other convention identifiers as well, but we will wait for the Specific Interfacing section (page 1161) to introduce those.

### 36.1.4 Package Interfaces

Package Interfaces must be provided by all Ada implementations. The package is intended to provide types that reflect the actual numeric types provided by the target hardware. Of course, the standard has no way to know what hardware is involved, therefore the actual content is implementation-defined. But even so, it is possible to standardize the names for these types, and that is what the language standard does.

Specifically, the standard defines the format for the names for the hardware's signed and modular (unsigned) integer types, and for the floating-point types.

The signed integers have names of the form Integer_n, where n is the number of bits used by the machine-supported type. The type for an eight-bit signed integer would be named Integer_8, for example, and then Integer_16 and so on for the larger types, for as many as the target machine supports.

Likewise, for the unsigned integers, the names are of the form Unsigned_n, with the same meaning for n. The colloquial eight-bit "byte" would be named Unsigned_8, with Unsigned_16 for the 16-bit version, and so on, again for as many as the machine supports.

For floating-point types it is harder to talk about a format that is sufficiently common to standardize. The IEEE floating-point standard is well known and widely used, however, so if the machine does support the IEEE format that name can be used. Such types would be named IEEE_Float_n, again with the same meaning for n. Thus we might see declarations for types IEEE_Float_32 and IEEE_Float_64 and so on, for all the machine supported floating-point types.

In addition to these type declarations, for the unsigned integers only, there will be declarations for shift and rotate operations provided as intrinsic functions.

The resulting package declaration might look something like this:
package Interfaces is

  type Integer_8 is range \(-2 \cdot 2^7..2 \cdot 2^7-1\);
  type Integer_16 is range \(-2 \cdot 2^{15}..2 \cdot 2^{15}-1\);
  type Integer_32 is range \(-2 \cdot 2^{31}..2 \cdot 2^{31}-1\);

  type Unsigned_8 is mod \(2^8\);
  function Shift_Left (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
  function Shift_Right (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
  function Rotate_Left (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
  function Rotate_Right (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
  function Shift_Right_Arithmetic (Value : Unsigned_8; Amount : Natural) return Unsigned_8;

  type Unsigned_16 is mod \(2^{16}\);
  function Shift_Left (Value : Unsigned_16; Amount : Natural) return Unsigned_16;
  function Shift_Right (Value : Unsigned_16; Amount : Natural) return Unsigned_16;

  type Unsigned_32 is mod \(2^{32}\);
  function Shift_Left (Value : Unsigned_32; Amount : Natural) return Unsigned_32;
  function Shift_Right (Value : Unsigned_32; Amount : Natural) return Unsigned_32;

  type IEEE_Float_32 is digits 6;
  type IEEE_Float_64 is digits 15;

end Interfaces;

As you can see, when you need to write code in terms of the hardware's numeric types, this package is a great resource. There's no need to declare your own UInt32 type, for example, although of course you could, trivially:

  type UInt32 is mod \(2^{32}\);

But if you do, realize that you won't get the shift and rotate operations for your type. Those are only defined for the types in package Interfaces. If you do need to declare such a type, and you do want the additional shift/rotate operations, use inheritance:

  type UInt32 is new Interfaces.Unsigned_32;

GNAT also defines a pragma, as an alternative to inheritance:

  type UInt32 is mod \(2^{32}\);
  pragma Provide_Shift_Operators (UInt32);

The approach using inheritance is preferable because it is portable, all other things being equal.

One reason to make up your own unsigned type is that you need one that does not in fact
Learning Ada

reflect the target hardware's numeric types. For example, a hardware device register might have gaps of bits that are currently not used by the device. Those gaps are frequently not the size of a type declared in package Interfaces. We might need an Unsigned_3 type, for example. That's a reasonable thing to do.

36.2 Language-Specific Interfacing

In addition to the aspects and pragmas for importing and exporting entities that work with any language, Ada also defines standard language-specific facilities for interfacing with a set of foreign languages. The standard defines which languages, but vendors can (and do) expand the set.

Specifically, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. Package Interfaces is the root package for a hierarchy of packages that organize these declarations by language, with one or more child packages per language.

Note that the declarations within package Interfaces are, by definition, compile-time visible to any child package in the subsystem. Thus whenever one of the language-specific packages needs to mention the machine types they are automatically available.

The standard defines specific support for foreign languages C, COBOL, and Fortran. Thus there are one or more child packages rooted at Interfaces that have those language names as their child package names: Interfaces.C, Interfaces.COBOL, and Interfaces.Fortran.

The material below will focus on C and, to a lesser extent, Fortran, ignoring altogether the support for COBOL. That's not because COBOL is unimportant. There is a lot of COBOL business software out there in use. Rather, we skip COBOL because it is not relevant to embedded systems. We will provide some information about the Fortran support but will not dwell on it.

Even though we do not consider C to be appropriate for large development projects, neither technically not economically, it has its place in small, low-criticality embedded systems. Ada developers can profit from existing device drivers and mature libraries coded in C, for example. Hence interfacing to it is important.

What about C++? Interfacing to C++ is tricky compared to C, because of the vendor-defined name-mangling, automatic invocations of constructors and destructors, exceptions, and so on. Generally, interfacing with C++ code can be facilitated by preventing much of those difficulties using the extern "C" {... } linkage-specification. Doing so then makes the bracketed C++ code look like C, so the C interfacing facilities then can be used.

36.2.1 Package Interfaces.C

The child package Interfaces.C supports interfacing with units written in the C programming language. Support is in the form of Ada constants and types, and some subprograms. The constants correspond to C's limits.h header file, and the Ada types correspond to types for C's int, short, unsigned_short, unsigned_long, unsigned_char, size_t, and so on. There is also support for converting Ada's type String to/from char_array, and similarly for type Wide_String, etc.

It's a large package so we will elide parts. The idea is to give you a feel for what's there. If you want the details, see either the Ada reference manual or bring up the source code in GNAT Studio.
package Interfaces.C is

-- Declaration's based on C's <limits.h>

CHAR_BIT : constant := 8;
SCHAR_MIN : constant := -128;
SCHAR_MAX : constant := 127;
UCHAR_MAX : constant := 255;

-- Signed and Unsigned Integers. Note that in GNAT, we have ensured that
-- the standard predefined Ada types correspond to the standard C types

type int is new Integer;
type short is new Short_Integer;
type long is range (2 ** (System.Parameters.long_bits - Integer'(1)))
.. +(2 ** (System.Parameters.long_bits - Integer'(1))) - 1;
type long_long is new Long_Long_Integer;

type signed_char is range SCHAR_MIN .. SCHAR_MAX;
for signed_char'Size use CHAR_BIT;

type unsigned is mod 2 ** int'Size;
type unsigned_short is mod 2 ** short'Size;
type unsigned_long is mod 2 ** long'Size;
type unsigned_long_long is mod 2 ** long_long'Size;

...

-- Floating-Point

type C_float is new Float;
type double is new Standard.Long_Float;
type long_double is new Standard.Long_Long_Float;

-----------------------------
-- Characters and Strings --
-----------------------------

type char is new Character;
nul : constant char := char'First;

function To_C (Item : Character) return char;
function To_Ada (Item : char) return Character;

type char_array is array (size_t range =>) of aliased char;
for char_array'Component_Size use CHAR_BIT;

...
end Interfaces.C;

The primary purpose of these types is for use in the formal parameters of Ada subprograms imported from C or exported to C. The various conversion functions can be called from within Ada to manipulate the actual parameters.

When writing the Ada subprogram declaration corresponding to a C function, an Ada procedure directly corresponds to a void function. An Ada procedure also corresponds to a C function if the return value is always to be ignored. Otherwise, the Ada declaration should be a function.

As we said, the types declared in this package can be used as the formal parameter types. That is the intended and recommended approach. However, some Ada types naturally
correspond to C types, and you might see them used instead of those from Interfaces.C. Type `int` is the C native integer type for the target, for example, as is type `Integer` in Ada. Likewise, C's type `float` and type Ada's `Float` are likely compatible. GNAT goes to some lengths to maintain compatibility with C, since the two gcc compilers share so much internal technology. Other vendors might not do so. Best practice is use the types in Interfaces.C for your parameters.

Of course, the types in Interfaces.C are not sufficient for all uses. You will often need to use user-defined types for the formal parameters, such as enumeration types and record types.

Ada enumeration types are compatible with C's enums but note that C requires enum values to be the size of an `int`, whereas Ada does not. The Ada compiler uses whatever sized machine type will support the specified number of enumeral values. It might therefore be smaller than an `int` but it might also be larger. (Declaring more enumeration values than would fit in an integer is unlikely except in tool-generated code, but it is possible.) For example:

```ada
type Small_Enum is (A, B, C);
```

If we printed the object size for `Small_Enum` we'd get 8 (on a typical machine with GNAT). Therefore, applying the aspect Convention to the Ada enumeration type declaration is a good idea:

```ada
type Small_Enum is (A, B, C) with Convention => C;
```

Now the object size will be 32, the same as `int`.

Speaking of enumeration types, note that Ada 2022 added a boolean type to Interfaces.C named `C_Bool` to match that of C99, so you should use it instead of Ada's `Boolean` type for formal parameters.

A simple Ada record type is compatible with a C struct, but remember that the Ada compiler is allowed to reorder the record components. The compiler would do that if it saw that the layout was inefficient, but the point here is that the compiler could do it silently. As a result, you should specify the record layout explicitly using a record representation clause, matching the layout of the C struct in question. Then there will be no question of the layouts matching. Once your record types get more complicated, for example with discriminants or tagged record extensions, things get tricky. Your best bet it to stick with the simple cases when interfacing to C.

Some types that you might think would correspond do not, at least not necessarily. For example, an Ada access type's value might be represented as a simple address, but it might not. In GNAT, an access value designating a value of some unconstrained array type (e.g., `String`) is comprised of two addresses, by default. One designates the characters and the other designates the bounds. You can override that with a pragma, but you must know to do so. For example, if we run the following program, we will see that the object size for the access type Name is twice the object size of System.Address:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System; use System;

procedure Demo is
  type Name is access String;
begin
  Put_Line (Address'Object_Size'Image);
  Put_Line (Name'Object_Size'Image);
end Demo;
```

Some Ada types simply have no corresponding type in C, such as record extensions, task
types, and protected types. You'll have to pass those as an "opaque" type, usually as an address. It isn't clear that a C function would know what to do with values of these types, but the general notion of passing an opaque type as an address is useful and not uncommon. Of course, that approach forgoes all type safety, so avoid it when possible.

In addition to the types for the formal parameters, you'll also need to know how parameters are passed to and from C functions. That affects the parameter profiles on both sides, Ada and C. The text in Annex B for Interfaces.C specifies how parameters are to be passed back and forth between Ada and C so that your subprogram declarations can be portable. That's the approach for each supported programming language, i.e., in the discussion of the corresponding child package under Interfaces.

The rules are expressed in terms of scalar types, "elementary" types, array types, and record types. Remember that scalar types are composed of the discrete types and the real types, so we're talking about the signed and modular integers, enumerations, floating-point, and the two kinds of fixed-point types. The "elementary" types consist of the scalars and access types. The rules are fairly intuitive, but throw in Ada's access parameters and parameter modes and some subtleties arise. We won't cover all the various rules but will explore some of the subtleties.

First, the easy cases: mode in scalar parameters, such as int, as simply passed by copy. Scalar parameters are passed by copy anyway in Ada so the mechanism aligns with C in a straightforward manner. A record type T is passed by reference, so on the C side we'd see t* where t is a C struct corresponding to T. A constrained array type in Ada with a component type T would correspond to a C formal parameter t* where t corresponds to T. An Ada access parameter access T corresponds on the C side to t* where t corresponds to T. And finally, a private type is passed according to the full definition of the type; the fact that it is private is just a matter of controlling the client view, being private doesn't affect how it is passed. There are other simple cases, such as access-to-subprogram types, but we can leave that to the Annex.

Now to the more complicated cases. First, some C ABIs (application binary interfaces) pass small structs by copy instead of by reference. That can make sense, in particular when the struct is small, say the size of an address or smaller. In that case there's no performance benefit to be had by passing a reference. When that situation applies, there is another convention we have not yet mentioned: C_Pass_By_Copy. As a result the record parameter will be passed by copy instead of the default, by reference (i.e., T rather than *T), as long as the mode is in. For example:

```ada
type R2 is record
  V : int;
end record
with Convention => C_Pass_By_Copy;

procedure F2 (P : R2) with
  Import,
  Convention => C,
  External_Name => "f2";

struct R2 { int V; }

void f2 (R2 p);
```

On the C side we expect that p is passed by copy and indeed that is how we find it. That said, passing record values to structs by reference is the more common programmer choice. Like arrays, records are typically larger than an address. The point here is that the Ada code can be configured easily to match the C code.

Next, consider passing array values, both to and from C. When passing an array value to C, remember that Ada array types have bounds. Those bounds are either specified at compile
Array types are not first-class types in C, and C has no notion of unconstrained array types, or even of upper bounds. Therefore, passing an unconstrained array type value is interesting. One approach is to avoid them. Instead, declare a sufficiently large constrained array as a subtype of the unconstrained array type, and then just pass the actual upper bound you want, along with the array object itself.

```ada
type List is array (Integer range <>) of Interfaces.C.int;
subtype Constrained_List is List (1 .. 100);
procedure P (V : Constrained_List; Size : Interfaces.C.int);
pragma Import (C, P, "p");

Obj : Constrained_List := (others => 42);  -- arbitrary values
```

With that, we can just pass the value by reference as usual on the C side:

```c
void p (int* v, int size) {
  // whatever
}
```

But that's assuming we know how many array components are sufficient from the C code's point of view. In the example above we'll pass a value up to 100 to the Size parameter and hope that is sufficient.

Really, it would work to use the unconstrained array type as the formal parameter type instead:

```ada
procedure P (V : List; Size : Interfaces.C.int);
pragma Import (C, P, "p");
```

The C function parameter profile wouldn't change. But why does this work? With values of unconstrained array types, the bounds are stored with the value. Typically they are stored just ahead of the first component, but it is implementation-defined. So why doesn't the above accidentally pass the bounds instead of the first array component itself? It works because we are guaranteed by the Ada language that passing an array will pass (the address of) the components, not the bounds, even for Ada unconstrained array types.

Now for the other direction: passing an array from C to Ada. Here the lack of bounds information on the C side really makes a difference. We can't just pass the array by itself because that would not include the bounds, unlike an Ada call to an Ada routine. In this case the approach is the similar to the first alternative described above, in which we declare a very large array and then pass the bounds explicitly:

```ada
-- DO NOT DECLARE AN OBJECT OF THIS TYPE

procedure P (V : List; Size : Interfaces.C.int);
pragma Export (C, P, "p");
```

```ada
procedure P (V : List; Size : Interfaces.C.int) is
begin
  for J in 0 .. Size - 1 loop
    -- whatever
  end loop;
end P;
```
extern void p (int* v, int size);

int x [100];
p (x, 100);  // call to Ada routine, passing x

The fundamental idea is to declare an Ada type big enough to handle anything conceivably needed on the C side. Subtype Natural means 0 .. Integer'Last so List is quite large indeed. Just be sure never to declare an object of that type. You'll probably run out of storage on an embedded target.

Earlier we said that it is the Ada type that determines how parameters are passed, and that scalars and elementary types are always passed by copy. For mode in that's simple, the copy to the C formal parameter is done and that's all there is to it. But suppose the mode is instead out or in out? In that case the presumably updated value must be returned to the caller, but C doesn't do that by copy. Here the compiler will come to the rescue and make it work, transparently. Specifically, we just declare the Ada subprogram's formal parameter type as usual, but on the C Formal we use a reference. We're talking about scalar and elementary types so let's use int arbitrarily. We make the mode in out but out would also serve:

procedure P (Formal : in out int);

void function p (int* formal);

Now the compiler does its magic: it generates code to make a copy of the actual parameter, but it makes that copy into a hidden temporary object. Then, when calling the C routine, it passes the address of the hidden object, which corresponds to the reference expected on the C side. The C code updates the value of the temporary object via the reference, and then, on return, the compiler copies the value back from the temporary to the actual parameter. Problem solved, if a bit circuitous.

There are other aspects to interfacing with C, such as variadic functions that take a varying number of arguments, but you can find these elsewhere in the learn courses.

Next, we examine the child packages under Interfaces.C. These packages are not used as much as the parent Interfaces.C package so we will provide an overview. You can look up the contents within GNAT Studio or the Ada language standard.

36.2.2 Package Interfaces.C.Strings

Package Interfaces.C declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type chars_ptr corresponds to a common use of char * in C programs, and an object of this type can be passed to imported subprograms for which char * is the type of the argument of the C function. A subset of the package content is as follows:

package Interfaces.C.Strings is

  type chars_ptr is private;
  ...

  function New_Char_Array (Chars : in char_array) return chars_ptr;

  function New_String (Str : in String) return chars_ptr;

  procedure Free (Item : in out chars_ptr);

(continues on next page)
... function Value (Item : in chars_ptr) return char_array; function Value (Item : in chars_ptr) return String; ... function Strlen (Item : in chars_ptr) return size_t; procedure Update (Item : in chars_ptr; Offset : in size_t; Chars : in char_array; Check : in Boolean := True); ... end Interfaces.C.Strings;

Note that allocation might be via malloc, or via Ada’s allocator new. In either case, the returned value is guaranteed to be compatible with char*. Deallocation must be via the supplied procedure Free.

An amusing point is that you can overwrite the end of the char array just like you can in C, via procedure Update. The Check parameter indicates whether overwriting past the end is checked. The default is True, unlike in C, but you could pass an explicit False if you felt the need to do something questionable.

### 36.2.3 Package Interfaces.C.Pointers

The generic package Interfaces.C.Pointers allows us to perform C-style operations on pointers. It includes an access type named Pointer, various Value functions that dereference a Pointer value and deliver the designated array, several pointer arithmetic operations, and "copy" procedures that copy the contents of a source pointer into the array designated by a destination pointer.

We won't go into the details further. See the Ada RM for more.

### 36.2.4 Package Interfaces.Fortran

Like Interfaces.C, package Interfaces.Fortran defines Ada types to be used when working with subprograms using the Fortran calling convention. These types have representations that are identical to the default representations of the Fortran intrinsic types Integer, Real, Double Precision, Complex, Logical, and Character in some supported Fortran implementation. And like the C package, the ways that parameters of various types are passed are also specified.

We leave the details to you to look up in the language standard, if you find them needed in an embedded application.
36.2.5 Machine Code Insertions (MCI)

When working close to the hardware, especially when interacting with a device, it is not uncommon for the hardware to require a very specific set of assembly language instructions to be generated. There are two ways to achieve this: the right way and the wrong way.

The wrong way is to experiment with the source code and compiler switches until you get the exact assembly code you need generated (assuming it is possible at all). But what happens when the next compiler release arrives with a new optimization? And abandon all hope if you go to a new compiler vendor. This approach is both labor-intensive and very brittle.

The right way is to express the precise assembly code sequence explicitly within the Ada source code. (That’s true to any high level language, not just Ada.) Or you can call an intrinsic function, if there is one that does exactly what you need. We will focus on inserting it directly, in what is known as "machine code insertion", or "inline assembler."

As an example of the need for this capability, consider the GPIO (General Purpose I/O) port on an STM32 Arm microcontroller. Each port contains 16 individual I/O pins, each of which can be configured as an independent discrete input or output, or as a control line for a device, with pull-up or pull-down registers, with different clock speeds, and so on. Different on-chip devices use various collections of pins in ways specific to the devices, and require exclusive assignment of the pins. However, any given pin can be used by several different devices. For example, pin 11 on port A ("PA11") can be used by USART #1 as the clear-to-send ("CTS") line, or the CAN #1 bus Rx line, or Channel 4 of Timer 1, among others. Therefore, one of the responsibilities of the system designer is to allocate pins to devices, ensuring that they are allocated uniquely. It is difficult to debug the case in which a pin is accidentally configured for one device and then reconfigured for use with another device (assuming the first device remains in use). To help ensure exclusive allocations, every GPIO port on this Arm implementation has a way of locking the configuration of each I/O pin. That way, some other part of the software can't successfully change the configuration accidentally, for use with some other device. Even if the same configuration was to be used for another device, the lock prevents the accidental update so we find out about the unintentional sharing.

To lock a pin on a port requires a special sequence of reads and writes to a GPIO register for that port. A specific bit pattern is required during the reads and writes. The sequence and bit pattern is such that accidentally locking the pin is highly unlikely.

Once we see how to express assembly language sequences in general we will see how to get the necessary sequence to lock a port/pin pair. Unfortunately, although you can express exactly the code sequence required, such a sequence of assembly language instructions is clearly target hardware-specific. That means portability is inherently limited. Moreover, the syntax for expressing it varies with the vendor, even for the same target hardware. Being able to insert it at the Ada source level doesn't help with either portability issue. You should understand that the use-case for machine code insertion is for small, short sequences. Otherwise you would write the code in assembly language directly, in a separate file. That might obtain a degree of vendor independence, at least for the given target, but not necessarily. The use of inline assembler is intended for cases in which a separate file containing assembly language is not simpler.

With those caveats in place, let's first examine how to do it in general and then how to express it with GNAT specifically.

The right way to express an arbitrary sequence of one or more assembly language statements is to use so-called "code statements." A code statement is an Ada statement, but it is also a qualified expression of a type defined in package System.Machine_Code. The content of that package, and the details of code statements, are implementation-defined. Although that affects portability there really is no alternative because we are talking about machine instruction sets, which vary considerably and cannot be standardized at this level.

Package System.Machine_Code contains types whose values provide a way of expressing
assembly instructions. For example, let's say that there is a "HLT" instruction that halts the processor for some target. There is no other parameter required, just that op-code. Let's also say that one of the types in System.Machine_Code is for these "short" instructions consisting only of an op-code. The syntax for the type declaration would then allow the following code statement:

```ada
Short_Instruction'(Command => HLT);
```

Each of Short_Instruction, Command, and HLT are defined by the vendor in this hypothetical version of package System.Machine_Code. You can see why we say that it is both a statement (note the semicolon) and a qualified expression (note the apostrophe). Code statements must appear in a subprogram body, after the `begin`. Only code statements are allowed in such a body, only use-clauses can be in the declarative part, and no exception handlers are allowed. The complete example would be as follows:

```ada
procedure Halt -- stops processor
   with Inline;

procedure Halt is
begin
   Short_Instruction'(Command => HLT);
end Halt;
```

With that, to halt the processor the Ada code can simply call procedure Halt. When the optimizer is enabled there will be no code emitted to make the call, we'd simply see the halt instruction emitted directly in-line.

Package System.Machine_Code provides access to machine instructions but as we mentioned, the content is vendor-defined. In addition, the package itself is optional, but is required if Annex C, the Systems Programming Annex, is implemented by the vendor. In practice most all vendors provide this annex.

In GNAT, the content of System.Machine_Code looks something like this:

```ada
type Asm_Input_Operand is ...
type Asm_Output_Operand is ...
type Asm_Input_Operand_List is array (Integer range <>) of Asm_Input_Operand;
type Asm_Output_Operand_List is array (Integer range <>) of Asm_Output_Operand;

type Asm_Insn is private;
...

function Asm
   (Template : String;
    Outputs : Asm_Output_Operand := No_Output_Operands;
    Inputs : Asm_Input_Operand := No_Input_Operands;
    Clobber : String := "";
    Volatile : Boolean := False) return Asm_Insn;
```

With this package content, the expression in a code statement is of type Asm_Insn, short for "assembly instruction." Multiple overloaded functions named Asm return values of that type.

The Template parameter in a string containing one or more assembly language instructions. These instructions are specific to the target machine. The parameter Outputs provides mappings from registers to source-level entities that are updated by the assembly statement(s). Inputs provides mappings from source-level entities to registers for inputs. Volatile, when True, tells the compiler not to optimize the call away, and Clobber tells the compiler which registers, or memory, if any, are altered by the instructions in Template.
("Clobber" is colloquial English for "destroy.") That last is important because the compiler was likely already using some of those registers so the compiler will need to restore them after the call.

We could say, for example, the following, taking all the defaults except for Volatile:

```ada
Asm ("nop", Volatile => True);
```

As you can imagine the full details are extensive, beyond the scope of this introduction. See the GNAT User Guide ("Inline Assembler") for all the gory details.

Now, back to our GPIO port/bin locking example. The port type is declared as follows:

```ada
type GPIO_Port is limited record
  ... LCKR : Word with Atomic; -- lock register ... end record with ...;
```

We've elided all but the LCKR component representing the "lock register" within each port. We'd have a record representation clause to ensure the required layout but that's not important here. Word is an unsigned (modular) 32-bit integer type. One of the hardware requirements for accessing the lock register is that the entire register has to be read or written whenever any bits within it are accessed. The compiler must not, for example, write one of the bytes within the register in order to set or clear a bit within that part of the register. Therefore we mark the register as Atomic. If the compiler cannot honor that aspect the compilation will fail, so we would know there is a problem.

Per the ST Micro Reference Manual, the lock control bit is referred to as LCKK and is bit #16, i.e., the first in the upper half of the LCKR register word.

```
LCKK : constant Word := 16#0001_0000#; -- the "lock control bit"
```

That bit is also known as the "Lock Key" (hence the abbreviation) because it is used to control the locking of port/pin configurations.

There are 16 GPIO pins per port, represented by the lower 16 bits of the register. Each one of these 16 bits corresponds to one of the 16 GPIO pins on a port. If any given bit reads as a 1 then the corresponding pin is locked.

Graphically that looks like this:

![Diagram of GPIO Port Locking]

Therefore, the Ada types are:

```ada
type GPIO_Pin is
  (Pin_0, Pin_1, Pin_2, Pin_3, Pin_4, Pin_5, Pin_6, Pin_7,
   Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);
for GPIO_Pin use (Pin_0 => 16#0001#, Pin_1 => 16#0002#, Pin_2 => 16#0004#,
                  ... Pin_15 => 16#8000#);`
Note that we had to override the default enumeration representation so that each pin — each enumerai value — would occupy a single dedicated bit in the bit-mask.

With that in place, let’s lock a pin. A specific sequence is required to set a pin’s lock bit. The sequence writes and reads values from the port’s LCKR register. Remember that this 32-bit register has 16 bits for the pin mask (0 .. 15), with bit #16 used as the "lock control bit".

1. write a 1 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
2. write a 0 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
3. do step 1 again
4. read the entire LCKR register
5. read the entire LCKR register again (optional)

Throughout the sequence the same value for the lower 16 bits of the word must be maintained (i.e., the pin mask), including when clearing the LCCK bit in the upper half.

If we wrote this in Ada it would look like this:

```ada
procedure Lock (Port : in out GPIO_Port; Pin : GPIO_Pin) is
  Temp : Word with Volatile;
begin
  -- set the lock control bit and the pin bit, clear the others
  Temp := LCCK or Pin'Enum_Rep;
  -- write the lock and pin bits
  Port.LCKR := Temp;
  -- clear the lock bit in the upper half
  Port.LCKR := Pin'Enum_Rep;
  -- write the lock bit again
  Port.LCKR := Temp;
  -- read the lock bit
  Temp := Port.LCKR;
  -- read the lock bit again
  Temp := Port.LCKR;
end Lock;
```

`Pin'Enum_Rep` gives us the underlying value for the enumeration value. We cannot use 'Pos because that attribute provides the logical position number within the enumerated values, and as such always increases consecutively. We need the underlying representation value that we specified explicitly.

The Ada procedure works, but only if the optimizer is enabled (which also precludes debugging). But even so, there is no guarantee that the required assembly language instruction sequence would be generated, especially one that maintains that required bit mask value on each access. A machine-code insertion is appropriate for all the reasons presented earlier:

```ada
procedure Lock (Port : in out GPIO_Port;
               Pin : GPIO_Pin) is
  use System.Machine_Code, ASCII, System;
begin
  Asm ("orr r3, %1, #65536" & LF & HT & -- 0) Temp := LCCK or Pin'Enum_Rep
  "str r3, [%0, #28]" & LF & HT & -- 1) Port.LCKR := Temp
  "str %1, [%0, #28]" & LF & HT & -- 2) Port.LCKR := Pin'Enum_Rep
  "str r3, [%0, #28]" & LF & HT & -- 3) Port.LCKR := Temp
  "ldr r3, [%0, #28]" & LF & HT & -- 4) Temp := Port.LCKR
  "ldr r3, [%0, #28]" & LF & HT, -- 5) Temp := Port.LCKR
Inputs => (Address'Asm_Input ("r", This'Address), -- %0
            (GPIO_Pin'Asm_Input ("r", Pin)), -- %1
Volatile => True,
  (continues on next page)"
Learning Ada

We've combined the instructions into one Asm expression. As a result, we can use ASCII line-feed and horizontal tab characters to format the listing produced by the compiler so that each instruction is on a separate line and aligned with the previous instruction, as if we had written the sequence in assembly language directly. That enhances readability later, during examination of the compiler output to verify the required sequence was emitted.

In the above, "%0" is the first input, containing the address of the Port parameter. "%1" is the other input, the value of the Pin parameter. We're using register r3 explicitly, as the "temporary" variable, so we tell the compiler that it has been "clobbered."

If we examine the assembly language output from compiling the file, we find the body of procedure Lock is as hoped:

```
ldr  r2, [r0, #4]
ldrh r1, [r0, #8]
.syntax unified
orr r3, r1, #65536
str r3, [r2, #28]
str r1, [r2, #28]
str r3, [r2, #28]
ldr r3, [r2, #28]
ldr r3, [r2, #28]
```

The first two statements load register 2 (r2) and register 1 (r1) with the subprogram parameters, i.e., the port and pin, respectively. Register 2 gets the starting address of the port record, in particular. (Offset #28 is the location of the LCKR register. The port is passed by reference so that address is actually that of the hardware device.)

We will have separately declared procedure Lock with inlining enabled, so whenever we call the procedure we will get the exact assembly language sequence required to lock the indicated pin on the given port, without any additional code for a procedure call.

Note that we get the calling convention right automatically, because the subprogram is not a foreign entity written in some other language (such as assembly language). It's an Ada subprogram with special content so the Ada convention applies as usual.

36.3 When Ada Is Not the Main Language

When multiple programming languages are involved, the main procedure might not be implemented in Ada. Maybe the bulk of the program is written in C, for example, and this C code calls some Ada routines that have been exported (with the C convention).

That means the Ada builder does not create the executable image's entry point. In fact the Ada main procedure is never the entry point for the final executable image, it's just where the application code begins, like the C main function. There are setup and initialization steps that must happen before any program can execute on a target, and the entry point code is responsible for this functionality. For example, on a bare machine target, the hardware must be initialized, the trap vectors installed, the segments initialized, and so on. On a target running an operating system, the OS is responsible for that initialization but there will be OS-specific initialization steps too. For example, if command-line arguments are supported these may be gathered. All this initialization code is generated by the builder, regardless of the language, followed by a call to the main routine.

Some of the initialization is specific to Ada programming, and must occur before any calls occur to the exported Ada routines. In particular, the entry point code emitted by the Ada builder initializes the Ada run-time system and calls all the elaboration routines for the
library units in the application code. Only then does the emitted code invoke the Ada main. If the Ada builder is not going to create the executable it has no chance to emit the code to do that prior initialization. A foreign language builder will not emit such code, so we have a problem.

You could learn enough about how the foreign builder works, and how your Ada builder works, to create a work-around. You could learn what the Ada builder would emit, in other words, and ensure those routines are called manually, either directly or by augmenting the builder scripts (assuming that's possible). But the work-around would be labor-intensive and not robust to changes by the tool vendors. It would be an ugly hack, in other words.

That work-around would not be portable either. The Ada standard can't address hardware- or OS-specific initialization, but it can standardize the name for a routine to do the Ada-specific initialization. Specifically, procedure adainit initializes the Ada application code and the Ada run-time library. Similarly, one might need to shut down the Ada code when no further calls will be made to the exported Ada routines. Procedure adafinal performs this shut-down functionality. Neither procedure has parameters.

The main function in the other language is intended to import these routines and manually call them each exactly once. adainit must be called prior to any calls to the Ada code, and adafinal is to be called after all the calls to the Ada code.

For example:

```c
#include "stdio.h"

extern int checksum (char *input, int count);
extern void adainit (void);
extern void adafinal (void);

int main (int argc, char *argv[]) {
  char * Str = "Hello World!";
  int sum;
  adainit ();
  sum = checksum (Str, strlen (Str));
  adafinal ();
  printf ("checksum for '%s' is %d", Str, sum);
  return 0;
}
```

In the above, we have an Ada routine to compute a checksum, called by a C main function. Therefore, we use "extern" to tell the C compiler that the "checksum" function is defined elsewhere, i.e., in the Ada routine. Likewise, we tell the compiler that functions adainit and adafinal are defined elsewhere. The call to adainit is made before the call to any Ada code, thus all the elaboration code is guaranteed to happen before checksum needs it. Once the Ada code is not needed, the call to adafinal can be made.

Both adainit and adafinal have no effect after the first invocation. That means you cannot structure your foreign code to iteratively call the two routines whenever you want to invoke some Ada code. In practice you just call them once in the main and be done with it.
Interacting with hardware devices is one of the more frequent activities in embedded systems programming. It is also one of the most enjoyable because you can make something happen in the physical world. There’s a reason that making an LED blink is the “hello world” of embedded programming. Not only is it easy to do, it is surprisingly satisfying. I suspect that even the developers of "Full Authority Digital Engine Controllers" (FADEC) — the computers that are in complete, total control of commercial airline engines — have fond memories of making an LED blink early in their careers. And of course a blinking LED is a good way to indicate application status, especially if off-board I/O is limited, which is often the case.

Working at the device register level can be error prone and relatively slow, in terms of source-lines-of-code (SLOC) produced. That’s partly because the hardware is in some cases complicated, and partly because of the way the software is written. Using bit masks for setting and clearing bits is not a readable approach, comparatively speaking. There’s just not enough information transmitted to the reader. It might be clear enough when written, but will you see it that way months later? Readability is important because programs are read many more times than they are written. Also, an unreadable program is more difficult to maintain, and maintenance is where most money is spent in long-lived applications. Comments can help, until they are out of date. Then they are an active hindrance.

For example, what do you think the following code does? This is real code, where temp and temp2 are unsigned 32-bit integers:

```c
temp = ((uint32_t)(GPIO_AF) << 
        ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~(((uint32_t)0xF << 
        ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
```

That’s unfair to ask, absent any context. The code configures a general purpose I/O (GPIO) pin on an Arm microcontroller for one of the “alternate functions”. GPIOx is a pointer to a GPIO port, GPIO_PinSource is a GPIO pin number, and GPIO_AF is the alternate function number. But let’s say you knew that. Is the code correct? The longer it takes to know, the less productive you are.

The fact that the code above is in C is beside the point. If we wrote it the same way in Ada it would be equally opaque, if not more so. There are simpler approaches. Judicious use of record and array types is one. We’ll say more about that later, but the underlying idea is to let the compiler do as much work for us as possible. For example, the data structures used in the code above require explicit shifting whenever they are accessed. If we can avoid that at the source code level — by having the compiler do it for us — we will have simplified the code considerably. Furthermore, letting the compiler do the work for us makes the code more maintainable (which is where the money is). For example, if the code does the shifting explicitly and the data structures are changed, we’ll have to change the number of bits to shift left or right. Constants will help there, but we still have to remember to change them;
the compiler won't complain if we forget. In contrast, if we let the compiler do this shifting for us, the amounts to shift will be changed automatically.

Some devices are very simple. In these cases the application may interact directly with the device without unduly affecting productivity. For example, there was a board that had a user-accessible rotary switch with sixteen distinct positions. Users could set the switch to whatever the application code required, e.g., to indicate some configuration information. The entire software interface to this device consisted of a single read-only 8-bit byte in memory. That's all there was to it: you read the memory and thus got the numeric setting of the switch.

More complex devices, however, usually rely on software abstraction to deal with the complexity. Just as abstraction is a fundamental way to combat complexity in software, abstraction also can be used to combat the complexity of driving sophisticated hardware. The abstraction is presented to users by a software "device driver" that exists as a layer between the application code and the hardware device. The layer hides the gory details of the hardware manipulation behind subprograms, types, and parameters.

We say that the device driver layer is an abstraction because, at the least, the names of the procedures and functions indicate what they do, so at the call site you can tell what is being done. That's the point of abstraction: it allows us to focus on what, rather than how. Consider that GPIO pin configuration code block again. Instead of writing that block every time we need to configure the alternate function for a pin, suppose we called a function:

```ada
GPIO_PinAFConfig(USARTx_TX_GPIO_PORT, USARTx_TX_SOURCE, USARTx_TX_AF);
```

The `GPIO_PinAFConfig` function is part of the GPIO device driver provided by the STM32 Standard Peripherals Library (SPL). Even though that's not the best function name conceivable, calls to the function will be far more readable than the code of the body, and we only have to make sure the function implementation is correct once. And assuming the device drivers' subprograms can be inlined, the subprogram call imposes no performance penalty.

Note the first parameter to the call above: `USARTx_TX_GPIO_PORT`. There are multiple GPIO ports on an Arm implementation; the vendor decides how many. In this case one of them has been connected to a USART (Universal Synchronous Asynchronous Receiver Transmitter), an external device for sending and receiving serial data. When there are multiple devices, good software engineering suggests that the device driver present a given device as one of a type. That's what an "abstract data type" (ADT) provides for software and so the device driver applies the same design. An ADT is essentially a class, in class-oriented languages. In Ada, an ADT is represented as a private type declared in a package, along with subprograms that take the type as a parameter.

The Ada Drivers Library (ADL) provided by AdaCore and the Ada community uses this design to supply Ada drivers for the timers, I2C, A/D and D/A converters, and other devices common to microcontrollers. Multiple devices are presented as instances of abstract data types. A variety of development platforms from various vendors are supported, including the STM32 series boards. The library is available on GitHub for both non-proprietary and commercial use here: [https://github.com/AdaCore/Ada_Drivers_Library](https://github.com/AdaCore/Ada_Drivers_Library). We are going to use some of these drivers as illustrations in the following sections.
37.1 Non-Memory-Mapped Devices

Some devices are connected to the processor on a dedicated bus that is separate from the memory bus. The Intel processors, for example, used to have (and may still have) instructions for sending and receiving data on this bus. These are the "in" and "out" instructions, and their data-length specific variants.

The original version of Ada defined a package named `Low_Level_IO` for such architectures, but there were very few implementations (maybe just one, known to support the Intel processors). As a result, the package was actually removed from the language standard. Implementations could still support the package, it just wouldn't be a standard package. That's different from constructs that are marked as "obsolescent" by the standard, e.g., the pragmas replaced by aspects, among other things. Obsolescent constructs are still part of the standard.

If a given target machine has such I/O instructions for the device bus, these can be invoked in Ada via machine-code insertions. For example:

```ada
procedure Send_Control (Device : Port; Data : Unsigned_16) is
    pragma Suppress (All_Checks);
begin
    asm (
        "outw %1, (%0)",
        Inputs  => (Port'Asm_Input("dx",Device),
                    Unsigned_16'Asm_Input("ax",Data)),
        Clobber => "ax, dx");
end Send_Control;

procedure Receive_Control (Device : Port; Data : out Unsigned_16) is
    pragma Suppress (All_Checks);
begin
    asm (
        "inw (%1), %0",
        Inputs  => (Port'Asm_Input("dx",Device)),
        Outputs => (Unsigned_16'Asm_Output("=ax",Data)),
        Clobber => "ax, dx",
        Volatile => True);
end Receive_Control;
```

Applications could use these subprograms to set the frequency of the Intel PC tone generator, for example, and to turn it on and off. (You can't do that any more in application code because modern operating systems don't give applications direct access to the hardware, at least not by default.)

Although the `Low_Level_IO` package is no longer part of the language, you can write this sort of thing yourself, or vendors can do it. That's possible because the Systems Programming Annex, when implemented, guarantees fully effective use of machine-code inserts. That means you can express anything the compiler could emit. The guarantee is important because otherwise the compiler might "get in the way." For example, absent the guarantee, the compiler would be allowed to insert additional assembly language statements in between yours. That can be a real problem, depending on what your statements do. For instance, if your MCI assembly statements do something and then check a resulting condition code, such as the overflow flag, those interleaved compiler-injected statements might clear that condition code before your code can check it. Fortunately, the annex guarantees that sort of thing cannot happen.
37.2 Memory-Mapped Devices

In another earlier chapter (page 1117), we said that we could query the address of some object, and we also showed how to use that result to specify the address of some other object. We used that capability to create an "overlay," in which two objects are used to refer to the same memory locations. As we indicated in that discussion, you would not use the same type for each object — the point, after all, is to provide a view of the shared underlying memory cells that is not already available otherwise. Each distinct type would provide a distinct view of the memory values, that is, a set of operations providing some required functionality.

For example, here’s an overlay composed of a 32-bit signed integer object and a 32-bit array object:

```ada
type Bits32 is array (0 .. 31) of Boolean
   with Component_Size => 1;
X : aliased Integer_32;
Y : Bits32 with Address => X'Address;
```

Because one view is as an integer and the other as an array, we can access that memory using the two different views' operations. Using the view as an array object (Y) we can access individual bits of the memory shared with X. Using the view as an integer (X), we can do arithmetic on the contents of that memory. (We could have used an unsigned integer instead of the signed type, and thereby gained the bit-oriented operations, but that’s not the point.)

Very often, though, there is only one Ada object that we place at some specific address. That’s because the Ada object is meant to be the software interface to some memory-mapped hardware device. In this scenario we don’t have two overlaid Ada objects, we just have one. The other "object" is the hardware device mapped to that starting address. Since they are at the same memory location(s), accessing the Ada object accesses the hardware device.

For a real-world but nonetheless simple example, recall that example of a rotary switch on the front of our embedded computer that we mentioned in the introduction. This switch allows humans to provide some very simple input to the software running on the computer.

```ada
Rotary_Switch : Unsigned_8 with
   Address => System.Storage_Elements.To_Address (16#FFC0_0801#);
```

We declare the object and also specify the address, but not by querying some entity. We already know the address from the hardware documentation. But we cannot simply use an integer address literal from that documentation because type System.Address is almost always a private type. We need a way to compose an Address value from an integer value. The package System.Storage_Elements defines an integer representation for Address values, among other useful things, and a way to convert those integer values to Address values. The function To_Address does that conversion.

As a result, in the Ada code, reading the value of the variable Rotary_Switch reads the number on the actual hardware switch.

Note that if you specify the wrong address, it is hard to say what happens. Likewise, it is an error for an address clause to disobey the object's alignment. The error cannot be detected at compile time, in general, because the address is not necessarily known at compile time. There’s no requirement for a run-time check for the sake of efficiency, since efficiency seems paramount here. Consequently, this misuse of address clauses is just like any other misuse of address clauses — execution of the code is erroneous, meaning all bets are off. You need to know what you’re doing.

What about writing to the variable? Is that meaningful? In this particular example, no. It
is effectively read-only memory. But for some other device it very well could be meaningful, certainly. It depends on the hardware. But in this case, assigning a value to the Rotary_Switch variable would have no effect, which could be confusing to programmers. It looks like a variable, after all. We wouldn't declare it as a constant because the human user could rotate the switch, resulting in a different value read. Therefore, we would hide the Ada variable behind a function, precluding the entire issue. Clients of the function can then use it for whatever purpose they require, e.g., as the unique identifier for a computer in a rack.

Let's talk more about the type we use to represent a memory-mapped device. As we said, that type defines the view we have for the object, and hence the operations we have available for accessing the underlying mapped device.

We choose the type for the representative Ada variable based on the interface of the hardware mapped to the memory. If the interface is a single monolithic register, for example, then an integer (signed or unsigned) of the necessary size will suffice. But suppose the interface is several bytes wide, and some of the bytes have different purposes from the others? In that case, a record type is the obvious solution, with distinct record components dedicated to the different parts of the hardware interface. We could use individual bits too, of course, if that's what the hardware does. Ada is particularly good at this fine-degree of representation because record components of any types can be specified in the layout, down to the bit level, within the record.

In addition, we might want to apply more than one type, at any one time, to a given memory-mapped device. Doing so allows the client code some flexibility, or it might facilitate an internal implementation. For example, the STM32 boards from ST Microelectronics include a 96-bit device unique identifier on each board. The identifier starts at a fixed memory location. In this example we provide two different views — types — for the value. One type provides the identifier as a String containing twelve characters, whereas another type provides the value as an array of three 32-bit unsigned words (i.e., 12 bytes). The two types are applied by two overloaded functions that are distinguished by their return type:

```ada
package STM32.Device_Id is

  subtype Device_Id_Image is String (1 .. 12);

  function Unique_Id return Device_Id_Image;

  type Device_Id_Tuple is array (1 .. 3) of UInt32
    with Component_Size => 32;

  function Unique_Id return Device_Id_Tuple;
end STM32.Device_Id;
```

The subtype Device_Id Image is the view of the 96-bits as an array of twelve 8-bit characters. (Using type String here isn't essential. We could have defined an array of bytes instead of Character.) Similarly, subtype Device_Id_Tuple is the view of the 96-bits as an array of three 32-bit unsigned integers. Clients can then choose how they want to view the unique id by choosing which function to call.

In the package body we implement the functions as two ways to access the same shared memory:

```ada
with System;

package body STM32.Device_Id is

  ID_Address : constant System.Address := System'To_Address (16#1FFF_7A10#);

  function Unique_Id return Device_Id_Image is
```
Result : Device_Id_Image with Address => ID_Address, Import;
begin
  return Result;
end Unique_Id;

function Unique_Id return Device_Id_Tuple is
  Result : Device_Id_Tuple with Address => ID_Address, Import;
begin
  return Result;
end Unique_Id;
end STM32.Device_Id;

The GNAT-defined attribute System'To_Address in the declaration of ID_Address is the same as the function System.Storage_Elements.To_Address except that, if the argument is static, the function result is static. This means that such an expression can be used in contexts (e.g., preelaborable packages) which require a static expression and where the function call could not be used (because the function call is always non-static, even if its argument is static).

The only difference in the bodies is the return type and matching type for the local Result variable. Both functions read from the same location in memory.

Earlier we indicated that the bit-pattern implementation of the GPIO function could be expressed differently, resulting in more readable, therefore maintainable, code. The fact that the code is in C is irrelevant; the same approach in Ada would not be any better. Here's the complete code for the function body:

```c
void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                      uint16_t GPIO_PinSource,
                      uint8_t GPIO_AF)
{
    uint32_t temp = 0x00;
    uint32_t temp_2 = 0x00;

    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
    assert_param(IS_GPIO_AF(GPIO_AF));

    temp = ((uint32_t)(GPIO_AF)) <<
            (uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4);
    GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~(((uint32_t)0xF <<
                                             (uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
    temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
    GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}
```

The problem, other than the magic numbers (some named constants would have helped), is that the code is doing nearly all the work instead of off-loading it to the compiler. Partly that's because in C we cannot declare a numeric type representing a 4-bit quantity, so everything is done in terms of machine units, in this case 32-bit unsigned integers.

Why do we need 4-bit values? At the hardware level, each memory-mapped GPIO port has a sequence of 16 4-bit quantities, one for each of the 16 pins on the port. Those 4-bit quantities specify the "alternate functions" that the pin can take on, if needed. The alternate functions allow a given pin to do more than act as a single discrete I/O pin. For example, a pin could be connected to the incoming lines of a USART. We use the configuration routine to apply the specific 4-bit code representing the alternate function required for our application.

These 16 4-bit alternate function fields are contiguous in the register (hence memory) so we can represent them as an array with a total size of 64-bits (i.e., 16 times 4). In the C
version this array has two components of type uint32_t so it must compute where the corresponding 4-bit value for the pin is located within those two words. In contrast, the Ada version of the array has components of the 4-bit type, rather than two 32-bit components, and simply uses the pin number as the index. The resulting Ada procedure body is extremely simple:

```ada
procedure Configure_Alt Amountشاب Function
(Port : in out GPIO_Port;
 Pin : GPIO_Pin;
AF : GPIO_Alternate_Function_Code)
is
begin
  Port.AFR (Pin) := AF;
end Configure_Alt Amountشاب Function;
```

In the Ada version, AFR is a component within the GPIO_Port record type, much like in the C code's struct. However, Ada allows us to declare a much more descriptive set of types, and it is these types that allows the developer to off-load the work to the compiler.

First, in Ada we can declare a 4-bit numeric type:

```ada
type Bits_4 is mod 2**4 with Size => 4;
```

The Bits_4 type was already globally defined elsewhere so we just derive our 4-bit "alternate function code" type from it. Doing so allows the compiler to enforce simple strong typing so that the two value spaces are not accidentally mixed. This approach also increases understanding for the reader:

```ada
type GPIO_Alternate_Function_Code is new Bits_4;
-- We cannot use an enumeration type because there are duplicate binary
-- values
```

Hence type GPIO_Alternate_Function_Code is a copy of Bits_4 in terms of operations and values, but is not the same type as Bits_4 so the compiler will keep them separate for us.

We can then use that type as the array component type for the representation of the AFR:

```ada
type Alternate_Function_Fields is
array (GPIO_Pin) of GPIO_Alternate_Function_Code
with Component_Size => 4, Size => 64; -- both in units of bits
```

Note that we can use the GPIO Pin parameter directly as the index into the array type, obviating any need to massage the Pin value in the procedure. That's possible because the type GPIO_Pin is an enumeration type:

```ada
type GPIO_Pin is
(Pin_0, Pin_1, Pin_2, Pin_3, Pin_4, Pin_5, Pin_6, Pin_7,
 Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);
```

for GPIO_Pin use
(Pin_0 => 16#0001#,
 Pin_1 => 16#0002#,
 Pin_2 => 16#0004#,
 Pin_3 => 16#0008#,
 Pin_4 => 16#0010#,
 Pin_5 => 16#0020#,
 Pin_6 => 16#0040#,
 Pin_7 => 16#0080#,
 Pin_8 => 16#0100#,
 Pin_9 => 16#0200#,
 Pin_10 => 16#0400#,
(continues on next page)
In the hardware, the GPIO_Pin values don't start at zero and monotonically increase. Instead, the values are bit patterns, where one bit within each value is used. The enumeration representation clause allows us to express that representation.

Type Alternate_Function_Fields is then used to declare the AFR record component in the GPIO_Port record type:

```ada
type GPIO_Port is limited record
  MODER : Pin_Modes_Register;
  OTYPER : Output_Types_Register;
  Reserved_1 : Half_Word;
  OSPEEDR : Output_Speeds_Register;
  PUPDR : Resistors_Register;
  IDR : Half_Word; -- input data register
  Reserved_2 : Half_Word;
  ODR : Half_Word; -- output data register
  Reserved_3 : Half_Word;
  BSRR_Set : Half_Word; -- bit set register
  BSRR_Reset : Half_Word; -- bit reset register
  LCKR : Word with Atomic;
  AFR : Alternate_Function_Fields;
  Unused : Unaccessed_Gap;
end record
```

These declarations define a record type that matches the content and layout of the STM32 GPIO Port memory-mapped device.

Let's compare the two procedure implementations again. Here they are, for convenience:

```ada
void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                       uint16_t GPIO_PinSource,
                       uint8_t GPIO_AF)
{
  uint32_t temp = 0x00;
  uint32_t temp_2 = 0x00;

  /* Check the parameters */
```

(continues on next page)
assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
assert_param(IS_GPIO_AF(GPIO_AF));

    temp = ((uint32_t)(GPIO_AF) << (((uint32_t)(GPIO_PinSource & (uint32_t)0x07) * 4));
GPIOx->AFR[GPIO_PinSource >> 0x03] &= -((uint32_t)0x0F << (((uint32_t)(GPIO_PinSource & (uint32_t)0x07) * 4));

    temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}

procedure Configure_Alternate_Function
(Port : in out GPIO_Port;
 Pin : GPIO_Pin;
 AF : GPIO_Alternate_Function_Code)
is
begin
  Port.AFR (Pin) := AF;
end Configure_Alternate_Function;

Which one is correct? Both. But clearly, the Ada version is far simpler, so much so that it is immediately obvious that it is correct. Not so for the coding approach used in the C version, comparatively speaking. It is true that the Ada version required a couple more type declarations, but those make the procedure body far simpler. That resulting simplicity is a reflection of the balance between data structures and executable statements that we should always try to achieve. Ada just makes that easier to achieve than in some other languages.

Of course, the underlying hardware likely has no machine-supported 4-bit unsigned type so larger hardware numeric types are used in the generated code. Hence there are shifts and masking being done in the Ada version as well, but they do not appear in the source code. The developer has let the compiler do that work. An additional benefit of this approach is that the compiler will change the shifting and masking code for us if we change the explicit type declarations.

Why is simplicity so important? Simplicity directly increases understandability, which directly affects correctness and maintainability, which greatly affects the economic cost of the software. In large, long-lived projects, maintenance is by far the largest economic cost driver. In high-integrity applications, correctness is essential. Therefore, doing anything reasonable to keep the code as simple as possible is usually worth the effort. In some projects the non-functional requirements, especially performance, can dictate less simple code, but that won't apply to all of the code. Where possible, simplicity rules.

One more point about the GPIO ports. There are as many of these ports as the Arm microcontroller vendor decides to implement. And as we said, they are memory-mapped, at addresses specified by the vendor. If the memory used by all the ports is contiguous, we can conveniently use an array of the GPIO_Port record type to represent all the ports implemented. We would just set the array object's address at the address specified for the first port object in memory. Then, normal array indexing will provide access to any given port in the memory-mapped hardware.

This array approach requires each array component — the GPIO_Port record type — to be the right size so that all the array components start on addresses corresponding to the start of the next port in hardware.

That starting address correspondence for the array components is obtained automatically as long as the record type includes all the memory used by any individual device. In that case the next array component will indeed start at an address matching the next device in hardware. Note that this assumes the first array component matches the address of the first hardware device in memory. The first array component is at the same address as the
Learning Ada

whole array object itself (a fact that is guaranteed by the language), so the array address must be set to whatever the vendor documentation specified for the first port.

However, in some cases the vendor will leave gaps of unused memory for complicated memory-mapped objects like these ports. They do so for the sake of future expansion of the implementation, e.g., to add new features or capacity. The gaps are thus between consecutive hardware devices.

These gaps are presumably (hopefully!) included in the memory layout documented for the device, but it won't be highlighted particularly. You should check, therefore, that the documented starting addresses of the second and subsequent array components are what you will get with a simple array object having components of that record type.

For example, the datasheet for the STM32F407 Arm implementation indicates that the GPIO ports start at address 16#4002_0000#. That's where GPIO_A begins. The next port, GPIO_B, starts at address 16#4002_0400#, or a byte offset of 1024 in decimal. In the STM32F4 Reference Manual, however, the GPIO port register layout indicates a size for any one port that is much less than 1024 bytes. As you saw earlier in the corresponding record type declaration, on the STM32F4 each port only requires 40 (decimal) bytes. Hence there's a gap of unused memory between the ports, including after the last port, of 984 bytes (7872 bits).

To represent the gap, an "extra", unused record component was added, with the necessary location and size specified within the record type, so that the unused memory is included in the representation. As a result, each array component will start at the right address (again, as long as the first one does). Telling the compiler, and future maintainers, that this extra component is not meant to be referenced by the software would not hurt. You can use the pragma or aspect Unreferenced for that purpose. Here's the code again, for convenience:

```ada
type GPIO_Port is limited record
  MODER : Pin_Modes_Register;
  OTYPER : Output_Types_Register;
  Reserved_1 : Half_Word;
  OSPEEDR : Output_Speeds_Register;
  PUPDR : Resistors_Register;
  IDR : Half_Word; -- input data register
  Reserved_2 : Half_Word;
  ODR : Half_Word; -- output data register
  Reserved_3 : Half_Word;
  BSRR_Set : Half_Word; -- bit set register
  BSRR_Reset : Half_Word; -- bit reset register
  LCKR : Word with Atomic;
  AFR : Alternate_Function.Fields;
  Unused : Unaccessed_Gap with Unreferenced;
end record with
  Size => 16#400# * 8;

for GPIO_Port use record
  MODER at 0 range 0 .. 31;
  OTYPER at 4 range 0 .. 15;
  Reserved_1 at 6 range 0 .. 15;
  OSPEEDR at 8 range 0 .. 31;
  PUPDR at 12 range 0 .. 31;
  IDR at 16 range 0 .. 15;
  Reserved_2 at 18 range 0 .. 15;
  ODR at 20 range 0 .. 15;
  Reserved_3 at 22 range 0 .. 15;
  BSRR_Set at 24 range 0 .. 15;
  BSRR_Reset at 26 range 0 .. 15;
  LCKR at 28 range 0 .. 31;
  AFR at 32 range 0 .. 63;
  Unused at 40 range 0 .. 7871;
end record;
```
The type for the gap, Unaccessed_Gap, must represent 984 bytes so we declared an array like so:

```ada
Gap_Size : constant := 984; -- bytes
-- There is a gap of unused, reserved memory after the end of the
-- bytes used by any given memory-mapped GPIO port. The size of the
-- gap is indicated in the STM32F405xx etc. Reference Manual, RM 0090.
-- Specifically, Table 1 shows the starting and ending addresses mapped
-- to the GPIO ports, for an allocated size of 16#400#, or 1024 (decimal)
-- bytes per port. However, in the same document, the register map for
-- these ports shows only 40 bytes currently in use. Presumably this gap is
-- for future expansion when additional functionality or capacity is added,
-- such as more pins per port.

type Unaccessed_Gap is array (1 .. Gap_Size) of Unsigned_8 with
  Component_Size => Unsigned_8'Size,
  Size => Gap_Size * Unsigned_8'Size;
-- This type is used to represent the necessary gaps between GPIO
-- ports in memory. We explicitly allocate a record component of
-- this type at the end of the record type for that purpose.
```

We also set the size of the entire record type to 16#400# bytes since that is the total of the required bytes plus the gap, as per the documentation. As such, this is a "confirming" size clause because the reserved gap component increases the required size to that value (which is the point). We don't really need to do both, i.e., declare the reserved gap component and also set the record type size to the larger value. We could have done either one alone. One could argue that setting the size alone would have been simpler, in that it would obviate the type declaration and corresponding record component declaration. Being doubly explicit seemed a good idea at the time.

### 37.3 Dynamic Address Conversion

In the overlay example there were two distinct Ada objects, of two different types, sharing one (starting) address. The overlay provides two views of the memory at that address because there are two types involved. In this idiom the address is known when the code is written, either because it is a literal value specified in some hardware spec, or it is simply the address of the other object (in which case the actual address value is neither known nor relevant).

When there are several views required, declaring multiple overlaid variables at the same address absolutely can work, but can be less convenient than an alternative idiom. The alternative is to convert an address value to a value of an access type. Dereferencing the resulting access value provides a view of the memory corresponding to the designated type, starting at the converted address value.

For example, perhaps a networking component is given a buffer — an array of bytes — representing a received message. A subprogram is called with the buffer as a parameter, or the parameter can be the address of the buffer. If the subprogram must interpret this array via different views, this alternative approach works well. We could have an access type designating a message preamble, for example, and convert the first byte's address into such an access value. Dereferencing the conversion gives the preamble value. Likewise, the subprogram might need to compute a checksum over some of the bytes, so a different view, one of an array of a certain set size, could be used. Again, we could do that with overlaid objects but the alternative can be more convenient.

Here's a simple concrete example to illustrate the approach. Suppose we want to have a utility to swap the two bytes at any arbitrary address. Here's the declaration:
procedure Swap2 (Location : System.Address);

Callers pass the address of an object intended to have its (first) two bytes swapped:

Swap2 (Z'Address);

In the call, Z is of type Interfaces.Integer_16, for example, or Unsigned_16, or even something bigger as long as you only care about swapping the first two bytes.

The incomplete implementation using the conversion idiom could be like so:

```ada
procedure Swap2 (Location : System.Address) is
  X : Word renames To_Pointer (Location).all;
begind
  X := Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;
```

The declaration of X is the pertinent part.

In the declaration, X is of type Word, a type (not yet shown) derived from Interfaces.Unsigned_16. Hence X can have the inherited shift and logical or operations applied.

The To_Pointer (Location) part of the declaration is a function call. The function returns the conversion of the incoming address value in Location into an access value designating Word values. We'll explain how to do that momentarily. The .all explicitly dereferences the access value resulting from the function call.

Finally, X renames the Word designated by the converted access value. The benefit of the renaming, in addition to the simpler name, is that the function is only called once, and the access value dereference is only evaluated once.

Now for the rest of the implementation not shown earlier.

```ada
type Word is new Interfaces.Unsigned_16;

package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;
```

System.Address_To_Access_Conversions is a language-defined generic package that provides just two functions: one to convert an address value to an access type, and one to convert in the opposite direction:

```ada
generic
type Object (<>) is limited private;
package System.Address_To_Access_Conversions is
  type Object_Pointer is access all Object;

  function To_Pointer (Value : Address) return Object_Pointer;
  function To_Address (Value : Object_Pointer) return Address;

  pragma Convention (Intrinsic, To_Pointer);
  pragma Convention (Intrinsic, To_Address);
end System.Address_To_Access_Conversions;
```

Object is the generic formal type parameter, i.e., the type we want our converted addresses to designate via the type Object_Pointer. In the byte-swapping example, the type Word was passed to Object in the instantiation.

The access type used by the functions is Object_Pointer, declared along with the functions. Object_Pointer designates values of the type used for the generic actual parameter, in this case Word.
Learning Ada

Note the pragma Convention applied to each function, indicating that there is no actual function call involved; the compiler emits the code directly, if any code is actually required. Otherwise the compiler just treats the incoming Address bits as a value of type Object_Pointer.

The instantiation specifies type Word as the generic actual type parameter, so now we have a set of functions for that type, in particular To_Pointer.

Let's look at the code again, this time with the additional declarations:

```ada
type Word is new Interfaces.Unsigned_16;
package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;

procedure Swap2 (Location : System.Address) is
   X : Word renames To_Pointer(Location).all;
begin
   X := Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;
```

Word_Ops is the generic instance, followed immediately by a `use` clause so that we can refer to the visible content of the package instance conveniently.

In the renaming expression, To_Pointer(Location) converts the incoming address in Location to a pointer designating the Word at that address. The .all dereferences the resulting access value to get the designated Word value. Hence X refers to that two-byte value in memory.

We could almost certainly achieve the same affect by replacing the call to the function in To_Pointer with a call to an instance of Ada.Unchecked_Conversion. The conversion would still be between an access type and a value of type System.Address, but the access type would require declaration by the user. In both cases there would be an instantiation of a language-defined facility, so there's not much saving in lines of source code, other than the access type declaration. Because System.Address_To_Access_Conversions is explicitly intended for this purpose, good style suggests its use in preference to unchecked conversion, but both approaches are common in production code.

In either case, the conversion is not required to work, although in practice it will, most of the time. Representing an access value as an address value is quite common because it matches the typical underlying hardware's memory model. But even so, a single address is not necessarily sufficient to represent an access value for any given designated type. In that case problems arise, and they are difficult to debug.

For example, in GNAT, access values designating values of unconstrained array types, such as String, are represented as two addresses, known as "fat pointers". One address points to the bounds for the specific array object, since they can vary. The other address designates the characters. Therefore, conversions of a single address to an access value requiring fat pointers will not work using unchecked conversions. (There is a way, however, to tell GNAT to use a single address value, but it is an explicit step in the code. Once done, though, unchecked conversions would then work correctly.)

You can alternatively use generic package System.Address_To_Access_Conversions. That generic is defined for the purpose of converting addresses to access values, and vice versa. But note that the implementation of the generic's routines must account for the representation their compiler uses for unbounded types like String.
37.4 Address Arithmetic

Part of "letting the compiler do the work for you" is not doing address arithmetic in the source code if you can avoid it. Instead, for instance, use the normal "dot notation" to reference components, and let the compiler compute the offsets to those components. The approach to implementing procedure ConfigureAlternateFunction for a GPIO_Port is a good example.

That said, sometimes address arithmetic is the most direct expression of what you’re trying to implement. For example, when implementing your own memory allocator, you’ll need to do address arithmetic.

Earlier in this section we mentioned the package System.Storage_Elements, for the sake of the function that converts integer values to address values. The package also defines functions that provide address arithmetic. These functions work in terms of type System.Address and the package-defined type Storage_Offset. The type Storage_Offset is an integer type with an implementation-defined range. As a result you can have positive and negative offsets, as needed. Addition and subtraction of offsets to/from addresses is supported, as well as the mod operator.

Combined with package System (for type System.Address), the functions and types in this package provide the kinds of address arithmetic other languages provide. Nevertheless, you should prefer having the compiler do these computations for you, if possible.

Here’s an example illustrating the facilities. The procedure defines an array of record values, then traverses the array, printing the array components as it goes. (This is not the way to really implement such code. It’s just an illustration for address arithmetic.)

```ada
with Ada.Text_IO; use Ada.Text_IO;
with System.Storage_Elements; use System.Storage_Elements;
with System.Address_To_Access_Conversions;

procedure Demo_Address_Arithmetic is

    type R is record
        X : Integer;
        Y : Integer;
    end record;

    R_Size : constant Storage_Offset := R'Object_Size / System.Storage_Unit;

    Objects : aliased array (1 .. 10) of aliased R;  -- arbitrary bounds
    Objects_Base : constant System.Address := Objects'Address;

    Offset : Storage_Offset;

    -- display the object of type R at the address specified by Location
    procedure Display_R (Location : in System.Address) is

        package R_Pointers is new System.Address_To_Access_Conversions (R);
        use R_Pointers;

        Value : R renames To_Pointer (Location).all;

        -- The above converts the address to a pointer designating an R value
        -- and dereferences it, using the name Value to refer to the
        -- dereferenced R value.
        begin
            Put (Integer'Image (Value.X));
            Put (", ");
            Put (Integer'Image (Value.Y));
        end;

    end="}
```
                    New_Line;
                    end Display_R;

begin
    Objects := ((0,0), (1,1), (2,2), (3,3), (4,4),
                   (5,5), (6,6), (7,7), (8,8), (9,9));

    Offset := 0;

    -- walk the array of R objects, displaying each one individually by
    -- adding the offset to the base address of the array
    for K in Objects'Range loop
        Display_R (Objects_Base + Offset);
        Offset := Offset + R_Size;
    end loop;

end Demo_Address_Arithmetic;

Seriously, this is just for the purpose of illustration. It would be much better to just index into the array directly.
GENERAL-PURPOSE CODE GENERATORS

In another chapter (page 1153), we mentioned that the best way to get a specific set of machine instructions emitted from the compiler is to write them ourselves, in the Ada source code, using machine-code insertions (MCI). The rationale was that the code generator will make reasonable assumptions, including the assumption that performance is of uppermost importance, but that these assumptions can conflict with device requirements.

For example, the code generator might not issue the specific sequence of machine code instructions required by the hardware. The GPIO pin "lock" sequence in that referenced chapter is a good example. Similarly, the optimizer might remove what would otherwise be "redundant" read/writes to a memory-mapped variable.

The code generator might issue instructions to read a small field in a memory-mapped record object using byte-sized accesses, when instead the device requires whole-word or half-word access instructions.

The code generator might decide to load a variable from memory into a register, accessing the register when the value is required. Typically that approach will yield far better performance than going to memory every time the value is read or updated. But suppose the variable is for a memory-mapped device? In that case we really need the generated code to go to memory every time.

As you can see, there are times when we cannot let the code generator make the usual assumptions. Therefore, Ada provides aspects and pragmas that developers can use to inform the compiler of facts that affect code generation in this regard.

These facilities are defined in the Systems Programming Annex, C.6, specifically. The title of that sub-clause is "Shared Variables" because the objects (memory) can be shared between tasks as well as between hardware devices and the host computer. We ignore the context of variables shared between tasks, focusing instead of shared memory-mapped devices, as this course is about embedded systems.

When describing these facilities we will use aspects, but remember that the corresponding pragmas are defined as well, except for one. (We'll mention it later.) For the other aspects, the pragmas existed first and, although obsolescent, remain part of the language and supported. There's no need to change your existing source code using the pragmas to use the aspects instead, unless you need to change it for some other reason.

As this is an introduction, we will not go into absolutely all the details, but will instead give a sense of what the language provides, and why.
Learning Ada

38.1 Aspect Independent

To interface with a memory-mapped device, there will be an Ada object of an appropriate type that is mapped to one or more bytes of memory. The software interacts with the device by reading and/or writing to the memory locations mapped to the device, using the operations defined by the type in terms of normal Ada semantics.

Some memory-mapped devices can be directly represented by a single scalar value, usually of some signed or unsigned numeric type. More sophisticated devices almost always involve several distinct input and output fields. Therefore, representation in the software as a record object is very common. Ada record types have such extensive and flexible support for controlling their representation, down to the individual bit level, that using a record type makes sense. (And as mentioned, using normal record component access via the "dot notation" offloads to the compiler the address arithmetic needed to access individual memory locations mapped to the device.) And of course the components of the mapped record type can themselves be of scalar and composite types too, so an extensive descriptive capability exists with Ada.

Let's say that one of these record components is smaller than the size of the smallest addressable memory unit on the machine, which is to say, smaller than the machine instructions can read/write memory individually. A Boolean record component is a good example, and very common. The machine cannot usually read/write single bits in memory, so the generated code will almost certainly read or write a byte to get the enclosed single-bit Boolean component. It might use a larger sized access too, a half-word or word. Then the generated code masks off the bits that are not of interest and does some shifts to get the desired component.

Reading and writing the bytes surrounding the component accessed in the source code can cause a problem. In particular, some devices react to being read or written by doing something physical in the hardware. That's the device designer's intent for the software. But we don't want that to happen accidentally due to surrounding bytes being accessed.

Therefore, to prevent these "extra" bytes from being accessed, we need a way to tell the compiler that we need the read or write accesses for the given object to be independent of the surrounding memory. If the compiler cannot do so, we'll get an error and the compilation will fail. That beats debugging, every time.

Therefore, the aspect Independent specifies that the code generated by the compiler must be able to load and store the memory for the specified object without also accessing surrounding memory. More completely, it declares that a type, object, or component must be independently addressable by the hardware. If applied to a type, it applies to all objects of the type.

Likewise, aspect Independent_Components declares that the individual components of an array or record type must be independently addressable.

With either aspect the compiler will reject the declaration if independent access is not possible for the type/object in question.

For example, if we try to mark each Boolean component of a record type as Independent we can do so, either individually or via Independent_Components, but doing so will require that each component is a byte in size (or whatever the smallest addressable unit happens to be on this machine). We cannot make each Boolean component occupy one bit within a given byte if we want them to be independently accessed.

```ada
package P is
    type R is record
        B0 : Boolean;  
        B1 : Boolean;  
        B2 : Boolean;

    end R;
```

(continues on next page)
Learning Ada

B3 : Boolean;
B4 : Boolean;
B5 : Boolean;
end record with
  Size => 8,
  Independent_Components;

for R use record
  B0 at 0 range 0 .. 0;
  B1 at 0 range 1 .. 1;
  B2 at 0 range 2 .. 2;
  B3 at 0 range 3 .. 3;
  B4 at 0 range 4 .. 4;
  B5 at 0 range 5 .. 5;
end record;
end P;

For a typical target machine the compiler will reject that code, complaining that the Size for R' must be at least 48 bits, i.e., 8 bits per component. That's because the smallest quantity this machine can independently address is an 8-bit byte.

But if we don't really need the individual bits to be independently accessed — and let's hope no hardware designer would define such a device — then we have more flexibility. We could, for example, require that objects of the entire record type be independently accessible:

package Q is
  type R is record
    B0 : Boolean;
    B1 : Boolean;
    B2 : Boolean;
    B3 : Boolean;
    B4 : Boolean;
    B5 : Boolean;
  end record with
    Size => 8,
    Independent;
  for R use record
    B0 at 0 range 0 .. 0;
    B1 at 0 range 1 .. 1;
    B2 at 0 range 2 .. 2;
    B3 at 0 range 3 .. 3;
    B4 at 0 range 4 .. 4;
    B5 at 0 range 5 .. 5;
  end record;
end Q;

This the compiler should accept, assuming a machine that can access bytes in memory individually, without having to read some number of other bytes.

But for another twist, suppose we need one of the components to be aliased, so that we can construct access values designating it via the Access attribute? For example, given the record type R above, and some object Foo of that type, suppose we want to say Foo. B0'Access? We'd need to mark the component as aliased:

package QQ is
  type R is record

(continues on next page)
The compiler will once again reject the code, complaining that the size of \( B0 \) must be a multiple of a Storage_Unit, in other words, the size of something independently accessible in memory on this machine.

Why? The issue here is that aliased objects, including components of composite types, must be represented in such a way that creating the designating access (“pointer”) value is possible. The component \( B0 \), if allocated only one bit, would not allow an access value to be created due to the usual machine accessibility limitation we’ve been discussing.

Similarly, a record component that is of some by-reference type, such as any tagged type, introduces the same issues as an aliased component. That's because the underlying implementation of by-reference parameter passing is much like a 'Access' attribute reference.

As important as the effect of this aspect is, you probably won't see it specified. There are other aspects that are more typically required. However, the semantics of Independent are part of the semantics of some of these other aspects. Applying them applies Independent too, in effect. So even though you don't typically apply it directly, you need to understand the independent access semantics. We discuss these other, more commonly applied aspects next.

These representation aspects may be specified for an object declaration, a component declaration, a full type declaration, or a generic formal (complete) type declaration. If any of these aspects are specified True for a type, then the corresponding aspect is True for all objects of the type.

### 38.2 Aspect Volatile

Earlier we said that the compiler (specifically the optimizer) might decide to load a variable from memory into a register, accessing the register when the value is required or updated. Similarly, the compiler might reorder instructions, and remove instructions corresponding to redundant assignments in the source code. Ordinarily we’d want those optimizations, but in the context of embedded memory-mapped devices they can be problematic.

The hardware might indeed require the source code to read or write to the device in a way that the optimizer would consider redundant, and in order to interact with the device we need every read and write to go to the actual memory for the mapped device, rather than a register. As developers we have knowledge about the context that the compiler lacks.
The compiler is aware of the fact that the Ada object is memory-mapped because of the address clause placing the object at a specific address. But the compiler does not know we are interacting with an external hardware device. Perhaps, instead, the object is mapped to a specific location because some software written in another language expects to access it there. In that case redundant reads or writes of the same object really would be redundant. The fact that we are interacting with a hardware device makes a difference.

In terms of the language rules, we need reading from, and writing to, such devices to be part of what the language refers to as the "external effects" of the software. These effects are what the code must actually produce. Anything else — the internal effects — could be removed by the optimizer.

For example, suppose you have a program that writes a value to some variable and also writes the string literal "42" to a file. That's absolutely all that the program contains.

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is
    Output : File_Type;
    Silly  : Integer;
begin
    Silly := 0;
    Create (Output, Out_File, "output.txt");
    Put (Output, "42");
    Close (Output);
end Demo;
```

The value of the variable Silly is not used in any way so there is no point in even declaring the variable, much less generating code to implement the assignment. The update to the variable has only an internal effect. With warnings enabled we'll receive notice from the compiler, but they're just warnings.

However, writing to the file is an external effect because the file persists beyond the end of the program's execution. The optimizer (when enabled) would be free to remove any access to the variable Silly, but not the write to the file.

We can make the compiler recognize that a software object is part of an external effect by applying the aspect `Volatile`. (Aspect `Atomic` is pertinent too. More in a moment.) As a result, the compiler will generate memory load or store instructions for every read or update to the object that occurs in the source code. Furthermore, it cannot generate any additional loads or stores to that variable, and it cannot reorder loads or stores from their order in the source code. "What You See Is What You Get" in other words.

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is
    Output : File_Type;
    Silly  : Integer with Volatile;
begin
    Silly := 0;
    Create (Output, Out_File, "output.txt");
    Put (Output, "42");
    Close (Output);
end Demo;
```

If we compile the above, we won't get the warning we got earlier because the compiler is now required to generate the assignment for Silly.

The variable Silly is not even a memory-mapped object, but remember that we said these aspects are important to the tasking context too, for shared variables. We're ignoring that context in this course.

There is another reason to mark a variable as `Volatile`. Sometimes you want to have...
Learning Ada

exactly the load and store instructions generated that match those of the Ada code, even though the volatile object is not a memory-mapped object. For example, elsewhere (page 1153) we said that the best way to achieve exact assembly instruction sequences is the use of machine-code inserts (MCI). That's true, but for the moment let's say we want to write it in Ada without the MCIs. Our earlier example was the memory-mapped GPIO ports on Arm microcontrollers produced by ST Microelectronics. Specifically, these ports have a "lock" per GPIO pin that allows the developer to configure the pin and then lock it so that no other configuration can accidentally change the configuration of that pin. Doing so requires an exact sequence of loads and stores. If we wrote this in Ada it would look like this:

```ada
procedure Lock
(Port : in out GPIO_Port;
 Pin : GPIO_Pin)
is
 Temp : Word with Volatile;
begin
  -- set the lock control bit and the pin
  -- bit, clear the others
  Temp := LCCK or Pin'Enum_Rep;

  -- write the lock and pin bits
  Port.LCKR := Temp;

  -- clear the lock bit in the upper half
  Port.LCKR := Pin'Enum_Rep;

  -- write the lock bit again
  Port.LCKR := Temp;

  -- read the lock bit
  Temp := Port.LCKR;

  -- read the lock bit again
  Temp := Port.LCKR;
end Lock;
```

Temp is marked volatile for the sake of getting exactly the load and stores that we express in the source code, corresponding to the hardware locking protocol. It's true that Port is a memory-mapped object, so it too would be volatile, but we also need Temp to be volatile.

This high-level coding approach will work, and is simple enough that MCIs might not be needed. However, what really argues against it is that the correct sequence of emitted code requires the optimizer to remove all the other cruft that the code generator would otherwise include. (The gcc code generator used by the GNAT compiler generates initially poor code, by design, relying on the optimizer to clean it up.) In other words, we've told the optimizer not to change or add loads and stores for Temp, but without the optimizer enabled the code generator generates other code that gets in the way. That's OK in itself, as far as procedure Lock is concerned, but if the optimizer is sufficiently enabled we cannot debug the rest of the code. Using MCIs avoids these issues. The point, though, is that not all volatile objects are memory mapped.

So far we've been illustrating volatility with scalar objects, such as Lock.Temp above. What about objects of array and record types? (There are other "composite" types in Ada but they are not pertinent here.)

When aspect Volatile is applied to a record type or an object of such a type, all the record components are automatically volatile too.

For an array type (but not a record type), a related aspect Volatile_Components declares that the components of the array type — but not the array type itself — are volatile. However, if the Volatile aspect is specified, then the Volatile_Components aspect is automatically applied too, and vice versa. Thus components of array types are covered auto-
If an object (of an array type or record type) is marked volatile then so are all of its sub-components, even if the type itself is not marked volatile.

Therefore aspects Volatile and Volatile_Components are nearly equivalent. In fact, Volatile_Components is superfluous. The language provides the Volatile_Components aspect only to give symmetry with the Atomic_Components and Independent_Components aspects. You can simply apply Volatile and be done with it.

Finally, note that applying aspect Volatile does not implicitly apply Independent, although you can specify it explicitly if need be.

38.3 Aspect Atomic

Consider the GPIO pin configuration lock we’ve mentioned a few times now, that freezes the configuration of a given pin on a given GPIO port. The register, named LCKR for "lock register", occupies 32-bits, but only uses 17 total bits (currently). The low-order 16 bits, [0:15], represent the 16 GPIO pins on the given port. Bit #16 is the lock bit. That bit is the first bit in the upper half of the entire word. To freeze the configuration of a given pin in [0:15], the lock bit must be set at the same time as the bit to be frozen. In other words, the lower half and the upper half of the 32-bit word representing the register must be written together, at the same time. That way, accidental (un)frieezing is unlikely to occur, because the most efficient, hence typical way for the generated code to access individual bits is for the compiler to load or store just the single byte that contains the bit or bits in question.

This indivisibility effect can be specified via aspect Atomic. As a result, all reads and updates of such an object as a whole are indivisible. In practice that means that the entire object is accessed with one load or store instruction. For a 16-bit object, all 16-bits are loaded and stored at once. For a 32-bit object, all 32-bits at once, and so on. The upper limit is the size of the largest machine scalar that the processor can manipulate with one instruction, as defined by the target processor. The typical lower bound is 8, for a byte-addressable machine.

Therefore, within the record type representing a GPIO port, we include the lock register component and apply the aspect Atomic:

```ada
type GPIO_Port is limited record
  ... LCKR : UInt32 with Atomic;
  ... end record with
  ... Size => 16#400# * 8;
```

Hence loads and stores to the LCKR component will be done atomically, otherwise the compiler will let us know that it is impossible. That's all we need to do for the lock register to be read and updated atomically.

You should understand that only accesses to the whole, entire object are atomic. In the case of the lock register, the entire object is a record component, but that causes no problems here.

There is, however, something we must keep in mind when manipulating the values of atomic objects. For the lock register we’re using a scalar type to represent the register, an unsigned 32-bit integer. There are no sub-components because scalar types don’t have components, by definition. We simply use the bit-level operations to set and clear the individual bits. But we cannot set the bits — the lock bit and the bit for the I/O pin to freeze — one at a time because the locking protocol requires all the bits to be written at the same time, and only the entire 32-bit load and stores are atomic. Likewise, if instead of a scalar we used
Learning Ada

a record type or an array type to represent the bits in the lock register, we could not write individual record or array components one at a time, for the same reason we could not write individual bits using the unsigned scalar. The Atomic aspect only applies to loads and stores of the entire register.

Therefore, to update or read individual parts of an atomic object we must use a coding idiom in which we explicitly read or write the entire object to get to the parts. For example, to read an individual record component, we'd first read the entire record object into a temporary variable, and then access the component of that temporary variable. Likewise, to update one or more individual components, we'd first read the record object into a temporary variable, update the component or components within that temporary, and then write the temporary back to the mapped device object. This is known as the "read-modify-write" idiom. You'll see this idiom often, regardless of the programming language, because the hardware requirement is not unusual. Fortunately Ada defines another aspect that makes the compiler do this for us. We'll describe it in the next section.

Finally, there are issues to consider regarding the other aspects described in this section.

If you think about atomic behavior in the context of machine instructions, loading and storing from/to memory atomically can only be performed for quantities that are independently addressable. Consequently, all atomic objects are considered to be specified as independently addressable too. Aspect specifications and representation items cannot change that fact. You can expect the compiler to reject any aspect or representation choice that would prevent this from being true.

Likewise, atomic accesses only make sense on actual memory locations, not registers. Therefore all atomic objects are volatile objects too, automatically.

However, unlike volatile objects, the components of an atomic object are not automatically atomic themselves. You'd have to mark these types or objects explicitly, using aspect Atomic_Components. Unlike Volatile_Components, aspect Atomic_Components is thus useful.

As is usual with Ada programming, you can rely on the compiler to inform you of problems. The compiler will reject an attempt to specify Atomic or Atomic_Components for an object or type if the implementation cannot support the indivisible and independent reads and updates required.

**38.4 Aspect Full_Access_Only**

Many devices have single-bit flags in the hardware that are not allocated to distinct bytes. They're packed into bytes and words shared with other flags. It isn't just individual bits either. Multi-bit fields that are smaller than a byte, e.g., two 4-bit quantities packed into a byte, are common. We saw that with the GPIO alternate functions codes earlier.

Ordinarily in Ada we represent such composite hardware interfaces using a record type. (Sometimes an array type makes more sense. That doesn't change anything here.) Compared to using bit-patterns, and the resulting bit shifting and masking in the source code, a record type representation and the resulting "dot notation" for accessing components is far more readable. It is also more robust because the compiler does all the work of retrieving these individual bits and bit-fields for us, doing any shifting and masking required in the generated code. The loads and stores are done by the compiler in whatever manner the compiler thinks most efficient.

When the hardware device requires atomic accesses to the memory mapped to such flags, we cannot let the compiler generate whatever width load and store accesses it thinks best. If full-word access is required, for example, then only loads and stores for full words can work. Yet aspect Atomic only guarantees that the entire object, in this case the record object, is loaded and stored indivisibly, via one instruction. The aspect doesn't apply to reads and updates to individual record components.
In the section on Atomic above, we mentioned that proper access to individual components of atomic types/objects can be achieved by a "read-modify-write" idiom. In this idiom, to read a component you first read into a temporary the entire enclosing atomic object. Then you read the individual component from that temporary variable. Likewise, to update an individual component, you start with the same approach but then update the component(s) within the temporary, then store the entire temporary back into the mapped atomic object. Applying aspect Atomic to the enclosing object ensures that reading and writing the temporary will be atomic, as required.

Using bit masks and bit patterns to access logical components as an alternative to a record type doesn't change the requirement for the idiom.

Consider the STM32F4 DMA device. The device contains a 32-bit stream configuration register that requires 32-bit reads and writes. We can map that register to an Ada record type like so:

```ada
type Stream_Config_Register is record
  -- ...
  Direction : DMA_Data_Transfer_Direction;
  P_Flow_Controller : Boolean;
  TCI_Enabled : Boolean; -- transfer complete
  HTI_Enabled : Boolean; -- half-transfer complete
  TEI_Enabled : Boolean; -- transfer error
  DMEI_Enabled : Boolean; -- direct mode error
  Stream_Enabled : Boolean;
end record
with Atomic, Size => 32;
```

The "confirming" size clause ensures we have declared the type correctly such that it will fit into 32-bits. There will also be a record representation clause to ensure the record components are located internally as required by the hardware. We don't show that part.

The aspect Atomic is applied to the entire record type, ensuring that the memory mapped to the hardware register is loaded and stored only as 32-bit quantities. In this example it isn't that we want the loads and stores to be indivisible. Rather, we want the generated machine instructions that load and store the object to use 32-bit word instructions, even if we are only reading or updating a component of the object. That's what the hardware requires for all accesses.

Next we'd use that type declaration to declare one of the components of an enclosing record type representing one entire DMA "stream":

```ada
type DMA_Stream is record
  CR : Stream_Config_Register;
  NDTR : Word; -- upper half must remain at reset value
  PAR : Address; -- peripheral address register
  M0AR : Address; -- memory 0 address register
  M1AR : Address; -- memory 1 address register
  FCR : FIFO_Control_Register;
end record
with Volatile, Size => 192; -- 24 bytes
```

Hence any individual DMA stream record object has a component named CR that represents the corresponding configuration register.

The DMA controllers have multiple streams per unit so we'd declare an array of DMA_Stream components. This array would then be part of another record type representing a DMA controller. Objects of the DMA_Controller type would be mapped to memory, thus mapping the stream configuration registers to memory.

Now, given all that, suppose we want to enable a stream on a given DMA controller. Using the read-modify-write idiom we would do it like so:
procedure Enable
   (Unit : in out DMA_Controller;
    Stream : DMA_Stream_Selector)
is
   Temp : Stream_Config_Register;
   -- these registers require 32-bit accesses, hence the temporary
begin
   Temp := Unit.Streams(Stream).CR;  -- read entire CR register
   Temp.Stream_Enabled := True;
   Unit.Streams(Stream).CR := Temp;  -- write entire CR register
end Enable;

That works, and of course the procedural interface presented to clients hides the details, as it should.

To be fair, the bit-pattern approach can express the idiom concisely, as long as you're careful. Here's the C code to enable and disable a selected stream:

```c
#define DMA_SxCR_EN ((uint32_t)0x00000001)

DMAy_Streamx->CR = DMA_SxCR_EN;

DMAy_Streamx->CR &= ~DMA_SxCR_EN;
```

The code reads and writes the entire CR register each time it is referenced so the requirement is met.

Nevertheless, the idiom is error-prone. We might forget to use it at all, or we might get it wrong in one of the very many places where we need to access individual components.

Fortunately, Ada provides a way to have the compiler implement the idiom for us, in the generated code. Aspect Full_Access_Only specifies that all reads of, or writes to, a component are performed by reading and/or writing all of the nearest enclosing full access object. Hence we add this aspect to the declaration of Stream_Config_Register like so:

```ada
type Stream_Config_Register is record
   Direction : DMA_Data_Transfer_Direction;
   P_Flow_Controller : Boolean;
   TCI_Enabled : Boolean;  -- transfer complete interrupt
   HTI_Enabled : Boolean;  -- half-transfer complete
   TEI_Enabled : Boolean;  -- transfer error interrupt
   DMEI_Enabled : Boolean;  -- direct mode error interrupt
   Stream_Enabled : Boolean;
end record
with Atomic, Full_Access_Only, Size => 32;
```

Everything else in the declaration remains unchanged.

Note that Full_Access_Only can only be applied to Volatile types or objects. Atomic types are automatically Volatile too, so either one is allowed. You'd need one of those aspects anyway because Full_Access_Only just specifies the accessing instruction requirements for the generated code when accessing components.

The big benefit comes in the source code accessing the components. Procedure Enable is now merely:

```ada
procedure Enable
   (Unit : in out DMA_Controller;
    Stream : DMA_Stream_Selector)
is
   (continues on next page)
```
This code works because the compiler implements the read-modify-write idiom for us in the generated code.

The aspect `Full_Access_Only` is new in Ada 2022, and is based on an implementation-defined aspect that GNAT first defined named `Volatile_Full_Access`. You'll see that GNAT aspect throughout the Arm device drivers in the Ada Drivers Library, available here: https://github.com/AdaCore/Ada_Drivers_Library. Those drivers were the motivation for the GNAT aspect.

Unlike the other aspects above, there is no pragma corresponding to the aspect `Full_Access_Only` defined by Ada 2022. (There is such a pragma for the GNAT-specific version named `Volatile_Full_Access`, as well as an aspect.)
CHAPTER
THIRTYNINE

HANDLING INTERRUPTS

39.1 Background

Embedded systems developers offload functionality from the application processor onto external devices whenever possible. These external devices may be on the same "chip" as the central processor (e.g., within a System-on-Chip) or they may just be on the same board, but the point here is that they are not the processor executing the application. Offloading work to these other devices enables us to get more functionality implemented in a target platform that is usually very limited in resources. If the processor has to implement everything we might miss deadlines or perhaps not fit into the available code space. And, of course, some specialized functionality may simply require an external device, such as a sensor.

For a simple example, a motor encoder is a device attached to a motor shaft that can be used to count the number of full or partial rotations that the shaft has completed. When the shaft is rotating quickly, the application would need to interact with the encoder frequently to get an up-to-date count, representing a non-trivial load on the application processor. There are ways to reduce that load, which we discuss shortly, but by far the simplest and most efficient approach is to do it all in hardware: use a timer device driven directly by the encoder. The timer is connected to the encoder such that the encoder signals act like an external clock driving the timer's internal counter. All the application processor must do to get the encoder count is query the timer's counter. The timer is almost certainly memory-mapped, so querying the timer amounts to a memory access.

In some cases, we even offload communication with these external devices onto other external devices. For example, the I2C (Inter-Integrated Circuit) protocol is a popular two-wire serial protocol for communicating between low-level hardware devices. Individual bits of the data are sent by driving the data line high and low in time with the clock signal on the other line. The protocol has been around for a long time and many embedded devices use it to communicate. We could have the application drive the data line for each individual bit in the protocol. Known as "bit-banging," that would be a significant load on the processor when the overall traffic volume is non-trivial. Fortunately, there are dedicated devices — I2C transceivers — that will implement the protocol for us. To send application data to another device using the I2C protocol, we just give the transceiver the data and destination address. The rest is done in the transceiver hardware. Receiving data is of course also possible. I2C transceivers are ubiquitous because the protocol is so common among device implementations. A USART / UART is a similar example.

Having offloaded some of the work, the application must have some way to interact with the device in order to know what is happening. Maybe the application has requested the external device perform some service — an analog-to-digital conversion, say — and must know when that function has completed. Maybe a communications device is receiving

---

incoming data for the application to process. Or maybe that communications device has completed sending outgoing data and is ready for more to send.

Ultimately, interaction with the external device will be either synchronous or asynchronous, and has system-level design implications.

For synchronous interaction, the application periodically queries the device, typically a status flag or function on the device. Known as "polling," this approach is simple to implement but wastes cycles when the external device has not yet completed the request. After all, the point of offloading the work is to allow the application processor to execute other functionality. Polling negates that benefit. On the other hand, if the expected time to completion is extremely short, polling can be sufficiently efficient to make sense.

Usually, there's enough time involved so that polling is undesirable. The external environment takes time to respond and change state. Maybe a sensor has been designed to wait passively for something to happen in the external world, and only on the infrequent occurrence of that event should the application be notified. Perhaps a switch is to be toggled in certain circumstances, or an intruder detected. In this case, nothing happens for extended intervals.

As a consequence of all this, there's a very good chance that the internal processor should not poll these external devices.

Before we discuss the asynchronous alternative, there's another issue to consider. However the notification from the external device is implemented, a very quick response from the internal processor may be required. Think back to that serial port with a USART again. The USART is responsible for composing the arriving characters (or bytes) from their individual incoming bits on the receiving line. When all the bits for a single character have arrived, what happens next depends on the software design. In the simplest case, the internal processor copies the single character from the USART to an internal buffer and then goes back to doing something else while the next full character arrives in the USART. The response to the USART must be fairly quick because the next incoming character's bits are arriving. The internal processor must get the current character before it is overwritten by the next arriving character, otherwise we'll lose data. So we can say that the response to the notification from the external device must often be very quick.

Now, ideally in the USART case, we would further offload the work from the internal processor. Instead of having the processor copy each arriving character from the USART into an application buffer, we would have another external hardware device — a direct memory access (DMA) device — copy each arriving character from the USART to the buffer. A DMA device copies data from one location to another, in this case from the address of the USART's one-character memory-mapped register to the address of the application buffer in memory. The copy is performed by the DMA hardware so it is extremely fast and costs the main processor no cycles. But even with this approach, we need to notify the application that a complete message is ready for processing. We might need to do that quickly so that enough time remains for the application to process the message content prior to the arrival of the next message.

Therefore, the general requirement is for an external device to be able to asynchronously notify the internal processor, and for the notification to be implemented in such a way that the beginning of the response can be sufficiently and predictably quick.

Fortunately, computers already have such a mechanism: interrupts. The details vary considerably with the hardware architecture, but the overall idea is independent of the ISA:

an external event can trigger a response from the processor by becoming "active." The current state of the application is temporarily stored, and then an interrupt response routine, known as an "interrupt handler" is executed. Upon completion of the handler, the original state of the application is restored and the application continues execution. The time between the interrupt becoming active and the start of the responding handler execution is known as the "interrupt latency."
Hardware interrupts typically have priorities assigned, depending on the hardware. These priorities are applied when multiple interrupts are triggered at the same time, to define the order in which the interrupts are presented and the handlers invoked. The canonical model is that only higher-priority interrupts can preempt handlers executing in response to interrupts with lower or equal priority.

Ada defines a model for hardware interrupts and interrupt handling that closely adheres to the conceptual model described above. If you have experience with interrupt handling, you will recognize them in the Ada model. One very important point to make about the Ada facilities is that they are highly portable, so they don't require extensive changes when moving to a new target computer. Part of that portability is due to the language-defined model.

Before we go into the Ada facility details, there's a final point. Sometimes we do want the application to wait for the external device. When would that be the case? To answer that, we need to introduce another term. The act of saving and restoring the state of the interrupted application software is known as "interrupt context switching." If the time for the device to complete the application request is approximately that of the context switching, the application might as well wait for the device after issuing the request.

Another reason to consider polling is that the architectural complexity of interrupt handling is greater than that of polling. If your system has some number of devices to control and polling them would be fast enough for the application to meet requirements, it is simpler to do so. But that will likely only work for a few devices, or at least a few that have short response time requirements.

The application code can wait for the device by simply entering a loop, exiting only when some external device status flag indicates completion of the function. The loop itself, in its simplest form, would contain only the test for exiting. As mentioned earlier, polling in a tight loop like this only makes sense for very fast device interactions. That's not the usual situation though, so polling should not be your default design assumption. Besides, active polling consumes power. On an embedded platform, conserving power is often important.

That loop polling the device will never exit if the device can fail to signal completion. Or maybe it might take too long in some odd case. If you don't want to be potentially stuck in the loop indefinitely, chewing up cycles and power, you can add an upper bound on the number of attempts, i.e., loop iterations. For example:

```ada
procedure Await_Data_Ready (This : in out Three_Axis_Gyroscope) is
  Max_Status_Attempts : constant := 10_000;
  -- This upper bound is arbitrary but must be sufficient for the
  -- slower gyro data rates and higher clock rates. It need
  -- not be as small as possible, the point is not to hang forever.
begin
  Polling: for K in 1 .. Max_Status_Attempts loop
    if Data_Status (This).ZYX_Available then
      return;
    end if;
  end loop Polling;
  raise Gyro_Failure;
end Await_Data_Ready;
```

In the above, Data_Status is a function that returns a record object containing Boolean flags. The if-statement queries one of those flags. Thus the loop either detects the desired device status or raises an exception after the maximum number of attempts have been made. In this version, the maximum is a known upper bound so a local constant will suffice. The maximum could be passed as a parameter instead, or declared in a global "configuration" package containing such constants.

Presumably, the upper bound on the attempts is either specified by the device documentation or empirically determined. Sometimes, however, the documentation will instead specify a maximum possible response time, for instance 30 milliseconds. Any time beyond
Learning Ada

that maximum indicates a device failure.

In the code above, the number of iterations indirectly defines the amount of elapsed time the caller waits. That time varies with the target's system clock and the generated instructions' required clock cycles, hence the approach is not portable. Alternatively, we can work in terms of actual time, which will be portable across all targets with a sufficiently precise clock.

You can use the facilities in package Ada.Real_Time to work with time values. That package defines a type Time_Span representing time intervals, useful for expressing relative values such as elapsed time. There is also type Time representing an absolute value on the timeline. A function Clock returns a value of type Time representing "now," along with overloaded addition and subtraction operators taking Time and Time_Span parameters. The package also provides operators for comparing Time values. (The value returned by Clock is monotonically increasing so you don't need to handle time zone jumps and other such things, unlike the function provided by Ada.Calendar.)

If the timeout is not context-specific then we'd use a constant as we did above, otherwise we'd allow the caller to specify the timeout. For example, here's a polling routine included with the DMA device driver we've mentioned a few times now. Some device-specific parts have been removed to keep the example simple. The appropriate timeout varies, so it is a parameter to the call:

```ada
procedure Poll_For_Completion
(This : in out DMA_Controller;
 Stream : DMA_Stream_Selector;
 Timeout : Time_Span;
 Result : out DMA_Error_Code)
is
  Deadline : constant Time := Clock + Timeout;
begin
  Result := DMA_No_Error; -- initially
  Polling : loop
    exit Polling when Status (This, Stream, Transfer_Complete_Indicated);
    if Clock <= Deadline then
      Result := DMA_Timeout_Error;
      return;
    end if;
  end loop Polling;
  Clear_Status (This, Stream, Transfer_Complete_Indicated);
end Poll_For_Completion;
```

In this approach, we compute the deadline as a point on the timeline by adding the value returned from the Clock function (i.e., "now") to the time interval specified by the parameter. Then, within the loop, we compare the value of the Clock to that deadline.

Finally, with another design approach we can reduce the processor cycles "wasted" when the polled device is not yet ready. Specifically, in the polling loop, when the device has not yet completed the requested function, we can temporarily relinquish the processor so that other tasks within the application can execute. That isn't perfect because we're still checking the device status even though we cannot exit the loop. And it requires other tasks to exist in your design, although that's probably a good idea for other reasons (e.g., logical threads having different, non-harmonic periods). This approach would look like this (an incomplete example):

```ada
procedure Poll_With_Delay is
  Next_Release : Time;
  Period : constant Time_Span := Milliseconds (30); -- let's say
begin
  Next_Release := Clock;
  loop
    exit when Status (...);
  end loop;
end Poll_With_Delay;
```

(continues on next page)
The code above will check the status of some device every 30 milliseconds (an arbitrary period just for illustration) until the Status function result allows the loop to exit. If the device "hangs" the loop is never exited, but as you saw there are ways to address that possibility. When the code does not exit the loop, the next point on the timeline is computed and the task executing the code then suspends, allowing the other tasks in the application to execute. Eventually, the next release point is reached and so the task becomes ready to execute again (and will, subject to priorities).

But how long should the polling task suspend when awaiting the device? We need to suspend long enough for the other tasks to get something done, but not so long that the device isn't handled fast enough. Finding the right balance is often not simple, and is further complicated by the "task switching" time. That's the time it takes to switch the execution context from one task to another, in this case in response to the "delay until" statement suspending the polling task. And it must be considered in both directions: when the delay expires we'll eventually switch back to the polling task.

As you can see, polling is easily expressed but has potentially significant drawbacks and architectural ramifications so it should be avoided as a default approach.

Now let's explore the Ada interrupt facilities.

### 39.2 Language-Defined Interrupt Model

The Ada language standard defines a model for hardware interrupts, as well as language-defined mechanisms for handling interrupts consistent with that model. The model is defined in Annex C, the "Systems Programming" annex, section 3 "Interrupt Support." The following is the text of that section with only a few simplifications and elisions.

- Interrupts are said to occur. An occurrence of an interrupt is separable into generation and delivery.
  - Generation of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program.
  - Delivery is the action that invokes part of the program as response to the interrupt occurrence.
- Between generation and delivery, the interrupt occurrence is pending.
- Some or all interrupts may be blocked. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered.
- Certain interrupts are reserved. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other RTL-defined means. The set of reserved interrupts is determined by the hardware and run-time library (RTL).
- Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be attached to that interrupt. The execution of that program unit, the interrupt handler, is invoked upon delivery of the interrupt occurrence.
- While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt.
- The corresponding interrupt is blocked while the handler executes. While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered.
Whether such occurrences remain pending or are lost is determined by the hardware and the RTL.

- Each interrupt has a default treatment which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is defined by the RTL.
- An exception propagated from a handler that is invoked by an interrupt has no effect. In particular, it is not propagated out of the handler, in the same way that exceptions do not propagate outside of task bodies.
- If the Ceiling_Locking policy is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object. ("Protected object" is abbreviated as "PO" for convenience).
- If the hardware or the underlying system holds pending interrupt occurrences, the RTL must provide for later delivery of these occurrences to the program.

(The above is not everything in the model but we can ignore the rest in this introduction.)

Because interrupt occurrences are generated by the hardware and delivered by the underlying system software (run-time library or real-time operating system), the application code is mainly responsible for responding to occurrences. Of course, the application must first configure the relevant external devices so that they generate the expected interrupts.

The actual response is application-specific but is also hardware-specific. The latter often (but not always) requires clearing the interrupt status within the generating device so that the same occurrence is not delivered again.

Furthermore, the standard model requires the underlying software to block further occurrences while the handler executes, and only allow preemption by higher-priority interrupt occurrences (if any). The application handlers are not responsible for these semantics either. As you will see, the choice of program unit used for expressing handlers makes this all very convenient for the developer.

As a consequence, in terms of the response, the application developer must write the specific handlers and attach those handlers to the corresponding interrupts. Attaching the handlers is implemented in the underlying system software, and it is this same underlying software that delivers the occurrences.

We will now explore the Ada facilities in detail. At the end of this chapter we will explore some common idioms using these mechanisms, especially with regard to the handlers' interaction with the rest of the application.

### 39.3 Interrupt Handlers

Interrupt handling is, by definition, asynchronous: some event occurs that causes the processor to suspend the application, respond to the event, and then resume application execution.

Because these events are asynchronous, the actions performed by the interrupt handler and the application are subject to the same sorts of race conditions as multiple tasks acting on shared data.

For example, a "reader" task may be in the act of reading (copying) the value of some shared variable, only to be preempted by a "writer" task that updates the value of the variable. In that case, when the "reader" task resumes execution, it will finish the read operation but will, as a result, have a value that is partly from the old value and partly from the new value. The effect is unpredictable. An interrupt handler can have the same effect on shared data as the preempting "writer" task that interrupts the "reader" task. This problem is possible for shared data of any type that is not atomically read or written. You can think of large record objects if that helps, but it even applies to some scalars.
That scenario applies even if no explicit tasks are declared in the application. That’s because an implicit "environment task" is executing the main subprogram. In that case, the main subprogram is the entire application, but more typically some non-null application code is actively executing in one or more tasks.

But it's not just a matter of tasks. We said that interrupts usually have priorities. Typically that means a higher-priority interrupt will preempt the execution of the handler for a lower-priority interrupt. It's the same issue.

Furthermore, the fact that an interrupt has occurred needs to be communicated to the application, for example to say that updated data are available, perhaps a sensor reading or characters from a serial port. As we said above, we usually don't want to poll for that fact, so the application must be able to suspend until the event has occurred. Often we'll have a dedicated task within the application that suspends, rather than the entire application, but that's an application detail.

Ada's protected objects address all these asynchronous issues. Shared data declared within a protected object can be accessed only via protected procedures or protected entries, both of which execute with mutually exclusive access. Hence no race conditions are possible.

Here is an extremely simple, but realistic, example of a PO. This is not an interrupt handler example — we'll get to that — but it does show a shared variable and a protected procedure that executes with mutually exclusive access no matter how many tasks concurrently call it. The PO provides unique serial numbers.

```
protected Serial_Number is
  procedure Get_Next (Number : out Positive);
private
  Value : Positive := 1;
end Serial_Number;

protected body Serial_Number is
  procedure Get_Next (Number : out Positive) is
  begin
    Number := Value;
    Value := Value + 1;
  end Get_Next;
end Serial_Number;
```

Imagine there are multiple assembly lines creating devices of various sorts. Each device gets a unique serial number. These assembly lines run concurrently, so the calls to Get_Next occur concurrently. Without mutually exclusive access to the Value variable, multiple devices could get the same serial number.

Protected entries can suspend a caller until some condition is true; in this case, the fact that an interrupt has occurred and been handled. (As we will see, a protected entry is not the only way to synchronize with an accessing task, but it is the most robust and general.)

Here's an example of a PO with a protected entry:

```
protected type Persistent_Signal is
  entry Wait;
  procedure Send;
private
  Signal_Arrived : Boolean := False;
end Persistent_Signal;

protected body Persistent_Signal is
```

(continues on next page)
This is a PO providing a "Persistent Signal" abstraction. It allows a task to wait for a "signal" from another task. The signal is not lost if the receiving task is not already waiting, hence the term "persistent." Specifically, if \( \text{Signal\_Arrived} \) is \( \text{False} \), a caller to \text{Wait} will be suspended until \( \text{Signal\_Arrived} \) becomes \( \text{True} \). A caller to \text{Send} sets \( \text{Signal\_Arrived} \) to \( \text{True} \), and will be allowed to continue. In either case, the \( \text{Signal\_Arrived} \) flag will be set back to \( \text{False} \) before the \text{Wait} caller is released. Protected objects can have a priority assigned, similar to tasks, so they are integrated into the global priority semantics including interrupt priorities.

Therefore, in Ada an interrupt handler is a protected procedure declared within some protected object (PO). A given PO may handle more than one interrupt, and if so, may use one or more protected procedures to do so.

Interrupts can be attached to a protected procedure handler using a mechanism we'll discuss shortly. When the corresponding interrupt occurs, the attached handler is invoked. Any exceptions propagated by the handler's execution are ignored and do not go past the procedure.

While the protected procedure handler executes, the corresponding interrupt is blocked. As a consequence, another occurrence of that same interrupt will not preempt the handler's execution. However, if the hardware does not allow interrupts to be blocked, no blocking occurs and a subsequent occurrence would preempt the current execution of the handler. In that case, your handlers must be written with that possibility in mind. Most targets do block interrupts so we will assume that behavior in the following descriptions.

The standard mutually exclusive access provided to the execution of protected procedures and entries is enforced whether the "call" originates in hardware, via an interrupt, or in the application software, via some task. While any protected action in the PO executes, the corresponding interrupt is blocked, such that another occurrence will not preempt the execution of that actions' procedure or entry body execution in the PO.

On some processors blocked interrupts are lost, they do not persist. However, if the hardware can deliver an interrupt that had been blocked, the Systems Programming Annex requires the handler to be invoked again later, subject to the PO semantics described above.

The default treatment for a given interrupt depends on the RTL implementation. The default may be to jump immediately to system-defined handler that merely loops forever, thereby "hanging" the system and preventing any further execution of the application. On a bare-board target that would be a very common approach. Alternatively the default could be to ignore the interrupt entirely.

As mentioned earlier, some interrupts may be reserved, meaning that the application cannot install a replacement handler. For instance, most bare-board systems include a clock that is driven by a dedicated interrupt. The application cannot (or at least should not) override the interrupt handler for that interrupt. The determination of which interrupts are reserved is RTL-defined. Attempting to attach a user-defined handler for a reserved interrupt raises \text{Program\_Error}, and the existing treatment is unchanged.
39.4 Interrupt Management

Ada defines a standard package that provides a primary type for identifying individual interrupts, as well as subprograms that take a parameter of that type in order to manage the system's interrupts and handlers. The package is named Ada.Interrupts, appropriately.

The primary type in that package is named Interrupt_Id and is an compiler-defined discrete type, meaning that it is either an integer type (signed or not) or an enumeration type. That representation is guaranteed so you can be sure that Interrupt_Id can be used, for example, as the index for an array type.

Package Ada.Interrupts provides functions to query whether a given interrupt is reserved, or if an interrupt has a handler attached. Procedures are defined to allow the application to attach and detach handlers, among other things. These procedures allow the application to dynamically manage interrupts. For example, when a new external device is added, perhaps as a "hot spare" replacing a damaged device, or when a new external device is simply connected to the target, the application can arrange to handle the new interrupts without having to recompile the application or restart application execution.

However, typically you will not use these procedures or functions to manage interrupts. In part that's because the architecture is usually static, i.e., the handlers are set up once and then never changed. In that case you won't need to query whether a given exception is reserved at run-time, or to check whether a handler is attached. You'd know that already, as part of the system architecture choices. For the same reasons, another mechanism for attaching handlers is more commonly used, and will be explained in that section. The package's type Interrupt_Id, however, will be used extensively.

A child package Ada.Interrupts.Names defines a target-dependent set of constants providing meaningful names for the Interrupt_Id values the target supports. Both the number of constants and their names are defined by the compiler, reflecting the variations in hardware available. This package and the enclosed constants are used all the time. For the sake of illustration, here is part of the package declaration for a Cortex M4F microcontroller supported by GNAT:

```ada
package Ada.Interrupts.Names is
  Sys_Tick_Interrupt : constant Interrupt_ID := 1;
  ...
  EXTI0_Interrupt    : constant Interrupt_ID := 8;
  ....
  DMA1_Stream0_Interrupt : constant Interrupt_ID := 13;
  ....
  HASH_RNG_Interrupt : constant Interrupt_ID := 80;
end Ada.Interrupts.Names;
```

Notice HASH_RNG_Interrupt, the name for Interrupt_Id value 80 on this target. That is the interrupt that the on-chip random number generator hardware uses to signal that a new value is available. We will use this interrupt in an example at the end of this chapter.

The representation chosen by the compiler for Interrupt_Id is very likely an integer, as in the above package, so the child package provides readable names for the numeric values. If Interrupt_Id is represented as an enumeration type the enumeral values are probably sufficiently readable, but the child package must be provided by the vendor nonetheless.
39.5 Associating Handlers With Interrupts

As we mentioned above, the Ada standard provides two ways to attach handlers to interrupts. One is procedural, described earlier. The other mechanism is automatic, achieved during elaboration of the protected object enclosing the handler procedure. The behavior is not unlike the activation of tasks: declared tasks are activated automatically as a result of their elaboration, whereas dynamically allocated tasks are activated as a result of their allocations.

We will focus exclusively on the automatic, elaboration-driven attachment model because that is the more common usage, and as a result, that is what GNAT supports on bare-board targets. It is also the mechanism that the standard Ravenscar and Jorvik profiles require. Our examples are consistent with those targets.

In the elaboration-based attachment model, we specify the interrupt to be attached to a given protected procedure within a protected object. This interrupt specification occurs within the enclosing protected object declaration. (Details in a moment.) When the enclosing PO is elaborated, the run-time library installs that procedure as the handler for that interrupt. A given PO may contain one or more interrupt handler procedures, as well as any other protected subprograms and entries.

In particular, we can associate an interrupt with a protected procedure by applying the aspect `Attach_Handler` to that procedure as part of its declaration, with the `Interrupt_Id` value as the aspect parameter. The association can also be achieved via a pragma with the same name as the aspect. Strictly speaking, the pragma `Attach_Handler` is obsolescent, but that just means that there is a newer way to make the association (i.e., the aspect). The pragma is not illegal and will remain supported. Because the pragma existed in a version of Ada prior to aspects you will see a lot of existing code using the pragma. You should become familiar with it. There's no language-driven reason to change the source code to use the aspect. New code should arguably use the aspect, but there's no technical reason to prefer one over the other.

Here is an example of a protected object with one protected procedure interrupt handler. It uses the `Attach_Handler` aspect to tie a random number generator interrupt to the `RNG_Controller.Interrupt_Handler` procedure:

```ada
protected RNG_Controller is
  ...
  entry Get_Random (Value : out UInt32);
private
  Last_Sample : UInt32 := 0;
  Buffer : Ring_Buffer;
  Data_Available : Boolean := False;

  procedure Interrupt_Handler with
    Attach_Handler => Ada.Interrupts.NamesHASH_RNG_Interrupt;
end RNG_Controller;
```

That's all that the developer must do to install the handler. The compiler and run-time library do the rest, automatically.

The local variables are declared in the private part, as required by the language, because they are shared data meant to be protected from race conditions. Therefore, the only compile-time access possible is via visible subprograms and entries declare in the visible part. Those subprograms and entries execute with mutually exclusive access so no race conditions are possible, as guaranteed by the language.

Note that procedure `Interrupt_Handler` is declared in the private part of `RNG_Controller`, rather than the visible part. That location is purely a matter of choice (unlike the variables),
but there is a good reason to hide it: application software can call an interrupt handler
procedure too. If you don’t ever intend for that to happen, have the compiler enforce your
intention. An alert code reader will then recognize that clients cannot call that procedure. If,
on the other hand, the handler is declared in the visible part, the reader must examine more
of the code to determine whether there are any callers in the application code. Granted, a
software call to an interrupt handler is rare, but not illegal, so you should state your intention
in the code in an enforceable manner.

Be aware that the Ada compiler is allowed to place restrictions on protected procedure
handlers. The compiler can restrict the content of the procedure body, for example, or it
might forbid calls to the handler from the application software. The rationale is to allow
direct invocation by the hardware, to minimize interrupt latency to the extent possible.

For completeness, here’s the same RNG_Controller protected object using the pragma
instead of the aspect to attach the interrupt to the handler procedure:

```ada
protected RNG_Controller is
  ...
  entry Get_Random (Value : out UInt32);
private
  Last_Sample  : UInt32 := 0;
  Buffer       : Ring_Buffer;
  Data_Available : Boolean := False;
  
  procedure Interrupt_Handler;
  pragma Attach_Handler (Interrupt_Handler,
                         Ada.Interrupts.Names.HASH_RNG_Interrupt);
end RNG_Controller;
```

As you can see, there isn’t much difference. The aspect is somewhat more succinct. (The
choice of where to declare the procedure remains the same.)

In this attachment model, protected declarations containing interrupt handlers must be de-
clared at the library level. That means they must be declared in library packages. (Protected
objects cannot be library units themselves, just as tasks cannot. They must be declared
within some other unit.) Here is the full declaration for the RNG_Controller PO declared
within a package — in this case within a package body:

```ada
with Ada.Interrupts.Names;
with Bounded_RingBuffers;
package body STM32.RNG.Interrupts is
  package UInt32_Buffers is new Bounded_RingBuffers (Content => UInt32);
  use UInt32_Buffers;

  protected RNG_Controller is

    entry Get_Random (Value : out UInt32);

  private

    Last_Sample  : UInt32 := 0;
    Samples      : Ring_Buffer (Upper_Bound => 9); 
                  -- arbitrary
    Data_Available : Boolean := False;
    
    procedure Interrupt_Handler with
    Attach_Handler => Ada.Interrupts.Names.HASH_RNG_Interrupt;

  end RNG_Controller;
```

(continues on next page)
But note that we're talking about protected declarations, a technical term that encompasses not only protected types but also anonymously-typed protected objects. In the RNG_Controller example, the PO does not have an explicit type declared; it is anonymously-typed. (Task objects can also be anonymously-typed.) You don't have to use a two-step process of first declaring the type and then an object of the type. If you only need one, no explicit type is required.

Although interrupt handler protected types must be declared at library level, the Ada model allows you to have an object of the type declared elsewhere, not necessarily at library level. However, note that the Ravenscar and Jorvik profiles require protected interrupt handler objects — anonymously-typed or not — to be declared at the library level too, for the sake of analysis. The profiles also require the elaboration-based attachment mechanism we have shown. For the sake of the widest applicability, and because with GNAT the most likely use-case involves either Ravenscar or Jorvik, we are following those restrictions in our examples.

### 39.6 Interrupt Priorities

Many (but not all) processors assign priorities to interrupts, with blocking and preemption among priorities of different levels, much like preemptive priority-based task semantics. Consequently, the priority semantics for interrupt handlers are as if a hardware "task," executing at an interrupt level priority, calls the protected procedure handler.

Interrupt handlers in Ada are protected procedures, which do not have priorities individually, but the enclosing protected object can be assigned a priority that will apply to the handler(s) when executing.

Therefore, protected objects can have priorities assigned using values of subtype System.Interrupt_Priority, which are high enough to require the blocking of one or more interrupts. The specific values among the priority subtypes are not standardized but the intent is that interrupt priorities are higher (more urgent) than non-interrupt priorities, as if they are declared like so in package System:

```ada
subtype Any_Priority is Integer range compiler-defined;
subtype Priority is Any_Priority range Any_Priority'First .. compiler-defined;
subtype Interrupt_Priority is Any_Priority range Priority'Last + 1 .. Any_Priority'Last;
```

For example, here are the subtype declarations in the GNAT compiler for an Arm Cortex M4 target:

```ada
subtype Any_Priority is Integer range 0 .. 255;
subtype Priority is Any_Priority range Any_Priority'First .. 240;
subtype Interrupt_Priority is Any_Priority range Priority'Last + 1 .. Any_Priority'Last;
```

Although the ranges are compiler-defined, when the Systems Programming Annex is implemented the range of System.Interrupt_Priority must include at least one value. Vendors are not required to have a distinct priority value in Interrupt_Priority for each
hardware interrupt possible on a given target. On a bare-metal target, they probably will have a one-to-one correspondence, but might not in a target with an RTOS or host OS.

A PO containing an interrupt handler procedure must be given a priority within the Interrupt_Priority subtype's range. To do so, we apply the aspect Interrupt_Priority to the PO. Perhaps confusingly, the aspect and the value's required subtype have the same name.

```ada
with System; use System;

package Gyro_Interrupts is
  protected Handler with
    -- Interrupt_Priority => Interrupt_Priority'Last
    is
  private
    procedure IRQ_Handler;
    pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
  end Handler;
end Gyro_Interrupts;
```

The code above uses the highest (most urgent) interrupt priority value but some other value could be used instead, as long as it is in the Interrupt_Priority subtype's range. Constraint_Error is raised otherwise.

There is also an alternative pragma, now obsolescent, with the same name as the aspect and subtype. Here is an example:

```ada

package Gyro_Interrupts is
  protected Handler with
    pragma Interrupt_Priority (245);
  private
    procedure IRQ_Handler;
    pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
  end Handler;
end Gyro_Interrupts;
```

In the above we set the interrupt priority to 245, presumably a value conformant with this specific target. You should be familiar with this pragma too, because there is some much existing code using it. New code should use the aspect, ideally.

If we don't specify the priority for some protected object containing an interrupt handler (using either the pragma or the aspect), the initial priority of protected objects of that type is compiler-defined, but within the range of the subtype Interrupt_Priority. Generally speaking, you should specify the priorities per those of the interrupts handled, assuming they have distinct values, so that you can reason concretely about the relative blocking behavior at run-time.

Note that the parameter specifying the priority is optional for the Interrupt_Priority pragma. When none is given, the effect is as if the value Interrupt_Priority'Last was specified.

```ada

package Gyro_Interrupts is
```

(continues on next page)
protected Handler is  
pragma Interrupt_Priority;  
private ...  
end Handler;  
end Gyro_Interrupts;

No pragma parameter is given in the above, therefore Gyro_Interrupts_Handler executes at Interrupt_Priority 'Last when invoked.

While an interrupt handler is executing, the corresponding interrupt is blocked. Therefore, the same interrupt will not be delivered again while the handler is executing. Plus, the protected object semantics mean that no software caller is also concurrently executing within the protected object. So no data race conditions are possible. If the system does not support blocking, however, the interrupt is not blocked when the handler executes.

In addition, when interrupt priorities are involved, hardware blocking typically extends to interrupts of equal or lower priority.

You should understand that a higher-priority interrupt could preempt the execution of a lower-priority interrupt's handler. Handlers do not define "critical sections" in which the processor cannot be preempted at all (other than the case of the highest priority interrupt).

Preemption does not cause data races, usually, because the typical case is to have a given protected object handle only one interrupt. It follows that only that one interrupt handler has visibility to the protected data in any given protected object, therefore only that one handler can update it. Any preempting handler would be in a different protected object, hence the preempting handler could not possibly update the data in the preempted handler's PO. No data race condition is possible.

However, protected objects can contain handlers for more than one interrupt. In that case, depending on the priorities, the execution of a higher-priority handler could preempt the execution of a lower priority handler in that same PO. Because each handler in the PO can update the local protected data, these data are effectively shared among asynchronous writers. Data race conditions are, as a result, possible.

The solution to the case of multiple handlers in a single PO is to assign the PO a priority not less than the highest of the interrupt priorities for which it contains handlers. That's known as the "ceiling priority" and works the same as when applying the ceiling for the priorities of caller tasks in the software. Then, whenever any interrupt handled by that PO is delivered, the handler executes at the ceiling priority, not necessarily the priority of the specific interrupt handled. All interrupts at a priority equal or lower than the PO priority are blocked, so no preemption by another handler within that same PO is possible. As a result, a handler for a higher priority interrupt must be in a different PO. If that higher priority handler is invoked, it can indeed preempt the execution of the handler for the lower priority interrupt in another PO. But because these two handlers will not be in the same PO, they will not share the data, so again no race condition is possible.

Note also that software callers will execute at the PO priority as well, so their priority may be increased during that execution. As you can see, the Ceiling Priority Protocol integrates application-level priorities, for tasks and protected objects, with interrupt-level priorities for interrupt handlers.

The Ceiling Locking Protocol is requested by specifying the Ceiling_Locking policy (see ARM D.3) to the pragma Locking_Policy. Both Ravenscar and Jorvik do so, automatically.
Learning Ada

39.7 Common Design Idioms

In this section we explore some of the common idioms used when writing interrupt handlers in Ada.

39.7.1 Parameterizing Handlers

Suppose we have more than one instance of a kind of device. For example, multiple DMA controllers are often available on a System-on-Chip such as an Arm microcontroller. We can simplify our code by defining a device driver type, with one object of the type per supported hardware device. This is the same abstract data type (ADT) approach we'd take for software objects in application code, and in general for device drivers when multiple hardware instances are available.

We can also apply the ADT approach to interrupt handlers when we have multiple devices of a given kind that can generate interrupts. In this case, the type will be fully implemented as a protected type containing at least one interrupt handling procedure, with or without additional protected procedures or entries.

As is the case with abstract data types in general, we can tailor each object with discriminants defined with the type, in order to "parameterize" the type and thus allow distinct objects to have different characteristics. For example, we might define a bounded buffer ADT with a discriminant specifying the upper bound, so that distinct objects of the single type could have different bounds. In the case of hardware device instances, one of these parameters will often specify the device being driven, but we can also specify other device-specific characteristics. In particular, for interrupt handler types both the interrupt to handle and the interrupt priority can be discriminants. That's possible because the aspects/pragmas do not require their values to be specified via literals, unlike what was done in the RNG_Controller example above.

For example, here is the declaration for an interrupt handler ADT named DMA_Interrupt_Controller. This type manages the interrupts for a given DMA device, known as a DMA_Controller. Type DMA_Controller is itself an abstract data type, declared elsewhere.

```ada
protected type DMA_Interrupt_Controller
  (Controller : not null access DMA_Controller;
   Stream : DMA_Stream_Selector;
   IRQ : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System.Interrupt_Priority)
with
  Interrupt_Priority => IRQ_Priority
is
    procedure Start_Transfer
      (Source : Address;
       Destination : Address;
       Data_Count : UInt16);

    procedure Abort_Transfer (Result : out DMA_Error_Code);

    procedure Clear_Transfer_State;

    function Buffer_Error return Boolean;

    entry Wait_For_Completion (Status : out DMA_Error_Code);

private
```

(continues on next page)
procedure Interrupt_Handler with Attach_Handler => IRQ;

No_Transfer_In_Progress : Boolean := True;
Last_Status : DMA_Error_Code := DMA_No_Error;
Had_Buffer_Error : Boolean := False;

end DMA_Interrupt_Controller;

In the above, the Controller discriminant provides an access value designating the specific DMA_Controller device instance to be managed. Each DMA device supports multiple independent conversion "streams" so the Stream discriminant specifies that characteristic. The IRQ and IRQ_Priority discriminants specify the handler values for that specific device and stream. These discriminant values are then used in the Interrupt_Priority pragma and the Attach_Handler aspect in the private part. ("IRQ" is a command handler name across programming languages, and is an abbreviation for "interrupt request.")

Here then are the declarations for two instances of the interrupt handler type:

DMA2_Stream0 : DMA_Interrupt_Controller
(Controller => DMA_2'Access,
Stream => Stream_0,
IRQ => DMA2_Stream0_Interrupt,
IRQ_Priority => Interrupt_Priority'Last);

DMA2_Stream5 : DMA_Interrupt_Controller
(Controller => DMA_2'Access,
Stream => Stream_5,
IRQ => DMA2_Stream5_Interrupt,
IRQ_Priority => Interrupt_Priority'Last);

In the above, both objects DMA2_Stream0 and DMA2_Stream5 are associated with the same object named DMA2, an instance of the DMA_Controller type. The difference in the objects is the stream that generates the interrupts they handle. One object handles Stream_0 interrupts and the other handles those from Stream_5. Package Ada.Interrupts.Names for this target (for GNAT) declares distinct names for the streams and devices generating the interrupts, hence DMA2_Stream0_Interrupt and DMA2_Stream5_Interrupt.

On both objects the priority is the highest interrupt priority (and hence the highest overall), Interrupt_Priority'Last. That will work, but of course all interrupts will be blocked during the execution of the handler, as well as the execution of any other subprogram or entry in the same PO. That means that the clock interrupt is blocked for that interval, for example. We use that interrupt value in our demonstrations for expediency, but in a real application you'd almost certainly use a lower value specific to the interrupt handled.

We could reduce the number of discriminants, and also make the code more robust, by taking advantage of the requirement that type Interrupt_Id be a discrete type. As such, it can be used as the index type into arrays. Here is a driver example with only the Interrupt_Id discriminant required:

Device_Priority : constant array (Interrupt_Id) of Interrupt_Priority := ( ... );

protected type Device_Interface
(IRQ : Interrupt_Id)
with
Interrupt_Priority => Device_Priority (IRQ)
is
procedure Handler with Attach_Handler => IRQ;
...
end Device_Interface;

Now we use the one IRQ discriminant both to assign the priorities for distinct objects and
to attach their handler procedures.

### 39.7.2 Multi-Level Handlers

Interrupt handlers are intended to be very brief, in part because they prevent lower priority interrupts and application tasks from executing.

However, complete interrupt processing may require more than just the short protected procedure handler’s activity. Therefore, two levels of handling are common: the protected procedure interrupt handler and a task. The handler does the least possible and then signals the task to do the rest.

Of course, sometimes the handler does everything required and just needs to signal the application. In that case, the awakened task does no further "interrupt processing" but simply uses the result.

Regardless, the same issues apply: 1) How do application tasks synchronize with the handlers? Assuming the task is not polling the event, at some point the task must stop what it was doing and suspend, waiting for the handler to signal it. 2) Once synchronized, how can the handlers pass data to the tasks?

Using protected objects for interrupt handling provides an efficient mechanism that elegantly addresses both issues. In addition, when data communication is not required, another standard language mechanism is available. These give rise to two design idioms. We will explore both.

In the first idiom, the protected object contains a protected entry as well as the interrupt handler procedure. The task suspends on the entry when ready for the handler results, controlled by the barrier condition as usual. The protected handler procedure responds to interrupts, managing data (if any) as required. When ready, based on what the handler does, the handler sets the entry barrier to True. That allows the suspended task to execute the entry body. The entry body can do whatever is required, possibly just copying the local protected data to the entry parameters. Of course, the entry may be used purely for synchronizing with the handler, i.e., suspending and resuming the task, in which case there would be no parameters passed.

The image below depicts this design.
The DMA Interrupt_Controller described earlier actually uses this design.

```ada
protected type DMA_Interrupt_Controller
  (Controller : not null access DMA_Controller;
   Stream : DMA_Stream_Selector;
   IRQ : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System.Interrupt_Priority)
with
  Interrupt_Priority => IRQ_Priority
is

  procedure Start_Transfer
  (Source : Address;
   Destination : Address;
   Data_Count : UInt16);

  procedure Abort_Transfer (Result : out DMA_Error_Code);

  procedure Clear_Transfer_State;

  function Buffer_Error return Boolean;

  entry Wait_For_Completion (Status : out DMA_Error_Code);

private

  procedure Interrupt_Handler with Attach_Handler => IRQ;

  No_Transfer_In_Progress : Boolean := True;
  Last_Status : DMA_Error_Code := DMA_No_Error;
  Had_Buffer_Error : Boolean := False;

end DMA_Interrupt_Controller;
```

The client application code (task) calls procedure Start_Transfer to initiate the DMA transaction, then presumably goes off to accomplish something else, and eventually calls the Wait_For_Completion entry. That call blocks the task if the device has not yet completed the DMA transfer. The interrupt handler procedure, cleverly named Interrupt_Handler,
Learning Ada

handles the interrupts, one of which indicates that the transfer has completed. Device errors also generate interrupts so the handler detects them and acts accordingly. Eventually, the handler sets the barrier to True and the task can get the status via the entry parameter.

```ada
procedure Start_Transfer
(Source   : Address;
Destination: Address;
Data_Count: UInt16)
is
begin
No_Transfer_In_Progress := False;
Had_Buffer_Error := False;
Clear_All_Status (Controller.all, Stream);
Start_Transfer_with_Interrupts
(Controller.all, Stream,
Source,
...,
Enabled_Interrupts =>
   (Half_Transfer_Complete_Interrupt => False,
    others => True));
end Start_Transfer;

entry Wait_For_Completion
(Status : out DMA_Error_Code)
when
   No_Transfer_In_Progress
is
begin
Status := Last_Status;
end Wait_For_Completion;
```

In the above, the entry barrier consists of the Boolean variable No_Transfer_In_Progress. Procedure Start_Transfer first sets that variable to False so that a caller to Wait_For_Completion will suspend until the transaction completes one way or the other. Eventually, the handler sets No_Transfer_In_Progress to True.

```ada
procedure Interrupt_Handler is
   subtype Checked_Status_Flag is DMA_Status_Flag with
      Static_Predicate => Checked_Status_Flag /= Half_Transfer_Complete_Indicated;
begin
   for Flag in Checked_Status_Flag loop
      if Status (Controller.all, Stream, Flag) then
         case Flag is
            when FIFO_Error_Indicated =>
               Last_Status := DMA_FIFO_Error;
               Had_Buffer_Error := True;
               No_Transfer_In_Progress := not Enabled (Controller.all, Stream);
            when Direct_Mode_Error_Indicated =>
               Last_Status := DMA_Direct_Mode_Error;
               No_Transfer_In_Progress := not Enabled (Controller.all, Stream);
            when Transfer_Error_Indicated =>
               Last_Status := DMA_Transfer_Error;
               No_Transfer_In_Progress := True;
            when Transfer_Complete_Indicated =>
               Last_Status := DMA_No_Error;
               No_Transfer_In_Progress := True;
            end case;
            Clear_Status (Controller.all, Stream, Flag);
         end if;
      end loop;
```

(continues on next page)
This device driver doesn’t bother with interrupts indicating that transfers are half-way complete so that specific status flag is ignored. In response to an interrupt, the handler checks each status flag to determine what happened. Note the resulting assignments for both the protected variables Last_Status and No_Transfer_In_Progress. The variable No_Transfer_In_Progress controls the entry, and Last_Status is passed to the caller via the entry formal parameter. When the interrupt handler exits, the resulting protected action allows the now-enabled entry call to execute.

In the second design idiom, the handler again synchronizes with the application task, but not using a protected entry.

The image below depicts this design.

In this approach, the task synchronizes with the handler using a Suspension_Object variable. The type Suspension_Object is defined in the language standard package Ada.Synchronous_Task_Control. Essentially, the type provides a thread-safe Boolean flag. Callers can suspend themselves (hence the package name) until another task resumes them by setting the flag to True. Here’s the package declaration, somewhat elided:

```ada
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True (S : in out Suspension_Object);
private
  ...
end Ada.Synchronous_Task_Control;
```

Tasks call Suspend_Until_True to suspend themselves on some object of the type passed
as the parameter. The call suspends the caller until that object becomes True. If it is already True, the caller continues immediately. Objects of type Suspension_Object are automatically set to False initially, and become True via a call to Set_True. As part of the return from a call to Suspend_Until_True, the flag is set back to False. As a result, you probably only need those two subprograms.

The interrupt handler procedure responds to interrupts, eventually setting some visible Suspension_Object to True so that the caller will be signaled and resume. Here’s an example showing both the protected object, with handler, and a Suspension_Object declaration:

``` ADA
with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
package Gyro_Interrupts is
  Data_Available : Suspension_Object;
  protected Handler is
    pragma Interrupt_Priority;
  private
    procedure IRQ_Handler with Attach_Handler => EXTI2_Interrupt;
  end Handler;
end Gyro_Interrupts;
```

In the code above, Gyro_Interrupts.Data_Available is the Suspension_Object variable visible both to the interrupt handler PO and the client task. EXTI2_Interrupt is "external interrupt number 2" on this particular microcontroller. It is connected to an external device, not on the SoC itself. Specifically, it is connected to a L3GD20 MEMS motion sensor\(^\text{275}\), a three-axis digital output gyroscope. This gyroscope can be either polled or generate interrupts whenever data are available. The handler is very simple:

``` ADA
with STM32.EXTI; use STM32.EXTI;
package body Gyro_Interrupts is
  protected body Handler is
    procedure IRQ_Handler is
      begin
        if External_Interrupt_Pending (EXTI_Line_2) then
          Clear_External_Interrupt (EXTI_Line_2);
          Set_True (Data_Available);
        end if;
        IRQ_Handler;
      end Handler;
end Gyro_Interrupts;
```

The handler simply clears the interrupt and resumes the caller task via a call to Set_True on the variable declared in the package spec.

The lack of an entry means that no data can be passed to the task via entry parameters. It is possible to pass data to the task but doing so would require an additional protected procedure or function.

The gyroscope hardware device interface is in package L3GD20. Here are the pertinent parts:

```ada
package L3GD20 is
  type Three_Axis_Gyroscope is tagged limited private;
  procedure Initialize
    (This : in out Three_Axis_Gyroscope;
    Port : Any_SPI_Port;
    Chip_Select : Any_GPIO_Point);
  ...
  procedure Enable_Data_Ready_Interrupt (This : in out Three_Axis_Gyroscope);
  ...
  type Angle_Rate is new Integer_16;
  type Angle_Rates is record
    X : Angle_Rate; -- pitch, per Figure 2, pg 7 of the Datasheet
    Y : Angle_Rate; -- roll
    Z : Angle_Rate; -- yaw
  end record with Size => 3 * 16;
  ...
  procedure Get_Raw_Angle_Rates
    (This : Three_Axis_Gyroscope;
    Rates : out Angle_Rates);
  ...
end L3GD20;
```

With those packages available, we can write a simple main program to use the gyro. The real demo displayed the readings on an LCD but we've elided all those irrelevant details:

```ada
with Gyro_Interrupts;
with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
with L3GD20; use L3GD20;
with STM32.Board;
...
procedure Demo_L3GD20 is
  Axes : L3GD20.Angle_Rates;
  ...
  procedure Await_Raw_Angle_Rates (Rates : out L3GD20.Angle_Rates) is
  begin
    Suspend_Until_True (Gyro_Interrupts.Data_Available);
    L3GD20.Get_Raw_Angle_Rates (STM32.Board.Gyro, Rates);
    Await_Raw_Angle_Rates;
  end Await_Raw_Angle_Rates;
  ...
begin
  Configure_Gyro;
  Configure_Gyro_Interrupt;
```

(continues on next page)
The demo is a main procedure, even though we've been describing the client application code in terms of tasks. The main procedure is executed by the implicit "environment task" so it all still works. Await_Raw_Angle_Rates suspends (if necessary) on Gyro_Interrupts. Data_Available and then calls L3GD20.Get_Raw_Angle_Rates to get the rate values.

The operations provided by Suspension_Object are faster than protected entries, and noticeably so. However, that performance difference is due to the fact that Suspension_Object provides so much less capability than entries. In particular, there is no notion of protected actions, nor expressive entry barriers for condition synchronization, nor parameters to pass data while synchronized. Most importantly, there is no caller queue, so at most one caller can be waiting at a time on any given Suspension_Object variable. You'll get Program_Error if you try. Protected entries should be your first design choice. Note that the Ravenscar restrictions can make use of Suspension_Object much more likely.

### 39.8 Final Points

As you can see, the semantics of protected objects are a good fit for interrupt handling. However, other forms of handlers are allowed to be supported. For example, the compiler and RTL for a specific target may include support for interrupts generated by a device known to be available with that target. For illustration, let's imagine the target always has a serial port backed by a UART. In addition to handlers as protected procedure without parameters, perhaps the compiler and RTL support interrupt handlers with a single parameter of type Unsigned_8 (or larger) as supported by the UART.

Overall, the interrupt model defined and supported by Ada is quite close to the canonical model presented by most programming languages, in part because it matches the model presented by typical hardware.
CONCLUSION

In the introduction to this course, we defined an "embedded system" as a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system. That, in itself, sets this kind of programming apart from the more typical host-oriented programming. But the context also implies fewer resources are available, especially memory and electrical power, as well as processor power. Add to those limitations a frequent reliability requirement and you have a demanding context for development.

Using Ada can help you in this context, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal in this course was to get you started in using it well.

To that end, we spent a lot of time talking about how to use Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data. Ada has a lot of support for this activity so there was much to explore.

Likewise, we examined development using Ada in combination with other languages, a not uncommon approach. Specifically, we saw how to interface with code and data written in other languages, and how (and why) to work with assembly language. Development in just one language is becoming less common over time so these were important aspects to know.

One of the more distinctive activities of embedded programming involves interacting with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. (This can be one of the more entertaining activities as well.) We covered how to interact with these memory-mapped devices using representation specifications, data structures that simplified the functional code, and time-honored aspects of software engineering, including abstract data types.

Finally, we explored how to handle interrupts in Ada, another distinctive part of embedded systems programming. As we saw, Ada has extensive support for handling interrupts, using the same building blocks — protected objects — used in concurrent programming. These constructs provide a way to handle interrupts that is as portable as possible, in what is otherwise a very hardware-specific endeavor.

In the course, we mentioned a library of freely-available device drivers in Ada known as the Ada Driver Library (ADL). The ADL is a good resource for learning how Ada can be used to develop software for embedded systems using real-world devices and processors. Becoming familiar with it would be a good place to go next. Contributing to it would be even better! The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.
Part V

What's New in Ada 2022
This course presents an overview of the new features of the latest Ada 2022 standard. This document was written by Maxim Reznik and reviewed by Richard Kenner.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

**Note:** Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- **Project:** Courses.Intro_To_Ada.Imperative_Language.Greet
- **MD5:** cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

---

276 [http://creativecommons.org/licenses/by-sa/4.0](http://creativecommons.org/licenses/by-sa/4.0)
INTRODUCTION

This is a collection of short code examples demonstrating new features of the Ada 2022 Standard as they are implemented in GNAT Ada compiler.

To use some of these features, you may need to use a compiler command line switch or pragma. Compilers starting with GNAT Community Edition 2021 or GCC 11 use `pragma Ada_2022;` or the `-gnat2022 switch. Older compilers use `pragma Ada_2020;` or `-gnat2020. To use the square brackets syntax or ‘Reduce expressions, you need `pragma Extensions_Allowed (On);` or the `-gnatX switch.

41.1 References

- Draft Ada 2022 Standard
- Ada 202x support in GNAT blog post
'IMAGE ATTRIBUTE FOR ANY TYPE

Note: Attribute 'Image for any type is supported by
- GNAT Community Edition 2020 and latter
- GCC 11

42.1 'Image attribute for a value

Since the publication of the Technical Corrigendum \(^{283}\) in February 2016, the 'Image attribute can now be applied to a value. So instead of My_Type'Image (Value), you can just write Value'Image, as long as the Value is a name\(^{284}\). These two statements are equivalent:

```ada
Ada.Text_IO.Put_Line (Ada.Text_IO.Page_Length'Image);
```

42.2 'Image attribute for any type

In Ada 2022, you can apply the 'Image attribute to any type, including records, arrays, access types, and private types. Let's see how this works. We'll define array, record, and access types and corresponding objects and then convert these objects to strings and print them:

Listing 1: main.adb

```ada
pragma Ada_2022;

with Ada.Text_IO;

procedure Main is
    type Vector is array (Positive range <>) of Integer;
    V1 : aliased Vector := [1, 2, 3];

    type Text_Position is record
        Line, Column : Positive;
    end record;

    procedure Print_Vector is
        V1 Image : String := V1 Image;
    end procedure;

    procedure Print_Text_Position is
        TP Image : String := Text_Position Image;
    end procedure;

    with Ada.Text_IO;
    begin
        Ada.Text_IO.Put_Line (V1 Image);
        Ada.Text_IO.Put_Line (Text_Position Image (V1));
        Ada.Text_IO.Put_Line (TP Image);
    end Main;
```

\(^{283}\) https://reznikmm.github.io/ada-auth/rm-4-NC/RM-0-1.html
\(^{284}\) https://reznikmm.github.io/ada-auth/rm-4-NC/RM-4-1.html#S0091
end record;

Pos : constant Text_Position := (Line => 10, Column => 3);

type Vector_Access is access all Vector;

V1_Ptr : constant Vector_Access := V1'Access;

begin
  Ada.Text_IO.Put_Line (V1'Image);
  Ada.Text_IO.Put_Line (Pos'Image);
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put_Line (V1_Ptr'Image);
end Main;

Code block metadata
Project: Courses.Ada 2022 Whats New.Image_Attribute
MD5: 47945f0f8a4ba37b838f87b7e5acaa49

Runtime output

[ 1, 2, 3]
(LINE => 10,
 COLUMN => 3)
(access 7ffdd1e99348)

$ gprbuild -q -P main.gpr
Build completed successfully.
$ ./main
[ 1, 2, 3]
(LINE => 10,
 COLUMN => 3)
(access 7fff64b23988)

Note the square brackets in the array image output. In Ada 2022, array aggregates could be written this way (page 1245)!

42.3 References

- ARM 4.10 Image Attributes\textsuperscript{285}
- AI12-0020-1\textsuperscript{286}

\textsuperscript{285} http://www.ada-auth.org/standards/22aarm/html/AA-4-10.html
\textsuperscript{286} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0020-1.txt
REDEFINING THE 'IMAGE ATTRIBUTE

In Ada 2022, you can redefine 'Image attribute for your type, though the syntax to do this has been changed several times. Let's see how it works in GNAT Community 2021.

Note: Redefining attribute 'Image is supported by

- GNAT Community Edition 2021 (using Text_Buffers)
- GNAT Community Edition 2020 (using Text_Output_Utils)
- GCC 11 (using Text_Output_Utils)

In our example, let's redefine the 'Image attribute for a location in source code. To do this, we provide a new Put_Image aspect for the type:

Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;
with Ada.Strings.Text_Buffers;

procedure Main is
  type Source_Location is record
    Line : Positive;
    Column : Positive;
  end record
  with Put_Image => My_Put_Image;

procedure My_Put_Image
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Value : Source_Location);

procedure My_Put_Image
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Value : Source_Location)
is
  Line : constant String := Value.Line'Image;
  Column : constant String := Value.Column'Image;
  Result : constant String :=
    Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
begin
  Output.Put (Result);
end My_Put_Image;

Line_10 : constant Source_Location := (Line => 10, Column => 1);
begin
  (continues on next page)
```
43.1 What's the Root_Buffer_Type?

Let's see how it's defined in the Ada.Strings.Text_Buffers package.

```ada
type Root_Buffer_Type is abstract tagged limited private;
procedure Put
  (Buffer : in out Root_Buffer_Type;
   Item   : in   String) is abstract;
```

In addition to Put, there are also Wide_Put, Wide_Wide_Put, Put_UTF_8, Wide_Put_UTF_16. And also New_Line, Increase_Indent, Decrease_Indent.

43.2 Outdated draft implementation

GNAT Community Edition 2020 and GCC 11 both provide a draft implementation that's incompatible with the Ada 2022 specification. For those versions, My_Put_Image looks like:

```ada
procedure My_Put_Image
  (Sink   : in out Ada.Strings.Text_Output.Sink'Class;
   Value  : Source_Location)
is
  Line   : constant String := Value.Line'Image;
  Column : constant String := Value.Column'Image;
  Result : constant String :=
    Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
begin
  Ada.Strings.Text_Output.Utils.Put_UTF_8 (Sink, Result);
end My_Put_Image;
```

43.3 References

- ARM 4.10 Image Attributes\(^{287}\)
- AI12-0020-1\(^{288}\)
- AI12-0384-2\(^{289}\)

\(^{288}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0020-1.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0020-1.TXT)
\(^{289}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT)
Note: User-defined literals are supported by

- GNAT Community Edition 2020
- GCC 11

In Ada 2022, you can define string, integer, or real literals for your types. The compiler will convert such literals to your type at run time using a function you provide. To do so, specify one or more new aspects:

- Integer_Literal
- Real_Literal
- String_Literal

For our example, let's define all three for a simple type and see how they work. For simplicity, we use a Wide_Wide_String component for the internal representation:

Listing 1: main.adb

```ada
pragma Ada_2022;

with Ada.Wide_Wide_Text_IO;
with Ada.Characters.Conversions;

procedure Main is

   type My_Type (Length : Natural) is record
      Value : Wide_Wide_String (1 .. Length);
   end record
   with String_Literal => From_String,
   Real_Literal => From_Real,
   Integer_Literal => From_Integer;

   function From_String (Value : Wide_Wide_String) return My_Type is
      ((Length => Value'Length, Value => Value));

   function From_Real (Value : String) return My_Type is
      ((Length => Value'Length,
      Value => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));

   function From_Integer (Value : String) return My_Type renames From_Real;

   procedure Print (Self : My_Type) is
      begin
         Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
   end Print;
```

(continues on next page)
begin
  Print ("Test " "string" ");
  Print (123);
  Print (16#DEAD_BEEF#);
  Print (2.99_792_458e+8);
end Main;

Code block metadata


MD5: 3a4a12aa148b6845a1130e818e16c405

Runtime output

Test "string"
123
16#DEAD_BEEF#
2.99_792_458e+8

As you see, real and integer literals are converted to strings while preserving the formatting in the source code, while string literals are decoded: From_String is passed the specified string value. In all cases, the compiler translates these literals into function calls.

44.1 Turn Ada into JavaScript

Do you know that '5'+3 in JavaScript is 53?

> '5'+3
'53'

Now we can get the same result in Ada! But before we do, we need to define a custom + operator:

Listing 2: main.adb

pragma Ada_2022;
with Ada.Wide_Wide_Text_IO;
with Ada.Characters.Conversions;
procedure Main is
  type My_Type (Length : Natural) is record
    Value : Wide_Wide_String (1 .. Length);
  end record
  with String Literal => From_String,
       Real_Literal    => From_Real,
       Integer_Literal => From_Integer;
  function "+" (Left, Right : My_Type) return My_Type is
    (Left.Length + Right.Length, Left.Value & Right.Value);
  function From_String (Value : Wide_Wide_String) return My_Type is
    ((Length => Value'Length, Value => Value));
  function From_Real (Value : String) return My_Type is
    ((Length => Value'Length,
      Value => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
(continues on next page)
function From_Integer (Value : String) return My_Type renames From_Real;

procedure Print (Self : My_Type) is
begin
    Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
end Print;

begin
    Print ("5" + 3);
end Main;

Jokes aside, this feature is very useful. For example it allows a “native-looking API” for big integers (page 1265).

44.2 References

- ARM 4.2.1 User-Defined Literals
- AI12-0249-1
- AI12-0342-1
45.1 Square brackets

In Ada 2022, you can use square brackets in array aggregates. Using square brackets simplifies writing both empty aggregates and single-element aggregates. Consider this:

Listing 1: show_square_brackets.ads

```ada
pragma Ada_2022;
pragma Extensions_Allowed (On);

package Show_Square_Brackets is
  type Integer_Array is array (Positive range <>) of Integer;
  Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
  New_Style_Empty : Integer_Array := [];
  Old_Style_One_Item : Integer_Array := (1 => 5);
  New_Style_One_Item : Integer_Array := [5];
end Show_Square_Brackets;
```

Short summary for parentheses and brackets

- Record aggregates use parentheses
- Container aggregates (page 1249) use square brackets
- Array aggregates can use both square brackets and parentheses, but parentheses usage is obsolescent
45.2 Iterated Component Association

There is a new kind of component association:

Vector : Integer_Array := [for J in 1 .. 5 => J * 2];

This association starts with *for* keyword, just like a quantified expression. It declares an index parameter that you can use in the computation of a component.

Iterated component associations can nest and can be nested in another association (iterated or not). Here we use this to define a square matrix:

Matrix : array (1 .. 3, 1 .. 3) of Positive :=
[for J in 1 .. 3 =>
 [for K in 1 .. 3 => J * 10 + K]];  

Iterated component associations in this form provide both element indices and values, just like named component associations:

Data : Integer_Array (1 .. 5) :=
[for J in 2 .. 3 => J, 5 => 5, others => 0];

Here Data contains (0, 2, 3, 0, 5), not (2, 3, 5, 0, 0).

Another form of iterated component association corresponds to a positional component association and provides just values, but no element indices:

Vector_2 : Integer_Array := [for X of Vector => X / 2];

You cannot mix these forms in a single aggregate.

It's interesting that such aggregates were originally proposed more than 25 years ago!

Complete code snippet:

Listing 2: show_iterated_component_association.adb

```
pragma Ada_2022;
pragma Extensions_Allowed (On); -- for square brackets

with Ada.Text_IO;

procedure Show_Iterated_Component_Association is

  type Integer_Array is array (Positive range <>) of Integer;

  Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
  New_Style_Empty : Integer_Array := [];

  Old_Style_One_Item : Integer_Array := (1 => 5);
  New_Style_One_Item : Integer_Array := [5];

  Vector : constant Integer_Array := [for J in 1 .. 5 => J * 2];

  Matrix : constant array (1 .. 3, 1 .. 3) of Positive :=
  [for J in 1 .. 3 =>
   [for K in 1 .. 3 => J * 10 + K]];  

  Data : constant Integer_Array (1 .. 5) :=
  [for J in 2 .. 3 => J, 5 => 5, others => 0];

  Vector_2 : constant Integer_Array := [for X of Vector => X / 2];
```

(continues on next page)
begin
    Ada.Text_IO.Put_Line (Vector'Image);
    Ada.Text_IO.Put_Line (Matrix'Image);
    Ada.Text_IO.Put_Line (Data'Image);
    Ada.Text_IO.Put_Line (Vector_2'Image);
end Show_Iterated_Component_Association;

45.3 References

- ARM 4.3.3 Array Aggregates\textsuperscript{293}
- AI12-0212-1\textsuperscript{294}
- AI12-0306-1\textsuperscript{295}

\textsuperscript{293} http://www.ada-auth.org/standards/22aarm/html/AA-4-3-3.html
\textsuperscript{294} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT
\textsuperscript{295} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0306-1.TXT
CONTAINER AGGREGATES

Note: Container aggregates are supported by
• GNAT Community Edition 2021
• GCC 11

Ada 2022 introduces container aggregates, which can be used to easily create values for
vectors, lists, maps, and other aggregates. For containers such as maps, the aggregate
must use named associations to provide keys and values. For other containers it uses
positional associations. Only square brackets are allowed. Here's an example:

Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;
with Ada.Containers.Vectors;
with Ada.Containers.Ordered_Maps;

procedure Main is
  package Int_Vectors is new Ada.Containers.Vectors
     (Positive, Integer);
  X : constant Int_Vectors.Vector := [1, 2, 3];
  package Float_Maps is new Ada.Containers.Ordered_Maps
     (Integer, Float);
  Y : constant Float_Maps.Map := [-10 => 1.00000E+00, 0 => 2.50000E+00, 10 => 5.51000E+00];
begin
  Ada.Text_IO.Put_Line (X'Image);
  Ada.Text_IO.Put_Line (Y'Image);
end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Container_Aggregates_1
MD5: dd1dd78890d4bf6c78b79d56abba332d

Runtime output

```
[ 1, 2, 3]
[-10 => 1.00000E+00, 0 => 2.50000E+00, 10 => 5.51000E+00]
```
At run time, the compiler creates an empty container and populates it with elements one by one. If you define a new container type, you can specify a new Aggregate aspect to enable container aggregates for your container and let the compiler know what subprograms to use to construct the aggregate:

Listing 2: main.adb

```ada
pragma Ada_2022;

procedure Main is
  package JSON is
    type JSON_Value is private
      with Integer_Literal => To_JSON_Value;
    function To_JSON_Value (Text : String) return JSON_Value;
    type JSON_Array is private
      with Aggregate => (Empty => New_JSON_Array,
                          AddUnnamed => Append);
    function New_JSON_Array return JSON_Array;
    procedure Append
      (Self : in out JSON_Array;
       Value : JSON_Value) is null;
  private
    type JSON_Value is null record;
    type JSON_Array is null record;
    function To_JSON_Value (Text : String) return JSON_Value
      is (null record);
    function New_JSON_Array return JSON_Array is (null record);
  end JSON;
  List : JSON.JSON_Array := [1, 2, 3];
begin
  -- Equivalent old initialization code
  List := JSON.New_JSON_Array;
  JSON.Append (List, 1);
  JSON.Append (List, 2);
  JSON.Append (List, 3);
end Main;
```

The equivalent for maps is:

Listing 3: main.adb

```ada
pragma Ada_2022;

procedure Main is
  package JSON is
    type JSON_Value is private
      with Integer_Literal => To_JSON_Value;
    function To_JSON_Value (Text : String) return JSON_Value;
    type JSON_Map is private
      with Aggregate => (Empty => New_JSON_Map,
                          AddUnnamed => Append);
    function New_JSON_Map return JSON_Map;
    procedure Append
      (Self : in out JSON_Map;
       Value : JSON_Value) is null;
  private
    type JSON_Value is null record;
    type JSON_Map is null record;
    function To_JSON_Value (Text : String) return JSON_Value
      is (null record);
    function New_JSON_Map return JSON_Map is (null record);
  end JSON;
  Map : JSON.JSON_Map := [1 => 2, 3 => 4];
begin
  -- Equivalent old initialization code
  Map := JSON.New_JSON_Map;
  JSON.Append (Map, 1 => 2);
  JSON.Append (Map, 3 => 4);
end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Container_Aggregates_2
MD5: 9cf1fefa4a725083c50794146d5cbde7

---
with Integer_Literal => To_JSON_Value;

function To_JSON_Value (Text : String) return JSON_Value;

type JSON_Object is private
with Aggregate => (Empty => New_JSON_Object, Add_Named => Insert);

function New_JSON_Object return JSON_Object;

procedure Insert
(Self : in out JSON_Object;
Key : Wide_Wide_String;
Value : JSON_Value) is null;

private

type JSON_Value is null record;
type JSON_Object is null record;

function To_JSON_Value (Text : String) return JSON_Value
is (null record);

function New_JSON_Object return JSON_Object is (null record);
end JSON;

Object : JSON.JSON_Object := ["a" => 1, "b" => 2, "c" => 3];

begin
-- Equivalent old initialization code
Object := JSON.New_JSON_Object;
JSON.Insert (Object, "a", 1);
JSON.Insert (Object, "b", 2);
JSON.Insert (Object, "c", 3);
end Main;

Code block metadata

MD5: 758ced718aa9a4eefa32325543eb3b1e

You can't specify both Add_Named and Add_Unnamed subprograms for the same type. This prevents you from defining JSON_Value with both array and object aggregates present. But we can define conversion functions for array and object and get code almost as dense as the same code in native JSON. For example:

Listing 4: main.adb

pragma Ada_2022;

procedure Main is

package JSON is
    type JSON_Value is private
        with Integer_Literal => To_Value, String_Literal => To_Value;
    function To_Value (Text : String) return JSON_Value;
    function To_Value (Text : Wide_Wide_String) return JSON_Value;

    type JSON_Object is private
        with Aggregate => (Empty => New_JSON_Object, Add_Named => Insert);

(continues on next page)
function New_JSON_Object return JSON_Object;

procedure Insert
(Self : in out JSON_Object;
  Key : Wide_Wide_String;
  Value : JSON_Value) is null;

function From_Object (Self : JSON_Object) return JSON_Value;

type JSON_Array is private
  with Aggregate => (Empty => New_JSON_Array,
                     AddUnnamed => Append);

function New_JSON_Array return JSON_Array;

procedure Append
(Self : in out JSON_Array;
  Value : JSON_Value) is null;

function From_Array (Self : JSON_Array) return JSON_Value;

private
  type JSON_Value is null record;
  type JSON_Object is null record;
  type JSON_Array is null record;

  function To_Value (Text : String) return JSON_Value is
    (null record);
  function To_Value (Text : Wide_Wide_String) return JSON_Value is
    (null record);
  function New_JSON_Object return JSON_Object is
    (null record);
  function New_JSON_Array return JSON_Array is
    (null record);
  function From_Object (Self : JSON_Object) return JSON_Value is
    (null record);
  function From_Array (Self : JSON_Array) return JSON_Value is
    (null record);
end JSON;

function "+" (X : JSON.JSON_Object) return JSON.JSON_Value
renames JSON.From_Object;
function "+" (X : JSON.JSON_Array) return JSON.JSON_Value
renames JSON.From_Array;

Offices : JSON.JSON_Array :=
  [+["name" => "North American Office",
     "phones" => ["1 877 787 4628",
                  "1 866 787 4232",
                  "1 212 620 7300"],
     "email" => "info@adacore.com"],
  +["name" => "European Office",
     "phones" => ["33 1 49 70 67 16",
                   "33 1 49 70 05 52"],
     "email" => "info@adacore.com"]];

begin
  -- Equivalent old initialization code is too long to print it here
  null;
end Main;
The Offices variable is supposed to contain this value:

```json
[{
  "name": "North American Office",
  "phones": [18777874628, 18667874232, 12126207300],
  "email": "info@adacore.com"
}, {
  "name": "European Office",
  "phones": [33149706716, 33149700552],
  "email": "info@adacore.com"
}]
```

### 46.1 References

- ARM 4.3.5 Container Aggregates[^296]
- AI12-0212-1[^297]

[^296]: http://www.ada-auth.org/standards/22aarm/html/AA-4-3-5.html
[^297]: http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT
Sometimes you need to create a copy of an object, but with a few modifications. Before Ada 2022, doing this involves a dummy object declaration or an aggregate with associations for each property. The dummy object approach doesn't work in contract aspects or when there are limited components. On the other hand, re-listing properties in an large aggregate can be very tedious and error-prone. So, in Ada 2022, you can use a delta aggregate instead.

### 47.1 Delta aggregate for records

The delta aggregate for a record type looks like this:

```ada
type Vector is record
  X, Y, Z : Float;
end record;

Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
```

The more components you have, the more you will like the delta aggregate.

### 47.2 Delta aggregate for arrays

You can also use delta aggregates for arrays to change elements, but not bounds. Moreover, it only works for one-dimensional arrays of non-limited components.

```ada
type Vector_3D is array (1 .. 3) of Float;

Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
```

You can use parentheses for array aggregates, but you can't use square brackets for record aggregates.

Here is the complete code snippet:
Learning Ada

Listing 1: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;
procedure Main is
  type Vector is record
    X, Y, Z : Float;
  end record;
  Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
  Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
  type Vector_3D is array (1 .. 3) of Float;
  Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
  Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
begin
  Ada.Text_IO.Put (Float'Image (Projection_1.X));
  Ada.Text_IO.Put (Float'Image (Projection_1.Y));
  Ada.Text_IO.Put (Float'Image (Projection_1.Z));
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put (Float'Image (Projection_2 (1)));
  Ada.Text_IO.Put (Float'Image (Projection_2 (2)));
  Ada.Text_IO.Put (Float'Image (Projection_2 (3)));
  Ada.Text_IO.New_Line;
end Main;
```

47.3 References

- ARM 4.3.4 Delta Aggregates\(^\text{298}\)
- AI12-0127-1\(^\text{299}\)

\(^{298}\) [http://www.ada-auth.org/standards/22aarm/html/AA-4-3-4.html]
\(^{299}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0127-1.TXT]
CHAPTER
FORTYEIGHT

TARGET NAME SYMBOL (@)

**Note:** Target name symbol is supported by
- GNAT Community Edition 2019
- GCC 9

Ada 2022 introduces a new symbol, @, which can only appear on the right hand side of an assignment statement. This symbol acts as the equivalent of the name on the left hand side of that assignment statement. It was introduced to avoid code duplication: instead of retyping a (potentially long) name, you can use @. This symbol denotes a constant, so you can't pass it into [in] out arguments of a subprogram.

As an example, let's calculate some statistics for My_Data array:

Listing 1: statistics.ads

```ada
pragma Ada_2022;

package Statistics is
  type Statistic is record
    Count : Natural := 0;
    Total : Float := 0.0;
  end record;

  My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];
  Statistic_For_My_Data : Statistic;
end Statistics;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Assignment_Tagged_Intro
MD5: 5cc813a4a22d3acc8418b0c1c6df3877

To do this, we loop over My_Data elements:

Listing 2: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;

procedure Main is
  type Statistic is record
    Count : Natural := 0;
  end record;
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
Total : Float := 0.0;
end record;

My_Data : constant array (1 .. 5) of Float :=
[for J in 1 .. 5 => Float (J)];

Statistic_For_My_Data : Statistic;

begin
  for Data of My_Data loop
    Statistic_For_My_Data.Count := @ + 1;
    Statistic_For_My_Data.Total := @ + Data;
  end loop;
  Ada.Text_IO.Put_Line (Statistic_For_My_Data'Image);
end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Assignment_Tagged_2
MD5: 10dd019f4c09bc950895a93b3a88b778

Runtime output

(COUNT => 5,
TOTAL => 1.50000E+01)

Each right hand side is evaluated only once, no matter how many @ symbols it contains.
Let's verify this by introducing a function call that prints a line each time it's called:

Listing 3: main.adb

```ada
pragma Ada_2022;
with Ada.Text_IO;

procedure Main is

  My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];

  function To_Index (Value : Positive) return Positive is
  begin
    Ada.Text_IO.Put_Line ("To_Index is called.");
    return Value;
  end To_Index;

begin
  My_Data (To_Index (1)) := @ ** 2 - 3.0 * @;
  Ada.Text_IO.Put_Line (My_Data'Image);
end Main;
```

Code block metadata

MD5: 98d6afbaea5c0f6cd2bebe6b39962ad3

Runtime output

To_Index is called.
[-2.00000E+00, 2.00000E+00, 3.00000E+00, 4.00000E+00, 5.00000E+00]
This use of @ may look a bit cryptic, but it's the best solution that was found. Unlike other languages (e.g., \( \text{sum} \ += \text{x}; \) in C), this approach lets you use @ an arbitrary number of times within the right hand side of an assignment statement.

### 48.1 Alternatives

In C++, the previous statement could be written with a reference type (one line longer!):

```cpp
auto& a = my_data[to_index(1)];
a = a * a - 3.0 * a;
```

In Ada 2022, you can use a similar renaming:

```ada
declare
   A renames My_Data (To_Index (1));
begin
   A := A ** 2 - 3.0 * A;
end;
```

Here we use a new short form of the rename declaration, but this still looks too heavy, and even worse, it can't be used for discriminant-dependent components.

### 48.2 References

- ARM 5.2.1 Target Name Symbols\(^{300}\)
- AI12-0125-3\(^{301}\)

---


\(^{301}\) [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT)
Enumeration representation attributes are supported by

- GNAT Community Edition 2019
- GCC 9

Enumeration types in Ada are represented as integers at the machine level. But there are actually two mappings from enumeration to integer: a literal position and a representation value.

### 49.1 Literal positions

Each enumeration literal has a corresponding position in the type declaration. We can easily obtain it from the `Type'Pos (Enum)` attribute.

```ada
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure Main is
begin
  Ada.Text_IO.Put (“Pos(False) =”);
  Ada.Integer_Text_IO.Put (Boolean'Pos (False));
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put (“Pos(True) =”);
  Ada.Integer_Text_IO.Put (Boolean'Pos (True));
end Main;
```

**Code block metadata**

- MD5: de7c39f83f7df231dd6486065799996a8

**Runtime output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos(False)</td>
<td>0</td>
</tr>
<tr>
<td>Pos(True)</td>
<td>1</td>
</tr>
</tbody>
</table>

For the reverse mapping, we use `Type'Val (Int)`: 
Listing 2: main.adb

```ada
with Ada.Text_IO;

procedure Main is
begin
   Ada.Text_IO.Put_Line (Boolean 'Val (0)'Image);
   Ada.Text_IO.Put_Line (Boolean 'Val (1)'Image);
end Main;
```

**Code block metadata**

MD5: 43f712d25552970bccc4c0c84089d927

**Runtime output**

FALSE
TRUE

### 49.2 Representation values

The representation value defines the *internal* code, used to store enumeration values in memory or CPU registers. By default, enumeration representation values are the same as the corresponding literal positions, but you can redefine them. Here, we created a copy of `Boolean` type and assigned it a custom representation.

In Ada 2022, we can get an integer value of the representation with `Type'Enum_Rep(Enum)` attribute:

Listing 3: main.adb

```ada
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure Main is
   type My_Boolean is new Boolean;
   for My_Boolean use (False => 3, True => 6);
begin
   Ada.Text_IO.Put ("Enum_Rep(False) = ");
   Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep (False));
   Ada.Text_IO.New_Line;
   Ada.Text_IO.Put ("Enum_Rep(True) = ");
   Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep (True));
end Main;
```

**Code block metadata**

MD5: 384ad9de7124c8131aa83ab71da58964

**Runtime output**

Enum_Rep(False) = 3
Enum_Rep(True) = 6

And, for the reverse mapping, we can use `Type'Enum_Val (Int):`
Learning Ada

Listing 4: main.adb

```ada
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure Main is
  type My_Boolean is new Boolean;
  for My_Boolean use (False => 3, True => 6);
begin
  Ada.Text_IO.Put_Line (My_Boolean'Enum_Val (3)'Image);
  Ada.Text_IO.Put_Line (My_Boolean'Enum_Val (6)'Image);
  Ada.Text_IO.Put ("Pos(False) =");
  Ada.Integer_Text_IO.Put (My_Boolean'Pos (False));
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put ("Pos(True) =");
  Ada.Integer_Text_IO.Put (My_Boolean'Pos (True));
end Main;
```

Code block metadata

MD5: 6e06202472d4cf0ea7c68461ac7afcb1

Runtime output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>Pos(False) =</td>
<td>0</td>
</tr>
<tr>
<td>Pos(True)   =</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that the 'Val(X)'/'Pos(X) behaviour still is the same. Custom representations can be useful for integration with a low level protocol or hardware.

49.3 Before Ada 2022

This doesn't initially look like an important feature, but let's see how we'd do the equivalent with Ada 2012 and earlier versions. First, we need an integer type of matching size, then we instantiate Ada.Unchecked_Conversion. Next, we call To_Int/From_Int to work with representation values. And finally an extra type conversion is needed:

Listing 5: main.adb

```ada
with Ada.Text_IO;
with Ada.Integer_Text_IO;
with Ada.Unchecked_Conversion;

procedure Main is
  type My_Boolean is new Boolean;
  for My_Boolean use (False => 3, True => 6);
  type My_Boolean_Int is range 3 .. 6;
  for My_Boolean_Int'Size use My_Boolean'Size;
  function To_Int is new Ada.Unchecked_Conversion
    (My_Boolean, My_Boolean_Int);
  function From_Int is new Ada.Unchecked_Conversion
```

(continues on next page)
begin
  Ada.Text_IO.Put ("To_Int(False) =");
  Ada.Integer_Text_IO.Put (Integer (To_Int (False)));
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put ("To_Int(True) =");
  Ada.Integer_Text_IO.Put (Integer (To_Int (True)));
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put ("From_Int (3) =");
  Ada.Text_IO.Put_Line (From_Int (3)'Image');
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put ("From_Int (6) =");
  Ada.Text_IO.Put_Line (From_Int (6)'Image');
end Main;

Even with all that, this solution doesn't work for generic formal type (because T’Size must be a static value)!

We should note that these new attributes may already be familiar to GNAT users because they've been in the GNAT compiler for many years.

### 49.4 References

- ARM 13.4 Enumeration Representation Clauses[^302]
- AI12-0237-1[^303]

[^303]: [http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0237-1.TXT](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0237-1.TXT)
Note:  Big numbers are supported by

- GNAT Community Edition 2020
- GCC 11
- GCC 10 (draft, no user defined literals)

Ada 2022 introduces big integers and big real types.

### 50.1 Big Integers

The package Ada.Numerics.Big_Numbers.Big_Integers contains a type Big_Integer and corresponding operations such as comparison (=, <, >, <=, >=), arithmetic (+, -, *, /, rem, mod, abs, **), Min, Max and Greatest_Common_Divisor. The type also has Integer_Literal and Put_Image aspects redefined, so you can use it in a natural manner.

```ada
Ada.Text_IO.Put_Line (Big_Integer'Image (2 ** 256));
```

1157920892316195423570985008687907853269984665640564039457584007913129639936

### 50.2 Tiny RSA implementation

Note:  Note that you shouldn't use Big_Numbers for cryptography because it's vulnerable to timing side-channels attacks.

We can implement the RSA algorithm\(^{304}\) in a few lines of code. The main operation of RSA is \((m^d) \pmod{n}\). But you can't just write \(m \times d\), because these are really big numbers and the result won't fit into memory. However, if you keep intermediate result \(m^d \pmod{n}\) during the \(m^d\) calculation, it will work. Let's write this operation as a function:

```
Listing 1: power_mod.ads
```

```ada
pragma Ada_2022;
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
```

Learning Ada

function Power_Mod (M, D, N : Big_Integer) return Big_Integer is

  function Is_Odd (X : Big_Integer) return Boolean is
    (X mod 2 /= 0);

  Result : Big_Integer := 1;
  Exp     : Big_Integer := D;
  Mult    : Big_Integer := M mod N;
begin
  while Exp /= 0 loop
    -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
    if Is_Odd (Exp) then
      Result := (Result * Mult) mod N;
    end if;
    Mult := Mult ** 2 mod N;
    Exp := Exp / 2;
  end loop;

  return Result;
end Power_Mod;

Let's check this with the example from Wikipedia. In that example, the public key is $n = 3233, e = 17$ and the message is $m = 65$. The encrypted message is $m^e \mod n = 65^{17} \mod 3233 = 2790 = c$.

```ada
Ada.Text_IO.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
```

2790

To decrypt it with the public key $n = 3233, d = 413$, we need to calculate $c^d \mod n = 2790^{413} \mod 3233$:

```ada
Ada.Text_IO.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);
```

65

So 65 is the original message $m$. Easy!

Here is the complete code snippet:

```
pragma Ada_2022;
with Ada.Text_IO;
```

305 https://en.wikipedia.org/wiki/RSA_(cryptosystem)
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;

procedure Main is
  -- Calculate M ** D mod N
  function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
    function Is_Odd (X : Big_Integer) return Boolean is
      Result : Big_Integer := 1;
      Exp    : Big_Integer := D;
      Mult   : Big_Integer := M mod N;
      begin
        while Exp /= 0 loop
          -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
          if Is_Odd (Exp) then
            Result := (Result * Mult) mod N;
          end if;
          Mult := Mult ** 2 mod N;
          Exp := Exp / 2;
        end loop;
        return Result;
      end Power_Mod;
      begin
        Ada.Text_IO.Put_Line (Big_Integer'Image (2 ** 256));
        Ada.Text_IO.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
        Ada.Text_IO.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);
      end Main;

Code block metadata

Project: Courses.Ada_2022_Whats_New.Big_Numbers_Tiny_RSA
MD5: 6178da9d6998db6d51f31fd5c7cc5391

Runtime output

1157920892371619542357098508687907853269984665640564039457584007913129639936
2790
65

50.3 Big Reals

In addition to Big_Integer, Ada 2022 provides Big Reals\(^\text{306}\).

50.4 References

- ARM A.5.6 Big Integers\textsuperscript{307}
- ARM A.5.7 Big Reals\textsuperscript{308}
- AI12-0208-1\textsuperscript{309}

\textsuperscript{309} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0208-1.TXT
INTERFACING C VARIADIC FUNCTIONS

Note: Variadic convention is supported by
- GNAT Community Edition 2020
- GCC 11

In C, variadic functions\(^\text{310}\) take a variable number of arguments and an ellipsis as the last parameter of the declaration. A typical and well-known example is:

```c
int printf (const char* format, ...);
```

Usually, in Ada, we bind such a function with just the parameters we want to use:

```ada
procedure printf_double
  (format : Interfaces.C.char_array;
   value : Interfaces.C.double)
with Import, Convention => C,
    External_Name => "printf";
```

Then we call it as a normal Ada function:

```ada
printf_double (Interfaces.C.To_C ("Pi=%f"), Ada.Numerics.pi);
```

Unfortunately, doing it this way doesn't always work because some ABI\(^\text{311}\)'s use different calling conventions for variadic functions. For example, the AMD64 ABI\(^\text{312}\) specifies:

- `%rax` — with variable arguments passes information about the number of vector registers used;
- `%xmm0–%xmm1` — used to pass and return floating point arguments.

This means, if we write (in C):

```c
printf("%d", 5);
```

The compiler will place 0 into `%rax`, because we don't pass any float argument. But in Ada, if we write:

```ada
procedure printf_int
  (format : Interfaces.C.char_array;
   value : Interfaces.C.int)
with Import, Convention => C,
```

(continues on next page)

\(^{310}\) https://en.cppreference.com/w/c/variadic
\(^{311}\) https://en.wikipedia.org/wiki/Application_binary_interface
learning Ada

(continued from previous page)

External_Name => "printf";
printf_int (Interfaces.C.To_C ("d=%d"), 5);

the compiler won't use the %rax register at all. (You can't include any float argument because there's no float parameter in the Ada wrapper function declaration.) As result, you will get a crash, stack corruption, or other undefined behavior.

To fix this, Ada 2022 provides a new family of calling convention names — C_Variadic_N:

The convention C_Variadic_n is the calling convention for a variadic C function taking n fixed parameters and then a variable number of additional parameters.

Therefore, the correct way to bind the printf function is:

procedure printf_int
    (format : Interfaces.C.char_array;
    value : Interfaces.C.int)
with Import,
    Convention => C_Variadic_1,
    External_Name => "printf";

And the following call won't crash on any supported platform:

printf_int (Interfaces.C.To_C ("d=%d"), 5);

Without this convention, problems cause by this mismatch can be very hard to debug. So, this is a very useful extension to the Ada-to-C interfacing facility.

Here is the complete code snippet:

Listing 1: main.adb

with Interfaces.C;

procedure Main is
    procedure printf_int
        (format : Interfaces.C.char_array;
        value : Interfaces.C.int)
with Import,
    Convention => C_Variadic_1,
    External_Name => "printf";
begin
    printf_int (Interfaces.C.To_C ("d=%d"), 5);
end Main;

Code block metadata

MD5: 94515f55a93f27e4f4ecec31256645d9

1270 Chapter 51. Interfacing C variadic functions
51.1 References

- ARM B.3 Interfacing with C and C++.\textsuperscript{313}
- AI12-0028-1\textsuperscript{314}

\textsuperscript{313} http://www.ada-auth.org/standards/22aarm/html/AA-B-3.html
\textsuperscript{314} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0028-1.TXT
Part VI

Ada for the C++ or Java Developer
This document will present the Ada language using terminology and examples that are familiar to developers that understand the C++ or Java languages.

This document was prepared by Quentin Ochem, with contributions and review from Richard Kenner, Albert Lee, and Ben Brosgol.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

**Note:** Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn web-site. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- **Project:** Courses.Intro_to_Ada.Imperative_Language.Greet
- **MD5:** cba89a34b87c9d715c33d98205e6ab

you will find it in this directory:

```
projects/Courses/Intro_to_Ada/Imperative_Language/Greet/cba89a34b87c9d715c33d98205e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).
Nowadays it seems like talking about programming languages is a bit passé. The technical wars of the past decade have subsided and today we see a variety of high-level and well-established languages offering functionality that can meet the needs of any programmer.

Python, Java, C++, C#, and Visual Basic are recent examples. Indeed, these languages make it easier to write code very quickly, are very flexible, offer features with highly dynamic behavior, and some even allow compilers to deduce the developer's probable intent.

Why, then, talk about yet another language? Well, by addressing the general programming market, the aforementioned languages have become poorly suited for working within the domain of high-integrity systems. In highly reliable, secure and safe applications such as those found in and around airplanes, rockets, satellites, trains, and in any device whose failure could jeopardize human life or critical assets, the programming languages used must support the high standard of software engineering necessary to maintain the integrity of the system.

The concept of verification — the practice of showing that the system behaves and performs as intended — is key in such environments. Verification can be accomplished by some combination of review, testing, static analysis, and formal proof techniques. The increasing reliance on software and increasing complexity of today's systems has made this task more difficult. Technologies and practices that might have been perfectly acceptable ten or fifteen years ago are insufficient today. Thankfully, the state of the art in analysis and proof tools and techniques has also advanced.

The latest revisions of the Ada language, Ada 2005 and Ada 2012, make enhanced software integrity possible. From its inception in the 1980s, Ada was designed to meet the requirements of high-integrity systems, and continues to be well-suited for the implementation of critical embedded or native applications. And it has been receiving increased attention recently. Every language revision has enhanced expressiveness in many areas. Ada 2012, in particular, has introduced new features for contract-based programming that are valuable to any project where verification is part of the engineering lifecycle. Along with these language enhancements, Ada compiler and tool technology has also kept pace with general computing developments over the past few years. Ada development environments are available on a wide range of platforms and are being used for the most demanding applications.

It is no secret that we at AdaCore are very enthusiastic about Ada, but we will not claim that Ada is always the solution; Ada is no more a silver bullet than any other language. In some domains other languages make sense because of the availability of particular libraries or development frameworks. For example, C++ and Java are considered good choices for desktop programs or applications where a shortened time to market is a major objective. Other areas, such as website programming or system administration, tend to rely on different formalisms such as scripting and interpreted languages. The key is to select the proper technical approach, in terms of the language and tools, to meet the requirements. Ada's strength is in areas where reliability is paramount.

Learning a new language shouldn't be complicated. Programming paradigms have not evolved much since object oriented programming gained a foothold, and the same
paradigms are present one way or another in many widely used languages. This document will thus give you an overview of the Ada language using analogies to C++ and Java — these are the languages you’re already likely to know. No prior knowledge of Ada is assumed. If you are working on an Ada project now and need more background, if you are interested in learning to program in Ada, or if you need to perform an assessment of possible languages to be used for a new development, this guide is for you.
Ada implements the vast majority of programming concepts that you're accustomed to in C++ and Java: classes, inheritance, templates (generics), etc. Its syntax might seem peculiar, though. It's not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a simpler language — its syntax favors making it easier to conceptualize and read program code, rather than making it faster to write in a cleverly condensed manner. For example, full words like `begin` and `end` are used in place of curly braces. Conditions are written using `if`, `then`, `elsif`, `else`, and `end if`. Ada's assignment operator does not double as an expression, smoothly eliminating any frustration that could be caused by `=` being used where `==` should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens `--` mark the start of a comment that continues to the end of the line. This is exactly the same as using `//` for comments in C++ and Java. There is no equivalent of `/*` ... `*/` block comments in Ada; use multiple `--` lines instead.

Ada compilers are stricter with type and range checking than most C++ and Java programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding more creatively, but this helps detect problems and vulnerabilities at compile time — early on in the development cycle. In addition, dynamic checks (such as array bounds checks) provide verification that could not be done at compile time. Dynamic checks are performed at run time, similar to what is done in Java.

Ada identifiers and reserved words are case insensitive. The identifiers `VAR`, `var` and `VaR` are treated as the same; likewise `begin`, `BEGIN`, `Begin`, etc. Language-specific characters, such as accents, Greek or Russian letters, and Asian alphabets, are acceptable to use. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:
Ada is designed to be portable. Ada compilers must follow a precisely defined international (ISO) standard language specification with clearly documented areas of vendor freedom where the behavior depends on the implementation. It's possible, then, to write an implementation-independent application in Ada and to make sure it will have the same effect across platforms and compilers.

Ada is truly a general purpose, multiple paradigm language that allows the programmer to employ or avoid features like run-time contract checking, tasking, object oriented programming, and generics. Efficiently programmed Ada is employed in device drivers, interrupt handlers, and other low-level functions. It may be found today in devices with tight limits on processing speed, memory, and power consumption. But the language is also used for programming larger interconnected systems running on workstations, servers, and supercomputers.
COMPILATION UNIT STRUCTURE

C++ programming style usually promotes the use of two distinct files: header files used to define specifications (.h*, .hxx, .hpp), and implementation files which contain the executable code (.c, .cxx, .cpp). However, the distinction between specification and implementation is not enforced by the compiler and may need to be worked around in order to implement, for example, inlining or templates.

Java compilers expect both the implementation and specification to be in the same .java file. (Yes, design patterns allow using interfaces to separate specification from implementation to a certain extent, but this is outside of the scope of this description.)

Ada is superficially similar to the C++ case: Ada compilation units are generally split into two parts, the specification and the body. However, what goes into those files is more predictable for both the compiler and for the programmer. With GNAT, compilation units are stored in files with a .ads extension for specifications and with a .adb extension for implementations.

Without further ado, we present the famous "Hello World" in three languages:

[Ada]

```ada
with Ada.Text_IO;
use Ada.Text_IO;

procedure Main is
begin
  Put_Line ("Hello World");
end Main;
```

[C++]

```cpp
#include <iostream>
using namespace std;

int main(int argc, const char* argv[]) {
  cout << "Hello World" << endl;
}
```

[Java]

```java
public class Main {
  public static void main(String [] argv) {
    System.out.println ("Hello World");
  }
}
```

The first line of Ada we see is the `with` clause, declaring that the unit (in this case, the Main subprogram) will require the services of the package Ada.Text_I0. This is different from how `#include` works in C++ in that it does not, in a logical sense, copy/paste the code of Ada.Text_I0 into Main. The `with` clause directs the compiler to make the public
interface of the Ada.Text_IO package visible to code in the unit (here Main) containing the with clause. Note that this construct does not have a direct analog in Java, where the entire CLASSPATH is always accessible. Also, the name Main for the main subprogram was chosen for consistency with C++ and Java style but in Ada the name can be whatever the programmer chooses.

The use clause is the equivalent of using namespace in C++, or import in Java (though it wasn't necessary to use import in the Java example above). It allows you to omit the full package name when referring to with'ed units. Without the use clause, any reference to Ada.Text_IO items would have had to be fully qualified with the package name. The Put_Line line would then have read:

```
Ada.Text_IO.Put_Line ("Hello World");
```

The word "package" has different meanings in Ada and Java. In Java, a package is used as a namespace for classes. In Ada, it's often a compilation unit. As a result Ada tends to have many more packages than Java. Ada package specifications ("package specs" for short) have the following structure:

```ada
package Package_Name is
    -- public declarations
private
    -- private declarations
end Package_Name;
```

The implementation in a package body (written in a .adb file) has the structure:

```ada
package body Package_Name is
    -- implementation
end Package_Name;
```

The private reserved word is used to mark the start of the private portion of a package spec. By splitting the package spec into private and public parts, it is possible to make an entity available for use while hiding its implementation. For instance, a common use is declaring a record (Ada's struct) whose fields are only visible to its package and not to the caller. This allows the caller to refer to objects of that type, but not to change any of its contents directly.

The package body contains implementation code, and is only accessible to outside code through declarations in the package spec.

An entity declared in the private part of a package in Ada is roughly equivalent to a protected member of a C++ or Java class. An entity declared in the body of an Ada package is roughly equivalent to a private member of a C++ or Java class.
55.1 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

[Ada]

```ada
-- Ada program to declare and modify Integers

procedure Main is
  -- Variable declarations
  A, B : Integer := 0;
  C : Integer := 100;
  D : Integer;
begin
  -- Ada uses a regular assignment statement for incrementation.
  A := A + 1;
  -- Regular addition
  D := A + B + C;
end Main;
```

[C++]

```cpp
/*
 * C++ program to declare and modify ints
 */
int main(int argc, const char* argv[]) {
  // Variable declarations
  int a = 0, b = 0, c = 100, d;

  // C++ shorthand for incrementation
  a++;

  // Regular addition
  d = a + b + c;
}
```

[Java]

```java
/*
 * Java program to declare and modify ints
 */
public class Main {
```

(continues on next page)
Statements are terminated by semicolons in all three languages. In Ada, blocks of code are surrounded by the reserved words `begin` and `end` rather than by curly braces. We can use both multi-line and single-line comment styles in the C++ and Java code, and only single-line comments in the Ada code.

Ada requires variable declarations to be made in a specific area called the declarative part, seen here before the `begin` keyword. Variable declarations start with the identifier in Ada, as opposed to starting with the type as in C++ and Java (also note Ada's use of the `:` separator). Specifying initializers is different as well: in Ada an initialization expression can apply to multiple variables (but will be evaluated separately for each), whereas in C++ and Java each variable is initialized individually. In all three languages, if you use a function as an initializer and that function returns different values on every invocation, each variable will get initialized to a different value.

Let's move on to the imperative statements. Ada does not provide `++` or `--` shorthand expressions for increment/decrement operations; it is necessary to use a full assignment statement. The `:=` symbol is used in Ada to perform value assignment. Unlike C++'s and Java's `=` symbol, `:=` can not be used as part of an expression. So, a statement like \( A := B := C; \) doesn't make sense to an Ada compiler, and neither does a clause like `if A := B then ....` Both are compile-time errors.

You can nest a block of code within an outer block if you want to create an inner scope:
55.2 Conditions

The use of the if statement:

[Ada]

```ada
if Variable > 0 then
  Put_Line (" > 0 ");
elsiif Variable < 0 then
  Put_Line (" < 0 ");
else
  Put_Line (" = 0 ");
end if;
```

[C++]

```cpp
if (Variable > 0)
  cout << "> 0 " << endl;
elsi if (Variable < 0)
  cout << " < 0 " << endl;
elsi
  cout << " = 0 " << endl;
```

[Java]

```java
if (Variable > 0)
  System.out.println (" > 0 ");
elsi if (Variable < 0)
  System.out.println (" < 0 ");
elsi
  System.out.println (" = 0 ");
```

In Ada, everything that appears between the if and then keywords is the conditional expression — no parentheses required. Comparison operators are the same, except for equality (=) and inequality (/=). The English words not, and, and or replace the symbols !, &, and |, respectively, for performing boolean operations.

It's more customary to use && and || in C++ and Java than & and | when writing boolean expressions. The difference is that && and || are short-circuit operators, which evaluate terms only as necessary, and & and | will unconditionally evaluate all terms. In Ada, and and or will evaluate all terms; and then and or else direct the compiler to employ short circuit evaluation.

Here are what switch/case statements look like:

[Ada]

```ada
case Variable is
  when 0 =>
    Put_Line ("Zero");
  when 1 .. 9 =>
    Put_Line ("Positive Digit");
  when 10 | 12 | 14 | 16 | 18 =>
    Put_Line ("Even Number between 10 and 18");
  when others =>
    Put_Line ("Something else");
end case;
```

[C++]

```cpp
switch (Variable) {
  case 0:
    // (continues on next page)
```
cout << "Zero" << endl;
break;
case 1: case 2: case 3: case 4: case 5:
case 6: case 7: case 8: case 9:
cout << "Positive Digit" << endl;
break;
case 10: case 12: case 14: case 16: case 18:
cout << "Even Number between 10 and 18" << endl;
break;
default:
cout << "Something else";
}

[Java]
switch (Variable) {
    case 0:
        System.out.println ("Zero");
        break;
    case 1: case 2: case 3: case 4: case 5:
    case 6: case 7: case 8: case 9:
        System.out.println ("Positive Digit");
        break;
    case 10: case 12: case 14: case 16: case 18:
        System.out.println ("Even Number between 10 and 18");
        break;
    default:
        System.out.println ("Something else");
}

In Ada, the case and end case lines surround the whole case statement, and each case starts with when. So, when programming in Ada, replace switch with case, and replace case with when.

Case statements in Ada require the use of discrete types (integers or enumeration types), and require all possible cases to be covered by when statements. If not all the cases are handled, or if duplicate cases exist, the program will not compile. The default case, default: in C++ and Java, can be specified using when others => in Ada.

In Ada, the break instruction is implicit and program execution will never fall through to subsequent cases. In order to combine cases, you can specify ranges using .. and enumerate disjoint values using | which neatly replaces the multiple case statements seen in the C++ and Java versions.

### 55.3 Loops

In Ada, loops always start with the loop reserved word and end with end loop. To leave the loop, use exit — the C++ and Java equivalent being break. This statement can specify a terminating condition using the exit when syntax. The loop opening the block can be preceded by a while or a for.

The while loop is the simplest one, and is very similar across all three languages:

[Ada]

```ada
while Variable < 10_000 loop
    Variable := Variable * 2;
end loop;
```
Ada’s `for` loop, however, is quite different from that in C++ and Java. It always increments or decrements a loop index within a discrete range. The loop index (or “loop parameter” in Ada parlance) is local to the scope of the loop and is implicitly incremented or decremented at each iteration of the loop statements; the program cannot directly modify its value. The type of the loop parameter is derived from the range. The range is always given in ascending order even if the loop iterates in descending order. If the starting bound is greater than the ending bound, the interval is considered to be empty and the loop contents will not be executed. To specify a loop iteration in decreasing order, use the `reverse` reserved word. Here are examples of loops going in both directions:

**Ada**

```ada
-- Outputs 0, 1, 2, ..., 9
for Variable in 0 .. 9 loop
  Put_Line (Integer'Image (Variable));
end loop;

-- Outputs 9, 8, 7, ..., 0
for Variable in reverse 0 .. 9 loop
  Put_Line (Integer'Image (Variable));
end loop;
```

**C++**

```cpp
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
  cout << Variable << endl;
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
  cout << Variable << endl;
}
```

**Java**

```java
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
  System.out.println (Variable);
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
  System.out.println (Variable);
}
```

Ada uses the `Integer` type's `Image` attribute to convert a numerical value to a String. There is no implicit conversion between `Integer` and `String` as there is in C++ and Java. We’ll have a more in-depth look at such attributes later on.

It’s easy to express iteration over the contents of a container (for instance, an array, a list,
or a map) in Ada and Java. For example, assuming that Int_List is defined as an array of Integer values, you can use:

[Ada]

```ada
for I of Int_List loop  
   Put_Line (Integer'Image (I));
end loop;
```

[Java]

```java
for (int i : Int_List) {
   System.out.println (i);
}
```
56.1 Strong Typing

One of the main characteristics of Ada is its strong typing (i.e., relative absence of implicit type conversions). This may take some getting used to. For example, you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

```ada
procedure Strong_Typing is
    Alpha : Integer := 1;
    Beta  : Integer := 10;
    Result : Float;
begin
    Result := Float (Alpha) / Float (Beta);
end Strong_Typing;
```

[C++]

```cpp
void weakTyping () {
    int alpha = 1;
    int beta = 10;
    float result;

    result = alpha / beta;
}
```

[Java]

```java
void weakTyping () {
    int alpha = 1;
    int beta = 10;
    float result;

    result = alpha / beta;
}
```

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from Integer to Float explicit. In fact it significantly changes the behavior of the computation. While the Ada code performs a floating point operation 1.0 / 10.0 and stores 0.1 in Result, the C++ and Java versions instead store 0.0 in result. This is because the C++ and Java versions perform an integer operation between two integer variables: 1 / 10 is 0. The result of the integer division is then converted to a float and stored. Errors of this sort can be very hard to locate in complex
Learning Ada

pieces of code, and systematic specification of how the operation should be interpreted helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to Float:

```
-- Perform an Integer division then convert to Float
Result := Float (Alpha / Beta);
```

In Ada, a floating point literal must be written with both an integral and decimal part. 10 is not a valid literal for a floating point value, while 10.0 is.

56.2 Language-Defined Types

The principal scalar types predefined by Ada are **Integer**, **Float**, **Boolean**, and **Character**. These correspond to **int**, **float**, **bool/boolean**, and **char**, respectively. The names for these types are not reserved words; they are regular identifiers.

56.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators +, -, *, and / for integer and floating-point types take operands of the same type and return a value of that type.

```ada
procedure Main is
  type Distance is new Float;
  type Area is new Float;
  D1 : Distance := 2.0;
  D2 : Distance := 3.0;
  A : Area;
begin
  D1 := D1 + D2;  -- OK
  D1 := D1 + A;   -- NOT OK: incompatible types for "+" operator
  A := D1 * D2;   -- NOT OK: incompatible types for "*" assignment
  A := Area (D1 * D2);  -- OK
end Main;
```

Even though the Distance and Area types above are just **Floats**, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two Distances yields a Distance) and prohibit some useful cases (for example multiplying two Distances should deliver an Area). These situations can be handled through other mechanisms. A predefined operation can be identified as **abstract** to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its
operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

Ada enumerations work similarly to C++ and Java's `enum`.

**[Ada]**

```ada
type Day is
  (Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday);
```

**[C++]**

```cpp
enum Day {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday};
```

**[Java]**

```java
enum Day {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday}
```

But even though such enumerations may be implemented using a machine word, at the language level Ada will not confuse the fact that Monday is a Day and is not an `Integer`. You can compare a Day with another Day, though. To specify implementation details like the numeric values that correspond with enumeration values in C++ you include them in the original `enum` statement:

**[C++]**

```cpp
enum Day {
  Monday = 10,
  Tuesday = 11,
  Wednesday = 12,
  Thursday = 13,
  Friday = 14,
  Saturday = 15,
  Sunday = 16};
```

But in Ada you must use both a type definition for Day as well as a separate `representation clause` for it like:

**[Ada]**

```ada
for Day use
  (Monday => 10,
   Tuesday => 11,
   Wednesday => 12,
   Thursday => 13,
   Friday => 14,
   Saturday => 15,
   Sunday => 16);
```

(continues on next page)
56.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a range constraint introduced with the range reserved word, for example:

```ada
procedure Main is
  type Grade is range 0 .. 100;
  G1, G2 : Grade;
  N : Integer;
begin
  ... -- Initialization of N
  G1 := 80; -- OK
  G1 := N; -- Illegal (type mismatch)
  G1 := Grade (N); -- Legal, run-time range check
  G2 := G1 + 10; -- Legal, run-time range check
  G1 := (G1 + G2)/2; -- Legal, run-time range check
end Main;
```

In the above example, Grade is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to G1 is correct and will not raise a run-time exception. Assigning N to G1 is illegal since Grade is a different type than Integer. Converting N to Grade makes the assignment legal, and a range check on the conversion confirms that the value is within 0 .. 100. Assigning G1+10 to G2 is legal since + for Grade returns a Grade (note that the literal 10 is interpreted as a Grade value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression G1 + G2 may be outside the range of Grade, but the final result will be in range. Nevertheless, depending on the representation chosen for Grade, the addition may overflow. If the compiler represents Grade values as signed 8-bit integers (i.e., machine numbers in the range -128 .. 127) then the sum G1+G2 may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

```ada
G1 := Grade ((Integer (G1) + Integer (G2)) / 2);
```

Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the Day example, one can write:

```ada
type Business_Day is new Day range Monday .. Friday;
type Weekend_Day is new Day range Saturday .. Sunday;
```
Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind subtypes in Ada. A subtype is a type with optional additional constraints. For example:

```ada
subtype Business_Day is Day range Monday .. Friday;
subtype Weekend_Day is Day range Saturday .. Sunday;
subtype Dice_Throw is Integer range 1 .. 6;
```

These declarations don't create new types, just new names for constrained ranges of their base types.

### 56.5 Generalized Type Contracts: Subtype Predicates

Range checks are a special form of type contracts; a more general method is provided by Ada subtype predicates, introduced in Ada 2012. A subtype predicate is a boolean expression defining conditions that are required for a given type or subtype. For example, the Dice_Throw subtype shown above can be defined in the following way:

```ada
subtype Dice_Throw is Integer
    with Dynamic_Predicate => Dice_Throw in 1 .. 6;
```

The clause beginning with `with` introduces an Ada aspect, which is additional information provided for declared entities such as types and subtypes. The Dynamic_Predicate aspect is the most general form. Within the predicate expression, the name of the (sub)type refers to the current value of the (sub)type. The predicate is checked on assignment, parameter passing, and in several other contexts. There is a Static_Predicate form which introduce some optimization and constrains on the form of these predicates, outside of the scope of this document.

Of course, predicates are useful beyond just expressing ranges. They can be used to represent types with arbitrary constraints, in particular types with discontinuities, for example:

```ada
type Not_Null is new Integer
    with Dynamic_Predicate => Not_Null /= 0;
type Even is new Integer
    with Dynamic_Predicate => Even mod 2 = 0;
```

### 56.6 Attributes

Attributes start with a single apostrophe (“tick”), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The `Image` and `Value` attributes allow you to transform a scalar value into a `String` and vice-versa. For example:

```ada
declare
    A : Integer := 99;
begin
    Put_Line (Integer'Image (A));
```
A := Integer'Value ("99");
end;

Certain attributes are provided only for certain kinds of types. For example, the 'Val and 'Pos attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]
declare
  C : Character := 'a';
begin
  C := Character'Val (Character'Pos (C) + 1);
end;

A more concise way to get the next value in Ada is to use the 'Succ attribute:

[Ada]
declare
  C : Character := 'a';
begin
  C := Character'Succ (C);
end;

You can get the previous value using the 'Pred attribute. Here is the equivalent in C++ and Java:

[C++]
char c = 'a';
c++;

[Java]
char c = 'a';
c++;

Other interesting examples are the 'First and 'Last attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, Integer'First returns \(-2^{31}\) and Integer'Last returns \(2^{31} - 1\).

56.7 Arrays and Strings

C++ arrays are pointers with offsets, but the same is not the case for Ada and Java. Arrays in the latter two languages are not interchangeable with operations on pointers, and array types are considered first-class citizens. Arrays in Ada have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds — the lower bound doesn't necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here's an example of declaring an array of 26 characters, initializing the values from 'a' to 'z':

[Ada]
declare
type Arr_Type is array (Integer range <>) of Character;
(continues on next page)
In C++ and Java, only the size of the array is given during declaration. In Ada, array index ranges are specified using two values of a discrete type. In this example, the array type declaration specifies the use of Integer as the index type, but does not provide any constraints (use \(<\), pronounced box, to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called 'Range. In our example, Arr'Range can also be expressed as Arr'First .. Arr'Last; both expressions will resolve to 1 .. 26. So the 'Range attribute supplies the bounds for our for loop. There is no risk of stating either of the bounds incorrectly, as one might do in C++ where \( I \leq 26 \) may be specified as the end-of-loop condition.

As in C++, Ada Strings are arrays of Characters. The C++ or Java String class is the equivalent of the Ada type Ada.Strings.Unbounded_String which offers additional capabilities in exchange for some overhead. Ada strings, importantly, are not delimited with the special character '\0' like they are in C++. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined String type is very straightforward to use:

```
My_String : String (1 .. 26);
```

Unlike C++ and Java, Ada does not offer escape sequences such as '\n'. Instead, explicit values from the ASCII package must be concatenated (via the concatenation operator, \&). Here for example, is how to initialize a line of text ending with a new line:

```
My_String : String := "This is a line with a end of line" & ASCII.LF;
```

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of My_String's bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C++ or Java, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada,
the actual array contents are duplicated. To get the above behavior, actual pointer types
would have to be defined and used.

[Ada]

```ada
declare
type Arr_Type is array (Integer range <>) of Integer;
A1 : Arr_Type (1 .. 2);
A2 : Arr_Type (1 .. 2);
begin
A1 (1) := 0;
A1 (2) := 1;
A2 := A1;
end;
```

[C++]

```cpp
int A1 [2];
int A2 [2];
A1 [0] = 0;
A1 [1] = 1;
for (int i = 0; i < 2; ++i) {
    A2 [i] = A1 [i];
}
```

[Java]

```java
int [] A1 = new int [2];
int [] A2 = new int [2];
A1 [0] = 0;
A1 [1] = 1;
A2 = Arrays.copyOf(A1, A1.length);
```

In all of the examples above, the source and destination arrays must have precisely
the same number of elements. Ada allows you to easily specify a portion, or slice, of an array.
So you can write the following:

[Ada]

```ada
declare
type Arr_Type is array (Integer range <>) of Integer;
A1 : Arr_Type (1 .. 10);
A2 : Arr_Type (1 .. 5);
begin
A2 (1 .. 3) := A1 (4 .. 6);
end;
```

This assigns the 4th, 5th, and 6th elements of A1 into the 1st, 2nd, and 3rd elements of A2.
Note that only the length matters here: the values of the indexes don't have to be equal;
they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as
opposed to their addresses:

[Ada]

```ada
declare
type Arr_Type is array (Integer range <>) of Integer;
```
A1 : Arr_Type (1 .. 2);
A2 : Arr_Type (1 .. 2);
begin
  if A1 = A2 then
  
[C++]
  int A1 [2];
  int A2 [2];
  bool eq = true;
  for (int i = 0; i < 2; ++i) {
    if (A1 [i] != A2 [i]) {
      eq = false;
    }
  }
  if (eq) {

[Java]
  int [] A1 = new int [2];
  int [] A2 = new int [2];
  if (Arrays.equals (A1, A2)) {

You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the others expression to define a default value for all the unspecified array elements. Therefore, you can write:

declare
  type Arr_Type is array (Integer range <>) of Integer;
  A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
  A2 : Arr_Type (-2 .. 42) := (others => 0);
begin
  A1 := (1, 2, 3, others => 10);
  -- use a slice to assign A2 elements 11 .. 19 to 1
  A2 (11 .. 19) := (others => 1);
end;

56.8 Heterogeneous Data Structures

In Ada, there's no distinction between struct and class as there is in C++. All heterogeneous data structures are records. Here are some simple records:

[Ada]

declare
  type R is record
    A, B : Integer;
    C : Float;
  end record;

  V : R;

56.8. Heterogeneous Data Structures
Learning Ada

begin
  V.A := 0;
end;

[C++]

struct R {
  int A, B;
  float C;
};

R V;
V.A = 0;

[Java]

class R {
  public int A, B;
  public float C;
}

R V = new R();
V.A = 0;

Ada allows specification of default values for fields just like C++ and Java. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by others => <> to specify that fields not listed will take their default values. For example:

type R is record
  A, B : Integer := 0;
  C : Float := 0.0;
end record;

V1 : R := (1, 2, 1.0);
V2 : R := (A => 1, B => 2, C => 1.0);
V3 : R := (C => 1.0, A => 1, B => 2);
V4 : R := (C => 1.0, others => <>);

56.9 Pointers

Pointers, references, and access types differ in significant ways across the languages that we are examining. In C++, pointers are integral to a basic understanding of the language, from array manipulation to proper declaration and use of function parameters. In Java, direct pointer manipulation is abstracted by the Java runtime. And in Ada, direct pointer manipulation is possible, but unlike C++, they are not required for basic usage with arrays and parameter passing.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

declare
  type R is record
    A, B : Integer;
  end record;
There's a fundamental difference between the Ada and C++ semantics above and the semantics for Java. In Ada and C++, objects are allocated on the stack and are directly accessed. \( V_1 \) and \( V_2 \) are two different objects and the assignment statement copies the value of \( V_1 \) into \( V_2 \). In Java, \( V_1 \) and \( V_2 \) are two references to objects of class \( R \). Note that when \( V_1 \) and \( V_2 \) are declared, no actual object of class \( R \) yet exists in memory: it has to be allocated later with the \texttt{new} allocator operator. After the assignment \( V_2 = V_1 \), there's only one \( R \) object in memory: the assignment is a reference assignment, not a value assignment. At the end of the Java code, \( V_1 \) and \( V_2 \) are two references to the same objects and the \( V_2.A = 1 \) statement changes the field of that one object, while in the Ada and the C++ case \( V_1 \) and \( V_2 \) are two distinct objects.

To obtain similar behavior in Ada, you can use pointers. It can be done through Ada's \texttt{access} type:

\begin{verbatim}
declarerecord
   type R is record
      A, B : Integer;
   end record;
   type R_Access is access R;
end record;

V1 : R_Access;
V2 : R_Access;
begin
   V1 := new R;
   V1.A := 0;
   V2 := V1;
   V2.A := 1;
end;
\end{verbatim}
Learning Ada

```ada
struct R {
    int A, B;
};

R * V1, * V2;
V1 = new R ();
V1->A = 0;
V2 = V1;
V2->A = 0;
```

For those coming from the Java world: there's no garbage collector in Ada, so objects allocated by the `new` operator need to be expressly freed.

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append `.all`. The equivalent of `V1->A` in C++ can be written either as `V1.A` or `V1.all.A`.

Pointers to scalar objects in Ada and C++ look like:

**Ada**

```ada
procedure Main is
    type A_Int is access Integer;
    Var : A_Int := new Integer;
begin
    Var.all := 0;
end Main;
```

**C++**

```cpp
int main (int argc, char *argv[]) {
    int * Var = new int;
    *Var = 0;
}
```

An initializer can be specified with the allocation by appending `(value)`:  

```ada
Var : A_Int := new Integer'(0);
```

When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being `aliased`. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either `access all` (if the referenced object needs to be assigned to) or `access constant` (if the referenced object is a constant). The `Access` attribute works like the C++ `&` operator to get a pointer to the object, but with a "scope accessibility" check to prevent references to objects that have gone out of scope. For example:

**Ada**

```ada
type A_Int is access all Integer;
Var : aliased Integer;
Ptr : A_Int := Var'Access;
```

**C++**

```cpp
int Var;
int * Ptr = &Var;
```

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs — it's called Ada.Unchecked_Deallocation. To create your customized
deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

[Ada]

```
with Ada.Unchecked_Deallocation;
procedure Main is
  type Integer_Access is access all Integer;
  procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);
  My_Pointer : Integer_Access := new Integer;
begin
  Free (My_Pointer);
end Main;
```

[C++]

```
int main (int argc, char *argv[]) {
  int * my_pointer = new int;
  delete my_pointer;
}
```
CHAPTER
FIFTYSEVEN

FUNCTIONS AND PROCEDURES

57.1 General Form

Subroutines in C++ and Java are always expressed as functions (methods) which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term "subprogram" to refer to both functions and procedures.

Parameters can be passed in three distinct modes: in, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram. out is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller. in_out is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant reference in C++). Ada also provides access parameters, in effect an explicit pass-by-reference indicator.

In Ada the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). (There are some exceptions to the "in general". For example, parameters of scalar types are always passed by copy, for all three modes.) C++ has the programmer specify how to pass the parameter, and Java forces primitive type parameters to be passed by copy and all other parameters to be passed by reference. For this reason, a 1:1 mapping between Ada and Java isn't obvious but here's an attempt to show these differences:

[Ada]

```ada
procedure Proc
(Var1 : Integer;
 Var2 : out Integer;
 Var3 : in out Integer);

function Func (Var : Integer) return Integer;

procedure Proc
(Var1 : Integer;
 Var2 : out Integer;
 Var3 : in out Integer)
is
begin
  Var2 := Func (Var1);
  Var3 := Var3 + 1;
end Proc;

function Func (Var : Integer) return Integer
is
begin
  return Var + 1;
end Func;
```
The first two declarations for Proc and Func are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C++, a function that has not yet been seen cannot be used. Here, Proc can call Func because its specification has been declared. In Java, it's fine to have the declaration of the subprogram later.

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax, including default values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.
57.2 Overloading

Different subprograms may share the same name; this is called "overloading." As long as the subprogram signatures (subprogram name, parameter types, and return types) are different, the compiler will be able to resolve the calls to the proper destinations. For example:

```ada
function Value (Str : String) return Integer;
function Value (Str : String) return Float;
```

```ada
V : Integer := Value ("8");
```

The Ada compiler knows that an assignment to V requires an `Integer`. So, it chooses the `Value` function that returns an `Integer` to satisfy this requirement.

Operators in Ada can be treated as functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To express an operator as a function, enclose it in quotes:

[Ada]
```
function "=" (Left : Day; Right : Integer) return Boolean;
```

[C++]
```
bool operator = (Day Left, int Right);
```

57.3 Subprogram Contracts

You can express the expected inputs and outputs of subprograms by specifying subprogram contracts. The compiler can then check for valid conditions to exist when a subprogram is called and can check that the return value makes sense. Ada allows defining contracts in the form of Pre and Post conditions; this facility was introduced in Ada 2012. They look like:

```ada
function Divide (Left, Right : Float) return Float
  with Pre => Right /= 0.0,
       Post => Divide'Result * Right < Left + 0.0001
             and then Divide'Result * Right > Left - 0.0001;
```

The above example adds a Pre condition, stating that Right cannot be equal to 0.0. While the IEEE floating point standard permits divide-by-zero, you may have determined that use of the result could still lead to issues in a particular application. Writing a contract helps to detect this as early as possible. This declaration also provides a Post condition on the result.

Postconditions can also be expressed relative to the value of the input:

```ada
procedure Increment (V : in out Integer)
  with Pre => V < Integer'Last,
       Post => V = V'Old + 1;
```

`V'Old` in the postcondition represents the value that V had before entering Increment.
58.1 Declaration Protection

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C++. An Ada package contains three parts that, for GNAT, are separated into two files: .ads files contain public and private Ada specifications, and .adb files contain the implementation, or Ada bodies.

Java doesn't provide any means to cleanly separate the specification of methods from their implementation: they all appear in the same file. You can use interfaces to emulate having separate specifications, but this requires the use of OOP techniques which is not always practical.

Ada and C++ do offer separation between specifications and implementations out of the box, independent of OOP.

```ada
package Package_Name is
   -- public specifications
private
   -- private specifications
end Package_Name;

package body Package_Name is
   -- implementation
end Package_Name;
```

Private types are useful for preventing the users of a package's types from depending on the types' implementation details. The `private` keyword splits the package spec into "public" and "private" parts. That is somewhat analogous to C++'s partitioning of the class construct into different sections with different visibility properties. In Java, the encapsulation has to be done field by field, but in Ada the entire definition of a type can be hidden. For example:

```ada
package Types is
   type Type_1 is private;
   type Type_2 is private;
   type Type_3 is private;
   procedure P (X : Type_1);
   ...
private
   procedure Q (Y : Type_1);
   type Type_1 is new Integer range 1 .. 1000;
   type Type_2 is array (Integer range 1 .. 1000) of Integer;
   type Type_3 is record
      A, B : Integer;
   end record;
end Types;
```
Learning Ada

Subprograms declared above the **private** separator (such as P) will be visible to the package user, and the ones below (such as Q) will not. The body of the package, the implementation, has access to both parts.

### 58.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

```ada
-- root-child.ads
package Root.Child is
  -- package spec goes here
end Root.Child;
-- root-child.adb
package body Root.Child is
  -- package body goes here
end Root.Child;
```

Here, `Root.Child` is a child package of `Root`. The public part of `Root.Child` has access to the public part of `Root`. The private part of `Child` has access to the private part of `Root`, which is one of the main advantages of child packages. However, there is no visibility relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

### 58.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a `with` clause that references the package, which is similar to the C++ `#include` directive. Visibility is implicit in Java: you can always access all classes located in your `CLASSPATH`. After a `with` clause, entities needs to be prefixed by the name of their package, like a C++ namespace or a Java package. This prefix can be omitted if a `use` clause is employed, similar to a C++ `using namespace` or a Java `import`.

**[Ada]**

```ada
-- pck.ads
package Pck is
  My_Glob : Integer;
end Pck;
-- main.adb
with Pck;
procedure Main is
begin
  Pck.My_Glob := 0;
end Main;
```

**[C++]**

```c++
1308 Chapter 58. Packages
```
Learning Ada

// pck.h
namespace pck {
    extern int myGlob;
}

// pck.cpp
namespace pck {
    int myGlob;
}

// main.cpp
#include "pck.h"

int main (int argc, char ** argv) {
    pck::myGlob = 0;
}

[Java]

// Globals.java
package pck;

public class Globals {
    public static int myGlob;
}

// Main.java
public class Main {
    public static void main (String [] argv) {
        pck.Globals.myGlob = 0;
    }
}
Chapter 58. Packages
59.1 Primitive Subprograms

Primitive subprograms in Ada are basically the subprograms that are eligible for inheritance / derivation. They are the equivalent of C++ member functions and Java instance methods. While in C++ and Java these subprograms are located within the nested scope of the type, in Ada they are simply declared in the same scope as the type. There’s no syntactic indication that a subprogram is a primitive of a type.

The way to determine whether P is a primitive of a type T is if

1. it is declared in the same scope as T, and
2. it contains at least one parameter of type T, or returns a result of type T.

In C++ or Java, the self reference this is implicitly declared. It may need to be explicitly stated in certain situations, but usually it's omitted. In Ada the self-reference, called the controlling parameter, must be explicitly specified in the subprogram parameter list. While it can be any parameter in the profile with any name, we’ll focus on the typical case where the first parameter is used as the self parameter. Having the controlling parameter listed first also enables the use of OOP prefix notation which is convenient.

A class in C++ or Java corresponds to a tagged type in Ada. Here’s an example of the declaration of an Ada tagged type with two parameters and some dispatching and non-dispatching primitives, with equivalent examples in C++ and Java:

[Ada]

type T is tagged record
    V, W : Integer;
end record;

type T_Access is access all T;

function F (V : T) return Integer;

procedure P1 (V : access T);

procedure P2 (V : T_Access);

[C++]

class T {
    public:
        int V, W;
        int F ();
        void P1 ();

    (continues on next page)
void P2 (T * v);

[Java]

```java
public class T {
    public int V, W;
    public int F () {};
    public void P1 () {};
    public static void P2 (T v) {};
}
```

Note that P2 is not a primitive of T — it does not have any parameters of type T. Its parameter is of type T_Access, which is a different type.

Once declared, primitives can be called like any subprogram with every necessary parameter specified, or called using prefix notation. For example:

[Ada]

```ada
declare
    V : T;
begin
    V.P1;
end;
```

[C++]

```cpp
{
    T v;
    v.P1();
}
```

[Java]

```java
{
    T v = new T ();
    v.P1 ();
}
```

**59.2 Derivation and Dynamic Dispatch**

Despite the syntactic differences, derivation in Ada is similar to derivation (inheritance) in C++ or Java. For example, here is a type hierarchy where a child class overrides a method and adds a new method:

[Ada]

```ada
type Root is tagged record
    F1 : Integer;
end record;

procedure Method_1 (Self : Root);
```
type Child is new Root with record
  F2 : Integer;
end record;

overriding procedure Method_1 (Self : Child);

procedure Method_2 (Self : Child);

[C++]

class Root {
  public:
    int f1;
    virtual void method1 ();
};

class Child : public Root {
  public:
    int f2;
    virtual void method1 ();
    virtual void method2 ();
};

[Java]

public class Root {
  public int f1;
  public void method1 ();
}

public class Child extends Root {
  public int f2;
  @Override
  public void method1 ();
  public void method2 ();
}

Like Java, Ada primitives on tagged types are always subject to dispatching; there is no need to mark them virtual. Also like Java, there's an optional keyword overriding to ensure that a method is indeed overriding something from the parent type.

Unlike many other OOP languages, Ada differentiates between a reference to a specific tagged type, and a reference to an entire tagged type hierarchy. While Root is used to mean a specific type, Root'Class — a class-wide type — refers to either that type or any of its descendants. A method using a parameter of such a type cannot be overridden, and must be passed a parameter whose type is of any of Root's descendants (including Root itself).

Next, we'll take a look at how each language finds the appropriate method to call within an OO class hierarchy; that is, their dispatching rules. In Java, calls to non-private instance methods are always dispatching. The only case where static selection of an instance method is possible is when calling from a method to the super version.

In C++, by default, calls to virtual methods are always dispatching. One common mistake is to use a by-copy parameter hoping that dispatching will reach the real object. For example:

void proc (Root p) {
  p.method1 ();
}
Root * v = new Child ();
proc (*v);

In the above code, p.method1() will not dispatch. The call to proc makes a copy of the Root part of v, so inside proc, p.method1() refers to the method1() of the root object. The intended behavior may be specified by using a reference instead of a copy:

```ada
void proc (Root & p) {
    p.method1 ();
}
```

Root * v = new Child ();
proc (*v);

In Ada, tagged types are always passed by reference but dispatching only occurs on class-wide types. The following Ada code is equivalent to the latter C++ example:

```ada
declare
    procedure Proc (P : Root'Class) is
    begin
        P.Method_1;
    end;

    type Root_Access is access all Root'Class;
    V : Root_Access := new Child;
begin
    Proc (V.all);
end;
```

Dispatching from within primitives can get tricky. Let's consider a call to Method_1 in the implementation of Method_2. The first implementation that might come to mind is:

```ada
procedure Method_2 (P : Root) is
begin
    P.Method_1;
end;
```

However, Method_2 is called with a parameter that is of the definite type Root. More precisely, it is a definite view of a child. So, this call is not dispatching; it will always call Method_1 of Root even if the object passed is a child of Root. To fix this, a view conversion is necessary:

```ada
procedure Method_2 (P : Root) is
begin
    Root'Class (P).Method_1;
end;
```

This is called "redispaching." Be careful, because this is the most common mistake made in Ada when using OOP. In addition, it's possible to convert from a class wide view to a definite view, and to select a given primitive, like in C++:

[Ada]

```ada
procedure Proc (P : Root'Class) is
begin
    Root (P).Method_1;
end;
```

[C++]

1314 Chapter 59. Classes and Object Oriented Programming
void proc (Root & p) {
    p.Root::method1 ();
}

59.3 Constructors and Destructors

Ada does not have constructors and destructors in quite the same way as C++ and Java, but there is analogous functionality in Ada in the form of default initialization and finalization.

Default initialization may be specified for a record component and will occur if a variable of the record type is not assigned a value at initialization. For example:

type T is tagged record
    F : Integer := Compute_Default_F;
end record;

function Compute_Default_F return Integer is
begin
    Put_Line ("Compute");
    return 0;
end Compute_Default_F;

V1 : T;
V2 : T := (F => 0);

In the declaration of V1, T.F receives a value computed by the subprogram Compute_Default_F. This is part of the default initialization. V2 is initialized manually and thus will not use the default initialization.

For additional expressive power, Ada provides a type called Ada.Finalization. Controlled from which you can derive your own type. Then, by overriding the Initialize procedure you can create a constructor for the type:

type T is new Ada.Finalization.Controlled with record
    F : Integer;
end record;

procedure Initialize (Self : in out T) is
begin
    Put_Line ("Compute");
    Self.F := 0;
end Initialize;

V1 : T;
V2 : T := (F => 0);

Again, this default initialization subprogram is only called for V1; V2 is initialized manually. Furthermore, unlike a C++ or Java constructor, Initialize is a normal subprogram and does not perform any additional initialization such as calling the parent's initialization routines.

When deriving from Controlled, it's also possible to override the subprogram Finalize, which is like a destructor and is called for object finalization. Like Initialize, this is a regular subprogram. Do not expect any other finalizers to be automatically invoked for you.

Controlled types also provide functionality that essentially allows overriding the meaning of the assignment operation, and are useful for defining types that manage their own storage reclamation (for example, implementing a reference count reclamation strategy).
59.4 Encapsulation

While done at the class level for C++ and Java, Ada encapsulation occurs at the package level and targets all entities of the language, as opposed to only methods and attributes. For example:

[Ada]

```ada
package Pck is
  type T is tagged private;
  procedure Method1 (V : T);
private
  type T is tagged record
    F1, F2 : Integer;
  end record;
  procedure Method2 (V : T);
end Pck;
```

[C++]

```cpp
class T {
  public:
    virtual void method1 ();
  protected:
    int f1, f2;
    virtual void method2 ();
};
```

[Java]

```java
public class T {
  public void method1 ();
  protected int f1, f2;
  protected void method2 ();
}
```

The C++ and Java code's use of `protected` and the Ada code's use of `private` here demonstrates how to map these concepts between languages. Indeed, the private part of an Ada child package would have visibility of the private part of its parents, mimicking the notion of `protected`. Only entities declared in the package body are completely isolated from access.

59.5 Abstract Types and Interfaces

Ada, C++ and Java all offer similar functionality in terms of abstract classes, or pure virtual classes. It is necessary in Ada and Java to explicitly specify whether a tagged type or class is `abstract`, whereas in C++ the presence of a pure virtual function implicitly makes the class an abstract base class. For example:

[Ada]

```ada
package P is
  type T is abstract tagged private;
  procedure Method (Self : T) is abstract;
private
  type T is abstract tagged record
```

(continues on next page)
F1, F2 : Integer;
end record;
end P;

[C++]

class T {
    public:
        virtual void method () = 0;
    protected:
        int f1, f2;
};

[Java]

public abstract class T {
    public abstract void method1 ();
    protected int f1, f2;
};

All abstract methods must be implemented when implementing a concrete type based on an abstract type.

Ada doesn't offer multiple inheritance the way C++ does, but it does support a Java-like notion of interfaces. An interface is like a C++ pure virtual class with no attributes and only abstract members. While an Ada tagged type can inherit from at most one tagged type, it may implement multiple interfaces. For example:

[Ada]

type Root is tagged record
    F1 : Integer;
end record;
procedure M1 (Self : Root);

type I1 is interface;
procedure M2 (Self : I1) is abstract;

type I2 is interface;
procedure M3 (Self : I2) is abstract;

type Child is new Root and I1 and I2 with record
    F2 : Integer;
end record;

-- M1 implicitly inherited by Child
procedure M2 (Self : Child);
procedure M3 (Self : Child);

[C++]

class Root {
    public:
        virtual void M1();
    int f1;
};

class I1 {
    public:
        virtual void M2 () = 0;
    };
class I2 {
    public:
    virtual void M3 () = 0;
};

class Child : public Root, I1, I2 {
    public:
    int f2;
    virtual void M2 ();
    virtual void M3 ();
};

[Java]

public class Root {
    public void M1();
    public int f1;
}

public interface I1 {
    public void M2 ();
}

public interface I2 {
    public void M3 ();
}

public class Child extends Root implements I1, I2 {
    public int f2;
    public void M2 ();
    public void M3 ();
}

59.6 Invariants

Any private type in Ada may be associated with a Type_Invariant contract. An invariant is a property of a type that must always be true after the return from of any of its primitive subprograms. (The invariant might not be maintained during the execution of the primitive subprograms, but will be true after the return.) Let's take the following example:

package Int_List_Pkg is

    type Int_List (Max Length : Natural) is private
    with Type_Invariant => Is_Sorted (Int_List);

    function Is_Sorted (List : Int_List) return Boolean;

    type Int_Array is array (Positive range <>) of Integer;

    function To_Int_List (Ints : Int_Array) return Int_List;

    function To_Int_Array (List : Int_List) return Int_Array;

    function "&" (Left, Right : Int_List) return Int_List;

    ... -- Other subprograms

(continues on next page)
The `Is_Sorted` function checks that the type stays consistent. It will be called at the exit of every primitive above. It is permissible if the conditions of the invariant aren't met during execution of the primitive. In `To_Int_List` for example, if the source array is not in sorted order, the invariant will not be satisfied at the "begin", but it will be checked at the end.
Ada, C++, and Java all have support for generics or templates, but on different sets of language entities. A C++ template can be applied to a class or a function. So can a Java generic. An Ada generic can be either a package or a subprogram.

### 60.1 Generic Subprograms

In this example, we will swap two generic objects. This is possible in Ada and C++ using a temporary variable. In Java, parameters are a copy of a reference value that is passed into the function, so modifying those references in the function scope has no effect from the caller's context. A generic swap method, like the below Ada or C++ examples is not possible in Java, so we will skip the Java version of this example.

**[Ada]**

```ada
generic
  type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
  Temp : A_Type := Left;
begin
  Left := Right;
  Right := Temp;
end Swap;
```

**[C++]**

```cpp
template <class AType>
AType swap (AType & left, AType & right) {
  AType temp = left;
  left = right;
  right = temp;
}
```

And examples of using these:

**[Ada]**

```ada
declare
type R is record
  F1, F2 : Integer;
end record;

  procedure Swap_R is new Swap (R);
  A, B : R;
begin
  ...
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
swap_R (A, B);
end;

[C++]

```class R {
  public:
    int f1, f2;
};
R a, b;
 swap (a, b);
```

The C++ template becomes usable once defined. The Ada generic needs to be explicitly instantiated using a local name and the generic's parameters.

### 60.2 Generic Packages

Next, we're going to create a generic unit containing data and subprograms. In Java or C++, this is done through a class, while in Ada, it's a *generic package*. The Ada and C++ model is fundamentally different from the Java model. Indeed, upon instantiation, Ada and C++ generic data are duplicated; that is, if they contain global variables (Ada) or static attributes (C++), each instance will have its own copy of the variable, properly typed and independent from the others. In Java, generics are only a mechanism to have the compiler do consistency checks, but all instances are actually sharing the same data where the generic parameters are replaced by `java.lang.Object`. Let's look at the following example:

[Ada]

```ada
generic
type T is private;
package Gen is
type C is tagged record
  V : T;
end record;
  G : Integer;
end Gen;
```

[C++]

```cpp
template <class T>
class C{
  public:
    T v;
    static int G;
};
```

[Java]

```java
public class C <T> {
  public T v;
  public static int G;
}
```

In all three cases, there's an instance variable (v) and a static variable (G). Let's now look at the behavior (and syntax) of these three instantiations:
In the Java case, we access the generic entity directly without using a parametric type. This is because there’s really only one instance of $C$, with each instance sharing the same global variable $G$. In C++, the instances are implicit, so it’s not possible to create two different instances with the same parameters. The first two assignments are manipulating the same global while the third one is manipulating a different instance. In the Ada case, the three instances are explicitly created, named, and referenced individually.

### 60.3 Generic Parameters

Ada offers a wide variety of generic parameters which is difficult to translate into other languages. The parameters used during instantiation — and as a consequence those on which the generic unit may rely on — may be variables, types, or subprograms with certain properties. For example, the following provides a sort algorithm for any kind of array:

```ada
generic
  type Component is private;
  type Index is (<>);
  with function "<" (Left, Right : Component) return Boolean;
  type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

The above declaration states that we need a type (Component), a discrete type (Index), a comparison subprogram (""), and an array definition (Array_Type). Given these, it’s possible to write an algorithm that can sort any Array_Type. Note the usage of the `with` reserved word in front of the function name, to differentiate between the generic parameter and the beginning of the generic subprogram.

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

```ada
type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type, such as String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<>); -- T is a discrete type
```

(continues on next page)
Learning Ada

(continued from previous page)

<table>
<thead>
<tr>
<th>Type Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type T is range &lt;&gt;;</code></td>
<td>T is an integer type</td>
</tr>
<tr>
<td><code>type T is digits &lt;&gt;;</code></td>
<td>T is a floating point type</td>
</tr>
<tr>
<td><code>type T is access T2;</code></td>
<td>T is an access type, T2 is its designated type</td>
</tr>
</tbody>
</table>
Exceptions are a mechanism for dealing with run-time occurrences that are rare, that usually correspond to errors (such as improperly formed input data), and whose occurrence causes an unconditional transfer of control.

### 61.1 Standard Exceptions

Compared with Java and C++, the notion of an Ada exception is very simple. An exception in Ada is an object whose “type” is `exception`, as opposed to classes in Java or any type in C++. The only piece of user data that can be associated with an Ada exception is a String. Basically, an exception in Ada can be raised, and it can be handled; information associated with an occurrence of an exception can be interrogated by a handler.

Ada makes heavy use of exceptions especially for data consistency check failures at run time. These include, but are not limited to, checking against type ranges and array boundaries, null pointers, various kind of concurrency properties, and functions not returning a value. For example, the following piece of code will raise the exception `Constraint_Error`:

```ada
procedures P is
  V : Positive;
begin
  V := -1;
end P;
```

In the above code, we're trying to assign a negative value to a variable that's declared to be positive. The range check takes place during the assignment operation, and the failure raises the `Constraint_Error` exception at that point. (Note that the compiler may give a warning that the value is out of range, but the error is manifest as a run-time exception.) Since there is no local handler, the exception is propagated to the caller; if `P` is the main procedure, then the program will be terminated.

Java and C++ can **throw** and **catch** exceptions when **try**ing code. All Ada code is already implicitly within **try** blocks, and exceptions are raised and handled.

[] (Ada)

```ada
begin
  Some_Call;
exception
  when Exception_1 =>
    Put_Line ("Error 1");
  when Exception_2 =>
    Put_Line ("Error 2");
  when others =>
    Put_Line ("Unknown error");
end;
```
Learning Ada

[C++]

```
try {
    someCall ();
} catch (Exception1) {
    cout << "Error 1" << endl;
} catch (Exception2) {
    cout << "Error 2" << endl;
} catch (...) {
    cout << "Unknown error" << endl;
}
```

[Java]

```
try {
    someCall ();
} catch (Exception1 e1) {
    System.out.println ("Error 1");
} catch (Exception2 e2) {
    System.out.println ("Error 2");
} catch (Throwable e3) {
    System.out.println ("Unknown error");
}
```

Raising and throwing exceptions is permissible in all three languages.

### 61.2 Custom Exceptions

Custom exception declarations resemble object declarations, and they can be created in Ada using the `exception` keyword:

```
My_Exception : exception;
```

Your exceptions can then be raised using a `raise` statement, optionally accompanied by a message following the `with` reserved word:

[Ada]

```
raise My_Exception with "Some message";
```

[C++]

```
throw My_Exception ("Some message");
```

[Java]

```
throw new My_Exception ("Some message");
```

Language defined exceptions can also be raised in the same manner:

```
raise Constraint_Error;
```


62.1 Tasks

Java and Ada both provide support for concurrency in the language. The C++ language has added a concurrency facility in its most recent revision, C++11, but we are assuming that most C++ programmers are not (yet) familiar with these new features. We thus provide the following mock API for C++ which is similar to the Java Thread class:

```java
class Thread {
    public:
        virtual void run (); // code to execute
        void start (); // starts a thread and then call run ()
        void join (); // waits until the thread is finished
};
```

Each of the following examples will display the 26 letters of the alphabet twice, using two concurrent threads/tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]

```ada
procedure Main is -- implicitly called by the environment task
    task My_Task;
    task body My_Task is
        begin
            for I in 'A' .. 'Z' loop
                Put_Line (I);
            end loop;
        end My_Task;
    begin
        for I in 'A' .. 'Z' loop
            Put_Line (I);
        end loop;
    end Main;
```

[C++]

```cpp
class MyThread : public Thread {
    public:
        void run () {
            for (char i = 'A'; i <= 'Z'; ++i) {
                cout << i << endl;
            }
        }
};
```

(continues on next page)
int main (int argc, char ** argv) {
    MyThread myTask;
    myTask.start ();

    for (char i = 'A'; i <= 'Z'; ++i) {
        cout << i << endl;
    }

    myTask.join ();

    return 0;
}

public class Main {
    static class MyThread extends Thread {
        public void run () {
            for (char i = 'A'; i <= 'Z'; ++i) {
                System.out.println (i);
            }
        }
    }

    public static void main (String args) {
        MyThread myTask = new MyThread ();
        myTask.start ();

        for (char i = 'A'; i <= 'Z'; ++i) {
            System.out.println (i);
        }

        myTask.join ();
    }
}

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the begin. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

An Ada task type is somewhat analogous to a Java Thread subclass, but in Java the instances of such a subclass are always dynamically allocated. In Ada an instance of a task type may either be declared or dynamically allocated.

Task types can be parametrized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the 'Z' Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the implementation's task scheduling algorithm.

[Ada]

task type My_Task (First : Character);

task body My_Task is
begin
    for I in First .. 'Z' loop

(continues on next page)
Put_Line (I);
end loop;
end My_Task;

procedure Main is
  Tab : array (0 .. 9) of My_Task ('G');
begin
  null;
end Main;

[C++]

class MyThread : public Thread {
  public:

    char first;

    void run () {
      for (char i = first; i <= 'Z'; ++i) {
        cout << i << endl;
      }
    }
};

int main (int argc, char ** argv) {
  MyThread tab [10];

  for (int i = 0; i < 9; ++i) {
    tab [i].first = 'G';
    tab [i].start ();
  }

  for (int i = 0; i < 9; ++i) {
    tab [i].join ();
  }
  return 0;
}

[Java]

public class MyThread extends Thread {
  public char first;

  public MyThread (char first){
    this.first = first;
  }

  public void run () {
    for (char i = first; i <= 'Z'; ++i) {
      cout << i << endl;
    }
  }
}

public class Main {
  public static void main (String args) {
    MyThread [] tab = new MyThread [10];

    for (int i = 0; i < 9; ++i) {
      tab [i] = new MyThread ('G');
    }
  }
}
In Ada a task may be allocated on the heap as opposed to the stack. The task will then start as soon as it has been allocated, and terminates when its work is completed. This model is probably the one that’s the most similar to Java:

[Ada]

```ada
type Ptr_Task is access My_Task;

procedure Main is
  T : Ptr_Task;
begin
  T := new My_Task ('G');
  end Main;
```

[C++]

```c++
int main (int argc, char ** argv) {
  MyThread * t = new MyThread ()
  t->first = 'G';
  t->start ();
  return 0;
}
```

[Java]

```java
public class Main {
  public static void main (String args) {
    MyThread t = new MyThread ('G');
    t.start ();
  }
}
```

### 62.2 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Ada’s rendezvous facility cannot be modeled with C++ or Java without complex machinery. Therefore, this section will just show examples written in Ada. Let’s consider the following example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  task After is
    entry Go;
  end After;
```

(continues on next page)
The Go entry declared in After is the external interface to the task. In the task body, the accept statement causes the task to wait for a call on the entry. This particular entry and accept pair doesn't do much more than cause the task to wait until Main calls After.Go. So, even though the two tasks start simultaneously and execute independently, they can coordinate via Go. Then, they both continue execution independently after the rendezvous.

The entry/accept pair can take/pass parameters, and the accept statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  task After is
    entry Go (Text : String);
    end After;

  task body After is
    begin
      accept Go (Text : String) do
        Put_Line ("After: " & Text);
        end Go;
    end After;

    begin
      Put_Line ("Before");
      After.Go ("Main");
    end;

In the above example, the Put_Line is placed in the accept statement. Here's a possible execution trace, assuming a uniprocessor:

1. At the begin of Main, task After is started and the main procedure is suspended.
2. After reaches the accept statement and is suspended, since there is no pending call on the Go entry.
3. The main procedure is awakened and executes the Put_Line invocation, displaying the string "Before".
4. The main procedure calls the Go entry. Since After is suspended on its accept statement for this entry, the call succeeds.
5. The main procedure is suspended, and the task After is awakened to execute the body of the accept statement. The actual parameter "Main" is passed to the accept statement, and the Put_Line invocation is executed. As a result, the string "After: Main" is displayed.
6. When the accept statement is completed, both the After task and the main proce-
Learning Ada

dure are ready to run. Suppose that the Main procedure is given the processor. It
reaches its end, but the local task After has not yet terminated. The main procedure
is suspended.

7. The After task continues, and terminates since it is at its end. The main procedure is
resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform
various optimizations to avoid unnecessary context switches.

### 62.3 Selective Rendezvous

The accept statement by itself can only wait for a single event (call) at a time. The select
statement allows a task to listen for multiple events simultaneously, and then to deal with
the first event to occur. This feature is illustrated by the task below, which maintains an
integer value that is modified by other tasks that call Increment, Decrement, and Get:

```ada
task Counter is
  entry Get (Result : out Integer);
  entry Increment;
  entry Decrement;
end Counter;

task body Counter is
  Value : Integer := 0;
begin
  loop
    select
      accept Increment do
        Value := Value + 1;
        end Increment;
      or
      accept Decrement do
        Value := Value - 1;
        end Decrement;
      or
      accept Get (Result : out Integer) do
        Result := Value;
        end Get;
      or
      delay 60.0; -- delay 1 minute
      exit;
    end select;
  end loop;
end Counter;
```

When the task's statement flow reaches the select, it will wait for all four events — three
entries and a delay — in parallel. If the delay of one minute is exceeded, the task will
execute the statements following the delay statement (and in this case will exit the loop,
in effect terminating the task). The accept bodies for the Increment, Decrement, or Get
entries will be otherwise executed as they're called. These four sections of the select
statement are mutually exclusive: at each iteration of the loop, only one will be invoked.
This is a critical point; if the task had been written as a package, with procedures for the
various operations, then a "race condition" could occur where multiple tasks simultaneously
calling, say, Increment, cause the value to only get incremented once. In the tasking
version, if multiple tasks simultaneously call Increment then only one at a time will be
accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one
of the entries and Counter is not ready to accept the call (i.e., if Counter is not suspended
at the `select` statement) then the calling task is suspended, and placed in the queue of
the entry that it is calling. From the perspective of the Counter task, at any iteration of the
loop there are several possibilities:

- There is no call pending on any of the entries. In this case Counter is suspended. It
  will be awakened by the first of two events: a call on one of its entries (which will then
  be immediately accepted), or the expiration of the one minute delay (whose effect
  was noted above).

- There is a call pending on exactly one of the entries. In this case control passes to the
  `select` branch with an `accept` statement for that entry. The choice of which caller to
  accept, if more than one, depends on the queuing policy, which can be specified via
  a pragma defined in the Real-Time Systems Annex of the Ada standard; the default is
  First-In First-Out.

- There are calls pending on more than one entry. In this case one of the entries with
  pending callers is chosen, and then one of the callers is chosen to be de-queued (the
  choices depend on the queueing policy).

### 62.4 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared
data object, an alternative (and generally preferable) style is through a protected object,
an efficiently implementable mechanism that makes the effect more explicit. A protected
object has a public interface (its protected operations) for accessing and manipulating the
object’s components (its private part). Mutual exclusion is enforced through a conceptual
lock on the object, and encapsulation ensures that the only external access to the compo-
nents are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by pro-
cedures or entries, and read-only operations by functions. The lock mechanism is imple-
mented so that it’s possible to perform concurrent read operations but not concurrent write
or read/write operations.

Let's reimplement our earlier tasking example with a protected object called Counter:

```ada
protected Counter is
  function Get return Integer;
  procedure Increment;
  procedure Decrement;
private
  Value : Integer := 0;
end Counter;

protected body Counter is
  function Get return Integer is
    begin
      return Value;
    end Get;

  procedure Increment is
    begin
      Value := Value + 1;
    end Increment;

  procedure Decrement is
    begin
      Value := Value - 1;
    end Decrement;
end Counter;
```
Learning Ada

Having two completely different ways to implement the same paradigm might seem complicated. However, in practice the actual problem to solve usually drives the choice between an active structure (a task) or a passive structure (a protected object).

A protected object can be accessed through prefix notation:

```
Counter.Increment;
Counter.Decrement;
Put_Line (Integer'Image (Counter.Get));
```

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```
declare
    protected type Counter is
        -- as above
end Counter;

protected body Counter is
    -- as above
end Counter;

C1 : Counter;
C2 : Counter;
begin
    C1.Increment;
    C2.Decrement;
    ...
end;
```

Protected objects and types can declare a procedure-like operation known as an "entry". An entry is somewhat similar to a procedure but includes a so-called barrier condition that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

```
protected type Binary_Semaphore is
    entry Wait;
    procedure Signal;
private
    Signaled : Boolean := False;
end Binary_Semaphore;

protected body Binary_Semaphore is
    entry Wait when Signaled is
        begin
            Signaled := False;
        end Wait;
    procedure Signal is
```

(continues on next page)
Adaconcurrency features provide much further generality than what’s been presented here. For additional information please consult one of the works cited in the References section.
63.1 Representation Clauses

We've seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let's start with the following example:

[Ada]

```ada
type R is record
  V : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record
with Pack;
```

[C++]

```cpp
struct R {
  unsigned int v:8;
  bool b1;
  bool b2;
};
```

[Java]

```java
public class R {
  public byte v;
  public boolean b1;
  public boolean b2;
}
```

The Ada and the C++ code above both represent efforts to create an object that's as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C++ and Ada code are equivalent in this particular example, there's an interesting semantic difference. In C++, the number of bits required by each field needs to be specified. Here, we're stating that v is only 8 bits, effectively representing values from 0 to 255. In Ada, it's the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The Pack
aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here's how to specify a specific data layout based on the previous example:

```ada
with Interfaces; use Interfaces;

function Get_Processor_Cycles return Unsigned_64 is
  begin
    Asm ("rdtsc",
         Outputs => (Unsigned_32'Asm_Output ("=a", Low),
                     (continues on next page)
The `Unsigned_32'Asm_Output` clauses above provide associations between machine registers and source-level variables to be updated. 

The `Volatile` parameter set to `True` tells the compiler that invoking this instruction multiple times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the eax and edx registers to implement the `High` and `Low` variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User’s Guide, and the GNAT Reference manual.

### 63.3 Interfacing with C

Much effort was spent making Ada easy to interface with other languages. The `Interfaces` package hierarchy and the pragmas `Convention`, `Import`, and `Export` allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

```c
struct my_struct {
    int A, B;
};

void call (my_struct * p) {
    printf("%d", p->A);
}
```

To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C `struct my_struct` is represented, we can use the following record along with a `pragma Convention`. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

```ada
type my_struct is record
    A : Interfaces.C.int;
    B : Interfaces.C.int;
end record;
pragma Convention (C, my_struct);
```

Describing a foreign subprogram call to Ada code is called "binding" and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a `void` function maps to an
Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a `pragma Import`:

```ada
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

The Import pragma specifies that whenever Call is invoked by Ada code, it should invoke the call function with the C calling convention.

And that's all that's necessary. Here's an example of a call to Call:

```ada
declare
  V : my_struct := (A => 1, B => 2);
begin
  Call (V);
end;
```

You can also make Ada subprograms available to C code, and examples of this can be found in the GNAT User's Guide. Interfacing with C++ and Java use implementation-dependent features that are also available with GNAT.
All the usual paradigms of imperative programming can be found in all three languages that we surveyed in this document. However, Ada is different from the rest in that it's more explicit when expressing properties and expectations. This is a good thing: being more formal affords better communication among programmers on a team and between programmers and machines. You also get more assurance of the coherence of a program at many levels. Ada can help reduce the cost of software maintenance by shifting the effort to creating a sound system the first time, rather than working harder, more often, and at greater expense, to fix bugs found later in systems already in production. Applications that have reliability needs, long term maintenance requirements, or safety/security concerns are those for which Ada has a proven track record.

It's becoming increasingly common to find systems implemented in multiple languages, and Ada has standard interfacing facilities to allow Ada code to invoke subprograms and/or reference data structures from other language environments, or vice versa. Use of Ada thus allows easy interfacing between different technologies, using each for what it's best at.

We hope this guide has provided some insight into the Ada software engineer’s world and has made Ada more accessible to programmers already familiar with programming in other languages.
The Ada Information Clearinghouse website http://www.adaic.org/learn/materials/, maintained by the Ada Resource Association, contains links to a variety of training materials (books, articles, etc.) that can help in learning Ada. The Development Center page http://www.adacore.com/knowledge on AdaCore's website also contains links to useful information including videos and tutorials on Ada.

The most comprehensive textbook is John Barnes’ Programming in Ada 2012, which is oriented towards professional software developers.
Part VII

Ada for the Embedded C Developer
This course introduces you to the Ada language by comparing it to C. It assumes that you have good knowledge of the C language. It also assumes that the choice of learning Ada is guided by considerations linked to reliability, safety or security. In that sense, it teaches you Ada paradigms that should be applied in replacement of those usually applied in C.

This course also introduces you to the SPARK subset of the Ada programming language, which removes a few features of the language with undefined behavior, so that the code is fit for sound static analysis techniques.

This course was written by Quentin Ochem, Robert Tice, Gustavo A. Hoffmann, and Patrick Rogers and reviewed by Patrick Rogers, Filip Gajowniczek, and Tucker Taft.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

**Note:** Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9d7a7153d982d05e6ab

you will find it in this directory:

`projects/Courses/Intro To Ada/Imperative_Language/Greet/cba89a34b87c9d7a7153d982d05e6ab/`

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;

[317](https://creativecommons.org/licenses/by-sa/4.0)
[318](https://learn.adacore.com/zip/learning-ada_code.zip)
5. Run the application (if a main procedure is available in the project).
66.1 So, what is this Ada thing anyway?

To answer this question let’s introduce Ada as it compares to C for an embedded application. C developers are used to a certain coding semantic and style of programming. Especially in the embedded domain, developers are used to working at a very low level near the hardware to directly manipulate memory and registers. Normal operations involve mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations as it is a low level language that was designed to replace assembly language for faster, more efficient programming. Because of this minimal abstraction, the programmer has to model the data that represents the problem they are trying to solve using the language of the physical hardware.

Let’s look at an example of this problem in action by comparing the same program in Ada and C:

[C]

Listing 1: main.c

```c
#include <stdio.h>
#include <stdlib.h>

#define DEGREES_MAX (360)

typedef unsigned int degrees;

#define MOD_DEGREES(x) (x % DEGREES_MAX)

degrees add_angles(degrees* list, int length)
{
    degrees sum = 0;
    for(int i = 0; i < length; ++i) {
        sum += list[i];
    }
    return sum;
}

int main(int argc, char** argv)
{
    degrees list[argc - 1];
    for(int i = 1; i < argc; ++i) {
        list[i - 1] = MOD_DEGREES(atoi(argv[i]));
    }
    printf("Sum: %d\n", add_angles(list, argc - 1));
}
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
29  return 0;
30 }
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Introduction.Add_Angles_C

MD5: a6d184caae372c538e34c57b0b5e144b

**Runtime output**

Sum: 0

List 2: sum_angles.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

procedure Sum_Angles is
  DEGREES_MAX : constant := 360;
  type Degrees is mod DEGREES_MAX;
  type Degrees_List is array (Natural range <>) of Degrees;
  function Add_Angles (List : Degrees_List) return Degrees is
    Sum : Degrees := 0;
    begin
      for I in List'Range loop
        Sum := Sum + List (I);
      end loop;
      return Sum;
    end Add_Angles;
    List : Degrees_List (1 .. Argument_Count);
    begin
      for I in List'Range loop
        List (I) := Degrees (Integer'Value (Argument (I)));
      end loop;
      Put_Line ("Sum:" & Add_Angles (List)'Img);
    end Sum_Angles;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Introduction.Add_Angles_Ada

MD5: b5a446e5c27aa18c917ae8c2cc6c1605

**Runtime output**

Sum: 0

Here we have a piece of code in C and in Ada that takes some numbers from the command line and stores them in an array. We then sum all of the values in the array and print the result. The tricky part here is that we are working with values that model an angle in degrees. We know that angles are modular types, meaning that angles greater than 360° can also be represented as Angle mod 360. So if we have an angle of 400°, this is equivalent to 40°. In order to model this behavior in C we had to create the MOD_DEGREES
Learning Ada

macro, which performs the modulus operation. As we read values from the command line, we convert them to integers and perform the modulus before storing them into the array. We then call add_angles which returns the sum of the values in the array. Can you spot the problem with the C code?

Try running the Ada and C examples using the input sequence 340 2 50 70. What does the C program output? What does the Ada program output? Why are they different?

The problem with the C code is that we forgot to call MOD_DEGREES in the for loop of add_angles. This means that it is possible for add_angles to return values greater than DEGREES_MAX. Let’s look at the equivalent Ada code now to see how Ada handles the situation. The first thing we do in the Ada code is to create the type Degrees which is a modular type. This means that the compiler is going to handle performing the modulus operation for us. If we use the same for loop in the Add_Angles function, we can see that we aren’t doing anything special to make sure that our resulting value is within the 360° range we need it to be in.

The takeaway from this example is that Ada tries to abstract some concepts from the developer so that the developer can focus on solving the problem at hand using a data model that models the real world rather than using data types prescribed by the hardware. The main benefit of this is that the compiler takes some responsibility from the developer for generating correct code. In this example we forgot to put in a check in the C code. The compiler inserted the check for us in the Ada code because we told the compiler what we were trying to accomplish by defining strong types.

Ideally, we want all the power that the C programming language can give us to manipulate the hardware we are working on while also allowing us the ability to more accurately model data in a safe way. So, we have a dilemma; what can give us the power of operations like the C language, but also provide us with features that can minimize the potential for developer error? Since this course is about Ada, it’s a good bet we’re about to introduce the Ada language as the answer to this question...

Unlike C, the Ada language was designed as a higher level language from its conception; giving more responsibility to the compiler to generate correct code. As mentioned above, with C, developers are constantly shifting, masking, and accessing bits directly on memory pointers. In Ada, all of these operations are possible, but in most cases, there is a better way to perform these operations using higher level constructs that are less prone to mistakes, like off-by-one or unintentional buffer overflows. If we were to compare the same application written using C and with Ada using high level constructs, we would see similar performance in terms of speed and memory efficiency. If we compare the object code generated by both compilers, it’s possible that they even look identical!

66.2 Ada — The Technical Details

Like C, Ada is a compiled language. This means that the compiler will parse the source code and emit machine code native to the target hardware. The Ada compiler we will be discussing in this course is the GNAT compiler. This compiler is based on the GCC technology like many C and C++ compilers available. When the GNAT compiler is invoked on Ada code, the GNAT front-end expands and translates the Ada code into an intermediate language which is passed to GCC where the code is optimized and translated to machine code. A C compiler based on GCC performs the same steps and uses the same intermediate GCC representation. This means that the optimizations we are used to seeing with a GCC based C compiler can also be applied to Ada code. The main difference between the two compilers is that the Ada compiler is expanding high level constructs into intermediate code. After expansion, the Ada code will be very similar to the equivalent C code.

It is possible to do a line-by-line translation of C code to Ada. This feels like a natural step for a developer used to C paradigms. However, there may be very little benefit to doing so. For the purpose of this course, we’re going to assume that the choice of Ada over C is guided by
considerations linked to reliability, safety or security. In order to improve upon the reliability, safety and security of our application, Ada paradigms should be applied in replacement of those usually applied in C. Constructs such as pointers, preprocessor macros, bitwise operations and defensive code typically get expressed in Ada in very different ways, improving the overall reliability and readability of the applications. Learning these new ways of coding, often, requires effort by the developer at first, but proves more efficient once the paradigms are understood.

In this course we will also introduce the SPARK subset of the Ada programming language. The SPARK subset removes a few features of the language, i.e., those that make proof difficult, such as pointer aliasing. By removing these features we can write code that is fit for sound static analysis techniques. This means that we can run mathematical provers on the SPARK code to prove certain safety or security properties about the code.
67.1 What we mean by Embedded Software

The Ada programming language is a general programming language, which means it can be used for many different types of applications. One type of application where it particularly shines is reliable and safety-critical embedded software; meaning, a platform with a microprocessor such as ARM, PowerPC, x86, or RISC-V. The application may be running on top of an embedded operating system, such as an embedded Linux, or directly on bare metal. And the application domain can range from small entities such as firmware or device controllers to flight management systems, communication based train control systems, or advanced driver assistance systems.

67.2 The GNAT Toolchain

The toolchain used throughout this course is called GNAT, which is a suite of tools with a compiler based on the GCC environment. It can be obtained from AdaCore, either as part of a commercial contract with GNAT Pro 319 or at no charge with the GNAT Community edition 320. The information in this course will be relevant no matter which edition you're using. Most examples will be runnable on the native Linux or Windows version for convenience. Some will only be relevant in the context of a cross toolchain, in which case we'll be using the embedded ARM bare metal toolchain.

As for any Ada compiler, GNAT takes advantage of implementation permissions and offers a project management system. Because we're talking about embedded platforms, there are a lot of topics that we'll go over which will be specific to GNAT, and sometimes to specific platforms supported by GNAT. We'll try to make the distinction between what is GNAT-specific and Ada generic as much as possible throughout this course.

For an introduction to the GNAT Toolchain for the GNAT Community edition, you may refer to the Introduction to GNAT Toolchain (page 1681) course.

319 https://www.adacore.com/gnatpro
320 https://www.adacore.com/community
67.3 The GNAT Toolchain for Embedded Targets

When we're discussing embedded programming, our target device is often different from the host, which is the device we're using to actually write and build an application. In this case, we're talking about cross compilation platforms (concisely referred to as cross platforms).

The GNAT toolchain supports cross platform compilation for various target devices. This section provides a short introduction to the topic. For more details, please refer to the GNAT User's Guide Supplement for Cross Platforms 321

GNAT supports two types of cross platforms:

- **cross targets**, where the target device has an embedded operating system.
  - ARM-Linux, which is commonly found in a Raspberry-Pi, is a prominent example.

- **bareboard targets**, where the run-times do not depend on an operating system.
  - In this case, the application has direct access to the system hardware.

For each platform, a set of run-time libraries is available. Run-time libraries implement a subset of the Ada language for different use cases, and they're different for each target platform. They may be selected via an attribute in the project's GPR project file or as a command-line switch to GPRbuild. Although the run-time libraries may vary from target to target, the user interface stays the same, providing portability for the application.

Run-time libraries consists of:

1. Files that are dependent on the target board.
   - These files are responsible for configuring and interacting with the hardware.
   - They are known as a Board Support Package — commonly referred to by their abbreviation BSP.

2. Code that is target-independent.
   - This code implements language-defined functionality.

The bareboard run-time libraries are provided as customized run-times that are configured to target a very specific micro-controller or processor. Therefore, for different micro-controllers and processors, the run-time libraries need to be ported to the specific target. These are some examples of what needs to be ported:

- startup code / scripts;
- clock frequency initializations;
- memory mapping / allocation;
- interrupts and interrupt priorities;
- register descriptions.

For more details on the topic, please refer to the following chapters of the GNAT User's Guide Supplement for Cross Platforms 322:

- Bareboard Topics 323
- Customized Run-Time Libraries 324

67.4 Hello World in Ada

The first piece of code to translate from C to Ada is the usual Hello World program:

[C]

Listing 1: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    printf("Hello World\n");
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Hello_World_C
MD5: 59685c72296a032893cda71dade24196

Runtime output

Hello World

[Ada]

Listing 2: hello_world.adb

```ada
with Ada.Text_IO;

procedure Hello_World
is
    begin
        Ada.Text_IO.Put_Line ("Hello World");
    end Hello_World;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Hello_World_Ada
MD5: f1a7c6a4fd679c4caea7ee31d14aab2e

Runtime output

Hello World

The resulting program will print Hello World on the screen. Let's now dissect the Ada version to describe what is going on:

The first line of the Ada code is giving us access to the Ada.Text_IO library which contains the Put_Line function we will use to print the text to the console. This is similar to C's `#include <stdio.h>`. We then create a procedure which executes Put_Line which prints to the console. This is similar to C's `printf` function. For now, we can assume these Ada and C features have similar functionality. In reality, they are very different. We will explore that more as we delve further into the Ada language.

You may have noticed that the Ada syntax is more verbose than C. Instead of using braces `{}` to declare scope, Ada uses keywords. `is` opens a declarative scope — which is empty here as there's no variable to declare. `begin` opens a sequence of statements. Within this sequence, we're calling the function Put_Line, prefixing explicitly with the name of the library unit where it's declared, Ada.Text_IO. The absence of the end of line `\n` can also be noted, as Put_Line always terminates by an end of line.
Ada syntax might seem peculiar at first glance. Unlike many other languages, it’s not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a more explicit language — its syntax was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner. For example:

- full words like `begin` and `end` are used in place of curly braces.
- Conditions are written using `if`, `then`, `elsif`, `else`, and `end if`.
- Ada’s assignment operator does not double as an expression, eliminating potential mistakes that could be caused by `=` being used where `==` should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens `--` mark the start of a comment that continues to the end of the line. This is exactly the same as using `//` for comments in C. Multi line comments like C's `/* */` do not exist in Ada.

Ada compilers are stricter with type and range checking than most C programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding, but this helps detect problems and vulnerabilities at compile time — early on in the development cycle. In addition, checks (such as array bounds checks) provide verification that could not be done at compile time but can be performed either at runtime, or through formal proof (with the SPARK tooling).

Ada identifiers and reserved words are case insensitive. The identifiers `VAR`, `var` and `VaR` are treated as the same identifier; likewise `begin`, `BEGIN`, `Begin`, etc. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:

<table>
<thead>
<tr>
<th>abort</th>
<th>else</th>
<th>null</th>
<th>select</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>elsif</td>
<td>of</td>
<td>separate</td>
</tr>
<tr>
<td>abstract</td>
<td>end</td>
<td>or</td>
<td>some</td>
</tr>
<tr>
<td>accept</td>
<td>entry</td>
<td>others</td>
<td>subtype</td>
</tr>
<tr>
<td>access</td>
<td>exception</td>
<td>out</td>
<td>synchronized</td>
</tr>
<tr>
<td>aliased</td>
<td>exit</td>
<td>overriding</td>
<td>tagged</td>
</tr>
<tr>
<td>all</td>
<td>for</td>
<td>package</td>
<td>task</td>
</tr>
<tr>
<td>and</td>
<td>function</td>
<td>pragma</td>
<td>terminate</td>
</tr>
<tr>
<td>array</td>
<td>generic</td>
<td>private</td>
<td>then</td>
</tr>
<tr>
<td>at</td>
<td>goto</td>
<td>procedure</td>
<td>type</td>
</tr>
<tr>
<td>begin</td>
<td>if</td>
<td>protected</td>
<td>until</td>
</tr>
<tr>
<td>body</td>
<td>in</td>
<td>raise</td>
<td>use</td>
</tr>
<tr>
<td>case</td>
<td>interface</td>
<td>range</td>
<td>when</td>
</tr>
<tr>
<td>constant</td>
<td>is</td>
<td>record</td>
<td>while</td>
</tr>
<tr>
<td>declare</td>
<td>limited</td>
<td>rem</td>
<td>with</td>
</tr>
<tr>
<td>delay</td>
<td>loop</td>
<td>renames</td>
<td>xor</td>
</tr>
<tr>
<td>delta</td>
<td>mod</td>
<td>requeue</td>
<td></td>
</tr>
<tr>
<td>digits</td>
<td>new</td>
<td>return</td>
<td></td>
</tr>
<tr>
<td>do</td>
<td>not</td>
<td>reverse</td>
<td></td>
</tr>
</tbody>
</table>
Both C and Ada were designed with the idea that the code specification and code implementation could be separated into two files. In C, the specification typically lives in the .h, or header file, and the implementation lives in the .c file. Ada is superficially similar to C. With the GNAT toolchain, compilation units are stored in files with an .ads extension for specifications and with an .adb extension for implementations.

One main difference between the C and Ada compilation structure is that Ada compilation units are structured into something called packages.

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C. A specification defines a package and the implementation implements the package. We saw this in an earlier example when we included the Ada.Text_IO package into our application. The package specification has the structure:

[Ada]

```ada
package My_Package is
  -- public declarations
private
  -- private declarations
end My_Package;
```

The package implementation, or body, has the structure:

```ada
package body My_Package is
  -- implementation
end My_Package;
```

### 67.7.1 Declaration Protection

An Ada package contains three parts that, for GNAT, are separated into two files: .ads files contain public and private Ada specifications, and .adb files contain the implementation, or Ada bodies.

[Ada]

```ada
package Package_Name is
  -- public specifications
private
  -- private specifications
end Package_Name;

package body Package_Name is
```
Private types are useful for preventing the users of a package's types from depending on the types' implementation details. Another use-case is the prevention of package users from accessing package state/data arbitrarily. The private reserved word splits the package spec into public and private parts. For example:

[Ada]

Listing 3: types.ads

```ada
package Types is
  type Type_1 is private;
  type Type_2 is private;
  type Type_3 is private;
  procedure P (X : Type_1);
-- ...
private
  procedure Q (Y : Type_1);
  type Type_1 is new Integer range 1 .. 1000;
  type Type_2 is array (Integer range 1 .. 1000) of Integer;
  type Type_3 is record
    A, B : Integer;
  end record;
end Types;
```

Subprograms declared above the private separator (such as P) will be visible to the package user, and the ones below (such as Q) will not. The body of the package, the implementation, has access to both parts. A package specification does not require a private section.

### 67.7.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

[Ada]

```ada
-- root-child.ads

package Root.Child is
  -- package spec goes here
end Root.Child;

-- root-child.adb

package body Root.Child is
  -- package body goes here
end Root.Child;
```

Here, Root.Child is a child package of Root. The public part of Root.Child has access to the public part of Root. The private part of Child has access to the private part of Root, which is one of the main advantages of child packages. However, there is no visibility
relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

67.7.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a \texttt{with} clause that references the package, which is similar to the C \texttt{#include} directive. After a \texttt{with} clause makes a package available, references to the package contents require the name of the package as a prefix, with a dot after the package name. This prefix can be omitted if a \texttt{use} clause is employed.

Ada

Listing 4: pck.ads

1\#pck.ads
2\begin{verbatim}
package Pck is
  My_Glob : Integer;
end Pck;
\end{verbatim}

Listing 5: main.adb

1\#main.adb
2\begin{verbatim}
with Pck;
procedure Main is
  begin
    Pck.My_Glob := 0;
  end Main;
\end{verbatim}

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Using_Pkg_Entities
MD5: 4215ba710eb54478538dc001bb74ce09

In contrast to C, the Ada \texttt{with} clause is a \textit{semantic inclusion} mechanism rather than a \textit{text inclusion} mechanism; for more information on this difference please refer to Packages (page 35).

67.8 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

C

Listing 6: main.c

1\#include <stdio.h>
2int main(int argc, const char * argv[])
3{
4  // variable declarations
5  int a = 0, b = 0, c = 100, d;
(continues on next page)
Learning Ada

You'll notice that, in both languages, statements are terminated with a semicolon. This means that you can have multi-line statements.

The shortcuts of incrementing and decrementing

You may have noticed that Ada does not have something similar to the a++ or a-- operators. Instead you must use the full assignment A := A + 1 or A := A - 1.
In the Ada example above, there are two distinct sections to the `procedure Main`. This first section is delimited by the `is` keyword and the `begin` keyword. This section is called the declarative block of the subprogram. The declarative block is where you will define all the local variables which will be used in the subprogram. C89 had something similar, where developers were required to declare their variables at the top of the scope block. Most C developers may have run into this before when trying to write a for loop:

[C]

Listing 8: main.c

```c
/* The C89 version */
#include <stdio.h>

int average(int* list, int length) {
    int i;
    int sum = 0;
    for(i = 0; i < length; ++i) {
        sum += list[i];
    }
    return (sum / length);
}

int main(int argc, const char * argv[]) {
    int vals[] = { 2, 2, 4, 4 }; // The modern C way
    printf("Average: %d\n", average(vals, 4));
    return 0;
}
```

Code block metadata

Project: Courses.Ada For Embedded C Dev.Perspective.Average_C89
MD5: 5c89aa28c8e0bae4d963b235c53aeedf2

Runtime output

Average: 3

[C]

Listing 9: main.c

```c
// The modern C way
#include <stdio.h>

int average(int* list, int length) {
    int sum = 0;
    for(int i = 0; i < length; ++i) {
        sum += list[i];
    }
    return (sum / length);
}
```

(continues on next page)
int main(int argc, const char * argv[])
{
    int vals[] = { 2, 2, 4, 4 };
    printf("Average: %d\n", average(vals, 4));
    return 0;
}

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Average_C_Modern
MD5: 6354863137d78adb974743915d1d4530

Runtime output
Average: 3

For the fun of it, let's also see the Ada way to do this:

[Ada]

Listing 10: main.adb

with Ada.Text_IO;

procedure Main is
    type Int_Array is array (Natural range <>) of Integer;
    function Average (List : Int_Array) return Integer
    is
        Sum : Integer := 0;
    begin
        for I in List'Range loop
            Sum := Sum + List (I);
        end loop;
        return (Sum / List'Length);
    end Average;
    Vals : constant Int_Array (1 .. 4) := (2, 2, 4, 4);
begin
    Ada.Text_IO.Put_Line ("Average: " & Integer'Image (Average (Vals)));
end Main;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Average_Ada
MD5: 52abb574d7a8b3b56715735dcd1d54

Runtime output
Average: 3

We will explore more about the syntax of loops in Ada in a future section of this course; but for now, notice that the I variable used as the loop index is not declared in the declarative section!

Declaration Flippy Floppy
Something peculiar that you may have noticed about declarations in Ada is that they are backwards from the way C does declarations. The C language expects the type followed by the variable name. Ada expects the variable name followed by a colon and then the type.

The next block in the Ada example is between the `begin` and `end` keywords. This is where your statements will live. You can create new scopes by using the `declare` keyword:

```ada
[Ada]
Listing 11: main.adb
```

```ada
with Ada.Text_IO;

procedure Main
is
    -- variable declaration
    A, B : Integer := 0;
    C : Integer := 100;
    D : Integer;

begin
    -- Ada does not have a shortcut format for increment like in C
    A := A + 1;

    -- regular addition
    D := A + B + C;

    -- printing the result
    Ada.Text_IO.Put_Line ("D =" & D'Img);

    declare
        E : constant Integer := D * 100;
    begin
        -- printing the result
        Ada.Text_IO.Put_Line ("E =" & E'Img);
    end;

end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Var_Decl_Block_Ada
MD5: 9239b993a7eadb13a27bd3618a03431f

Runtime output

D = 101
E = 10100

Notice that we declared a new variable `E` whose scope only exists in our newly defined block. The equivalent C code is:

```c
[C]
Listing 12: main.c
```

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // variable declarations
    int a = 0, b = 0, c = 100, d;
```

(continues on next page)
Learning Ada

(continued from previous page)

8 // c shorthand for increment
9 a++;
10
11 // regular addition
12 d = a + b + c;
13
14 // printing the result
15 printf("d = %d\n", d);
16
17 { const int e = d * 100;
18     printf("e = %d\n", e);
19 }
20
21 return 0;
22

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Var_Decl_Block_C
MD5: 1a83795575ddc026738d92c865ab6c

Runtime output
d = 101
e = 10100

Fun Fact about the C language assignment operator =: Did you know that an assignment in C can be used in an expression? Let's look at an example:

[C]

Listing 13: main.c

#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 0;
    if (a = 10)
        printf("True\n");
    else
        printf("False\n");
    return 0;
}

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Equal_C
MD5: 2d00ddf7e154cb888082c86b8fd36c58

Runtime output
True

Run the above code example. What does it output? Is that what you were expecting?

The author of the above code example probably meant to test if \texttt{a == 10} in the if statement but accidentally typed \texttt{=} instead of \texttt{==}. Because C treats assignment as an expression, it was able to evaluate \texttt{a = 10}.
Let's look at the equivalent Ada code:

[Ada]

Listing 14: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  A : Integer := 0;
begin
  if A := 10 then
    Put_Line ("True");
  else
    Put_Line ("False");
  end if;
end Main;
```

The above code will not compile. This is because Ada does not allow assignment as an expression.

**The "use" clause**

You'll notice in the above code example, after `with Ada.Text_IO;` there is a new statement we haven't seen before — `use Ada.Text_IO;`. You may also notice that we are not using the `Ada.Text_IO` prefix before the `Put_Line` statements. When we add the `use` clause it tells the compiler that we won't be using the prefix in the call to subprograms of that package. The `use` clause is something to use with caution. For example: if we use the `Ada.Text_IO` package and we also have a `Put_Line` subprogram in our current compilation unit with the same signature, we have a (potential) collision!

### 67.9 Conditions

The syntax of an if statement:

[C]

Listing 15: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
  // try changing the initial value to change the
  // output of the program
  int v = 0;
  if (v > 0) {
    printf("Positive\n");
  } else if (v < 0) {
```

(continues on next page)
printf("Negative\n");
}
else {
    printf("Zero\n");
}

return 0;
}

Code block metadata

Project: Courses.Ada_For_EMBEDDED_C_Dev.Perspective.Condition_C
MD5: 69203e679085e73394d3620a5954262a

Runtime output

Zero

[Ada]

Listing 16: main.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Main
is
    -- try changing the initial value to change the
    -- output of the program
    V : constant Integer := 0;
    begin
        if V > 0 then
            Put_Line ("Positive");
        elsif V < 0 then
            Put_Line ("Negative");
        else
            Put_Line ("Zero");
        end if;
    end Main;

Code block metadata

Project: Courses.Ada_For_EMBEDDED_C_Dev.Perspective.Condition_Ada
MD5: 417e557708472f9022db7d8c1ed6aa33

Runtime output

Zero

In Ada, everything that appears between the if and then keywords is the conditional expression, no parentheses are required. Comparison operators are the same except for:

<table>
<thead>
<tr>
<th>Operator</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>==</td>
<td>=</td>
</tr>
<tr>
<td>Inequality</td>
<td>!=</td>
<td>/=</td>
</tr>
<tr>
<td>Not</td>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td>And</td>
<td>&amp; &amp;</td>
<td>and</td>
</tr>
<tr>
<td>Or</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The syntax of a switch/case statement:
Learning Ada

Listing 17: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // try changing the initial value to change the
    // output of the program
    int v = 0;
    switch(v) {
    case 0:
        printf("Zero\n");
        break;
    case 1: case 2: case 3: case 4: case 5:
    case 6: case 7: case 8: case 9:
        printf("Positive\n");
        break;
    case 10: case 12: case 14: case 16: case 18:
        printf("Even number between 10 and 18\n");
        break;
    default:
        printf("Something else\n");
        break;
    }
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Switch_Case_C
MD5: 1bdb3d0c151d71280ef9039841f7ee58

Runtime output

Zero

Listing 18: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main
is
    -- try changing the initial value to change the
    -- output of the program
    V : constant Integer := 0;
    begin
    case V is
        when 0 =>
            Put_Line ("Zero");
        when 1 .. 9 =>
            Put_Line ("Positive");
        when 10 | 12 | 14 | 16 | 18 =>
            Put_Line ("Even number between 10 and 18");
        when others =>
            Put_Line ("Something else");
    end case;
end Main;
```
Learning Ada

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Switch_Case_Ada
MD5: 09e2318b56069281c95f23310dc121d1

Runtime output
Zero

Switch or Case?
A switch statement in C is the same as a case statement in Ada. This may be a little strange because C uses both keywords in the statement syntax. Let's make an analogy between C and Ada: C's `switch` is to Ada's `case` as C's `case` is to Ada's `when`.

Notice that in Ada, the case statement does not use the `break` keyword. In C, we use `break` to stop the execution of a case branch from falling through to the next branch. Here is an example:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int v = 0;
    switch(v) {
    case 0:
        printf("Zero\n");
    case 1:
        printf("One\n");
    default:
        printf("Other\n");
    }
    return 0;
}
```

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Switch_Case_Break_C
MD5: fd0389205476f161655caf32244d9054

Runtime output
Zero
One
Other

Run the above code with `v = 0`. What prints? What prints when we change the assignment to `v = 1`?

When `v = 0` the program outputs the strings Zero then One then Other. This is called fall through. If you add the `break` statements back into the `switch` you can stop this fall through behavior from happening. The reason why fall through is allowed in C is to allow the behavior from the previous example where we want a specific branch to execute for multiple inputs. Ada solves this a different way because it is possible, or even probable, that the developer might forget a `break` statement accidentally. So Ada does not allow
fall through. Instead, you can use Ada's syntax to identify when a specific branch can be executed by more than one input. If you want a range of values for a specific branch you can use the `First .. Last` notation. If you want a few non-consecutive values you can use the `Value1 | Value2 | Value3` notation.

Instead of using the word `default` to denote the catch-all case, Ada uses the `others` keyword.

### 67.10 Loops

Let's start with some syntax:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int v;

    // this is a while loop
    v = 1;
    while (v < 100) {
        v *= 2;
    }
    printf("v = %d\n", v);

    // this is a do while loop
    v = 1;
    do {
        v *= 2;
    } while (v < 200);
    printf("v = %d\n", v);

    // this is a for loop
    v = 0;
    for(int i = 0; i < 5; ++i) {
        v += (i * i);
    }
    printf("v = %d\n", v);

    // this is a forever loop with a conditional exit
    v = 0;
    while (1) {
        // do stuff here
        v += 1;
        if (v == 10)
            break;
    }
    printf("v = %d\n", v);

    // this is a loop over an array
    {
        #define ARR_SIZE (10)
        const int arr[ARR_SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;
        for (int i = 0; i < ARR_SIZE; ++i) {
```

(continues on next page)
sum += arr[i];

    printf("sum = %d\n", sum);
}

    return 0;
}

Code block metadata

Project: Courses.Ada_For_Embedded_C_DEV_Perspective.Loops_C
MD5: bcd8963884e2b2a5e364219f9b6b8fbc

Runtime output

v = 128
v = 256
v = 30
v = 10
sum = 55

[Ada]

Listing 21: main.adb

with Ada.Text_IO;

procedure Main is
    V : Integer;
begin
    -- this is a while loop
    V := 1;
    while V < 100 loop
        V := V * 2;
    end loop;
    Ada.Text_IO.Put_Line ("V = " & Integer’Image (V));

    -- Ada doesn't have an explicit do while loop
    -- instead you can use the loop and exit keywords
    V := 1;
    loop
        V := V * 2;
        exit when V >= 200;
    end loop;
    Ada.Text_IO.Put_Line ("V = " & Integer’Image (V));

    -- this is a for loop
    V := 0;
    for I in 0 .. 4 loop
        V := V + (I * I);
    end loop;
    Ada.Text_IO.Put_Line ("V = " & Integer’Image (V));

    -- this is a forever loop with a conditional exit
    V := 0;
    loop
        -- do stuff here
        V := V + 1;
        exit when V = 10;
    end loop;
    Ada.Text_IO.Put_Line ("V = " & Integer’Image (V));

(continues on next page)
-- this is a loop over an array

declare
  type Int_Array is array (Natural range 1 .. 10) of Integer;

  Arr : constant Int_Array := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  Sum : Integer := 0;
begin
  for I in Arr'Range loop
    Sum := Sum + Arr (I);
  end loop;
  Ada.Text_IO.Put_Line ("Sum = " & Integer'Image (Sum));
end;
end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loops_Ada
MD5: c09a092f8d2f682ce758d4bf059b954a

Runtime output

V = 128
V = 256
V = 30
V = 10
Sum = 55

The loop syntax in Ada is pretty straightforward. The loop and end loop keywords are used to open and close the loop scope. Instead of using the break keyword to exit the loop, Ada has the exit statement. The exit statement can be combined with a logic expression using the exit when syntax.

The major deviation in loop syntax is regarding for loops. You’ll notice, in C, that you sometimes declare, and at least initialize a loop counter variable, specify a loop predicate, or an expression that indicates when the loop should continue executing or complete, and last you specify an expression to update the loop counter.

[C]

for (initialization expression; loop predicate; update expression) {
  // some statements
}

In Ada, you don't declare or initialize a loop counter or specify an update expression. You only name the loop counter and give it a range to loop over. The loop counter is read-only! You cannot modify the loop counter inside the loop like you can in C. And the loop counter will increment consecutively along the specified range. But what if you want to loop over the range in reverse order?

[C]

Listing 22: main.c

#include <stdio.h>
#define MY_RANGE (10)

int main(int argc, const char * argv[])
{

}
for (int i = MY_RANGE; i >= 0; --i) {
    printf("%d\n", i);
}
return 0;

}
Strangely enough, Ada people call the single apostrophe symbol, ', "tick". This "tick" says the we are accessing an attribute of the variable. When we do 'Img on a variable of a numerical type, we are going to return the string version of that numerical type. So in the for loop above, I'Img, or "I tick image" will return the string representation of the numerical value stored in I. We have to do this because Put_Line is expecting a string as an input parameter.

We'll discuss attributes in more details later in this chapter (page 1389).

In the above example, we are traversing over the range in reverse order. In Ada, we use the reverse keyword to accomplish this.

In many cases, when we are writing a for loop, it has something to do with traversing an array. In C, this is a classic location for off-by-one errors. Let's see an example in action:

[C]

Listing 24: main.c

```c
#include <stdio.h>
#define LIST_LENGTH (100)

int main(int argc, const char * argv[]) {
    int list[LIST_LENGTH];
    for(int i = LIST_LENGTH; i > 0; --i) {
        list[i] = LIST_LENGTH - i;
    }
    for (int i = 0; i < LIST_LENGTH; ++i)
    {
        printf("%d ", list[i]);
        if (i % 10 == 0) {
            printf("\n");
        }
    }
    return 0;
}
```

[Ada]

67.10. Loops 1373
The above Ada and C code should initialize an array using a for loop. The initial values in the array should be contiguously decreasing from 99 to 0 as we index from the first index to the last index. In other words, the first index has a value of 99, the next has 98, the next 97 ... the last has a value of 0.

If you run both the C and Ada code above you'll notice that the outputs of the two programs are different. Can you spot why?

In the C code there are two problems:

1. There's a buffer overflow in the first iteration of the loop. We would need to modify the loop initialization to `int i = LIST_LENGTH - 1;`. The loop predicate should be modified to `i >= 0;`

2. The C code also has another off-by-one problem in the math to compute the value stored in `list[i]`. The expression should be changed to be `list[i] = LIST_LENGTH - i - 1;`

These are typical off-by-one problems that plagues C programs. You'll notice that we didn't have this problem with the Ada code because we aren't defining the loop with arbitrary
Learning Ada

numeric literals. Instead we are accessing attributes of the array we want to manipulate and are using a keyword to determine the indexing direction.

We can actually simplify the Ada for loop a little further using iterators:

[Ada]

Listing 26: main.adb

1 2 3
with Ada.Text_IO; use Ada.Text_IO;

procedure Main
is
  type Int_Array is array (Natural range 1 .. 100) of Integer;

  List : Int_Array;

begin
  for I in reverse List'Range loop
    List (I) := List'Last - I;
  end loop;

  for I of List loop
    Put (I'Img & " ");
    if I mod 10 = 0 then
      New_Line;
    end if;
  end loop;

end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loop.Reverse_Ada_Simplified
MD5: 612046826199b00ed61271d6215596fe

Runtime output

99 98 97 96 95 94 93 92 91 90
89 88 87 86 85 84 83 82 81 80
79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60
59 58 57 56 55 54 53 52 51 50
49 48 47 46 45 44 43 42 41 40
39 38 37 36 35 34 33 32 31 30
29 28 27 26 25 24 23 22 21 20
19 18 17 16 15 14 13 12 11 10
9 8 7 6 5 4 3 2 1 0

In the second for loop, we changed the syntax to for I of List. Instead of I being the index counter, it is now an iterator that references the underlying element. This example of Ada code is identical to the last bit of Ada code. We just used a different method to index over the second for loop. There is no C equivalent to this Ada feature, but it is similar to C++'s range based for loop.
67.11 Type System

67.11.1 Strong Typing

Ada is considered a "strongly typed" language. This means that the language does not define any implicit type conversions. C does define implicit type conversions, sometimes referred to as integer promotion. The rules for promotion are fairly straightforward in simple expressions but can get confusing very quickly. Let's look at a typical place of confusion with implicit type conversion:

[C]

Listing 27: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    unsigned char a = 0xFF;
    char b = 0xFF;

    printf("Does a == b?\n");
    if(a == b)
        printf("Yes.\n");
    else
        printf("No.\n");

    printf("a: 0x%08X, b: 0x%08X\n", a, b);
    return 0;
}
```

Code block metadata

Project: Courses.Ada For Embedded_C.Dev.Perspective.Strong_Typing_C
MD5: cab1ac9e2c86076d8435d53904783ba0

Runtime output

Does a == b?
No.
a: 0x000000FF, b: 0xFFFFFFFF

Run the above code. You will notice that a != b! If we look at the output of the last printf statement we will see the problem. a is an unsigned number where b is a signed number. We stored a value of 0xFF in both variables, but a treated this as the decimal number 255 while b treated this as the decimal number -1. When we compare the two variables, of course they aren't equal; but that's not very intuitive. Let's look at the equivalent Ada example:

[Ada]

Listing 28: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main
is
    type Char is range 0 .. 255;
    type Unsigned_Char is mod 256;

(continues on next page)```
A : Char := 16#FF#;
B : Unsigned_Char := 16#FF#;

begin
    Put_Line ("Does A = B?");
    if A = B then
        Put_Line ("Yes");
    else
        Put_Line ("No");
    end if;
end Main;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Strong_Typing_Ada
MD5: d6ef2668809159e9fb0d42f91e893222

Build output
main.adb:14:09: error: invalid operand types for operator "="
main.adb:14:09: error: left operand has type "Char" defined at line 5
main.adb:14:09: error: right operand has type "Unsigned_Char" defined at line 6
gprbuild: *** compilation phase failed

If you try to run this Ada example you will get a compilation error. This is because the compiler is telling you that you cannot compare variables of two different types. We would need to explicitly cast one side to make the comparison against two variables of the same type. By enforcing the explicit cast we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect.

Another example: you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

Listing 29: strong_typing.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Strong_Typing is
    Alpha : constant Integer := 1;
    Beta  : constant Integer := 10;
    Result : Float;
begin
    Result := Float (Alpha) / Float (Beta);
    Put_Line (Float'Image (Result));
end Strong_Typing;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Strong_Typing_Ada_2
MD5: bf91f01b499bcd7da1df751a9f91a767

Runtime output

67.11. Type System
Listing 30: main.c

```c
#include <stdio.h>

void weakTyping (void) {
    const int alpha = 1;
    const int beta = 10;
    float result;
    result = alpha / beta;
    printf("%f\n", result);
}

int main(int argc, const char *argv[]) {
    weakTyping();
    return 0;
}
```

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from `Integer` to `Float` explicit. In fact, it significantly changes the behavior of the computation. While the Ada code performs a floating point operation 1.0 / 10.0 and stores 0.1 in `Result`, the C version instead store 0.0 in `result`. This is because the C version perform an integer operation between two integer variables: 1 / 10 is 0. The result of the integer division is then converted to a `float` and stored. Errors of this sort can be very hard to locate in complex pieces of code, and systematic specification of how the operation should be interpreted helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to `Float`:

[Ada]

```ada
-- Perform an Integer division then convert to Float
Result := Float (Alpha / Beta);
```

The complete example would then be:

[Ada]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Strong_Typing is
    Alpha : constant Integer := 1;
    Beta : constant Integer := 10;
    Result : Float;
```

(continues on next page)
begin
    Result := Float (Alpha / Beta);
    Put_Line (Float'Image (Result));
end Strong_Typing;

Code block metadata
Project: Courses.Ada_For_EMBEDDED_C_Dev.Perspective.Strong_Typing_Ada_2
MD5: 50d6a6a3270b51880c43c07f077760b6

Runtime output
0.00000E+00

Floating Point Literals
In Ada, a floating point literal must be written with both an integral and decimal part. 10 is not a valid literal for a floating point value, while 10.0 is.

67.11.2 Language-Defined Types

The principal scalar types predefined by Ada are Integer, Float, Boolean, and Character. These correspond to int, float, int (when used for Booleans), and char, respectively. The names for these types are not reserved words; they are regular identifiers. There are other language-defined integer and floating-point types as well. All have implementation-defined ranges and precision.

67.11.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators +, -, *, and / for integer and floating-point types take operands of the same type and return a value of that type.

[Ada]

Listing 32: main.adb

procedure Main is
    type Distance is new Float;
    type Area is new Float;
    D1 : Distance := 2.0;
    D2 : Distance := 3.0;
    A : Area;
    begin
(continues on next page)
Even though the Distance and Area types above are just *Float*, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two Distance yields a Distance) and prohibit some useful cases (for example multiplying two Distances should deliver an Area). These situations can be handled through other mechanisms. A predefined operation can be identified as abstract to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

*Ada enumerations work similarly to C* `enum`:

```ada
procedure Main is
  type Day is
    (Monday,
     Tuesday,
     Wednesday,
     Thursday,
     Friday,
     Saturday,
     Sunday);
  D : Day := Monday;
begin
  null;
end Main;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enumeration_Ada
MD5: 51abd1863970e14ff06859c1aae1fe8

[C]
Listing 34: main.c

```c
enum Day {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
};

int main(int argc, const char * argv[]) {
    enum Day d = Monday;
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.PerspectiveEnumeration_C
MD5: d9f6724759375a126a6b5d8dceea3f24

But even though such enumerations may be implemented by the compiler as numeric values, at the language level Ada will not confuse the fact that Monday is a Day and is not an Integer. You can compare a Day with another Day, though. To specify implementation details like the numeric values that correspond with enumeration values in C you include them in the original enum declaration:

[C]

Listing 35: main.c

```c
#include <stdio.h>

enum Day {
    Monday = 10,
    Tuesday = 11,
    Wednesday = 12,
    Thursday = 13,
    Friday = 14,
    Saturday = 15,
    Sunday = 16
};

int main(int argc, const char * argv[]) {
    enum Day d = Monday;
    printf("d = %d\n", d);
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.PerspectiveEnumeration_Values_C
MD5: 48ae1c84dafabde7a16de5305e106a80

Runtime output
But in Ada you must use both a type definition for Day as well as a separate representation clause for it like:

[Ada]

Listing 36: main.adb

```ada
with Ada.Text_IO;

procedure Main is
  type Day is
    (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

  -- Representation clause for Day type:
  for Day use
    (Monday  => 10,
     Tuesday => 11,
     Wednesday => 12,
     Thursday => 13,
     Friday => 14,
     Saturday => 15,
     Sunday => 16);

  D : Day := Monday;
  V : Integer;

  begin
    V := Day'Enum_Rep (D);
    Ada.Text_IO.Put_Line (Integer'Image (V));
  end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enumeration_Values
MD5: 9a4fa1a899cb8c240105bf8ad6dbfde3

Runtime output

10

Note that however, unlike C, values for enumerations in Ada have to be unique.

67.11.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a range constraint introduced with the range reserved word, for example:

[Ada]
In the above example, Grade is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to G1 is correct and will not raise a run-time exception. Assigning N to G1 is illegal since Grade is a different type than Integer. Converting N to Grade makes the assignment legal, and a range check on the conversion confirms that the value is within 0..100. Assigning G1 + 10 to G2 is legal since + for Grade returns a Grade (note that the literal 10 is interpreted as a Grade value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression G1 + G2 may be outside the range of Grade, but the final result will be in range. Nevertheless, depending on the representation chosen for Grade, the addition may overflow. If the compiler represents Grade values as signed 8-bit integers (i.e., machine numbers in the range -128..127) then the sum G1 + G2 may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

```ada
with Ada.Text_IO;

procedure Main is
  type Grade is range 0 .. 100;
  G1, G2 : Grade := 99;
begin
  G1 := Grade ((Integer (G1) + Integer (G2)) / 2);
  Ada.Text_IO.Put_Line (Grade'Image (G1));
end Main;
```

[Ada]
Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the Day example, one can write:

[Ada]

```ada
procedure Main is
  type Day is
    (Monday,
     Tuesday,
     Wednesday,
     Thursday,
     Friday,
     Saturday,
     Sunday);

  type Business_Day is new Day range Monday .. Friday;
  type Weekend_Day is new Day range Saturday .. Sunday;

begin
  null;
end Main;
```

Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind subtypes in Ada. A subtype is a type with optional additional constraints. For example:

[Ada]

```ada
procedure Main is
  type Day is
    (Monday,
     Tuesday,
     Wednesday,
     Thursday,
     Friday,
     Saturday,
     Sunday);

  subtype Business_Day is Day range Monday .. Friday;
  subtype Weekend_Day is Day range Saturday .. Sunday;
```

(continues on next page)
Learning Ada

(subcontinued from previous page)

```
subtype Dice_Throw is Integer range 1 .. 6;
begin
  null;
end Main;
```

### Code block metadata

- **Project**: Courses.Ada_For_Embedded_C_Dev.Perspective.Enum_Ranges_2
- **MD5**: 5bcbde5b9f1aea57ff172fccc00e1c41

These declarations don't create new types, just new names for constrained ranges of their base types.

The purpose of numeric ranges is to express some application-specific constraint that we want the compiler to help us enforce. More importantly, we want the compiler to tell us when that constraint cannot be met — when the underlying hardware cannot support the range given. There are two things to consider:

- just a range constraint, such as `A : Integer range 0 .. 10;`, or
- a type declaration, such as `type Result is range 0 .. 1_000_000_000;`.

Both represent some sort of application-specific constraint, but in addition, the type declaration promotes portability because it won't compile on targets that do not have a sufficiently large hardware numeric type. That's a definition of portability that is preferable to having something compile anywhere but not run correctly, as in C.

### 67.11.5 Unsigned And Modular Types

Unsigned integer numbers are quite common in embedded applications. In C, you can use them by declaring `unsigned int` variables. In Ada, you have two options:

- declare custom `unsigned` range types;
  - In addition, you can declare custom range subtypes or use existing subtypes such as `Natural`.
- declare custom modular types.

The following table presents the main features of each type. We discuss these types right after.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Excludes negative value</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Wraparound</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

When declaring custom range types in Ada, you may use the full range in the same way as in C. For example, this is the declaration of a 32-bit unsigned integer type and the `X` variable in Ada:

[Ada]

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;
```

(continues on next page)
X : Unsigned_Int_32 := 42;
begin
  Put_Line ("X = " & Unsigned_Int_32'Image (X));
end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Unsigned_32_Ada
MD5: 0a179ce327c022468f66b6814a981b62

Runtime output

X = 42

In C, when unsigned int has a size of 32 bits, this corresponds to the following declaration:

[C]

Listing 42: main.c

#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
{
  unsigned int x = 42;
  printf("x = %u\n", x);
  return 0;
}

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Unsigned_32_C
MD5: 546068de216de96282490e81a0f7df26

Runtime output

x = 42

Another strategy is to declare subtypes for existing signed types and specify just the range
that excludes negative numbers. For example, let's declare a custom 32-bit signed type and its unsigned subtype:

[Ada]

Listing 43: main.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Signed_Int_32 is range -2 ** 31 .. 2 ** 31 - 1;
  subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. Signed_Int_32'Last;
  -- Equivalent to:
  -- subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. 2 ** 31 - 1;
  X : Unsigned_Int_31 := 42;
begin
  Put_Line ("X = " & Unsigned_Int_31'Image (X));
end Main;
In this case, we're just skipping the sign bit of the Signed_Int_32 type. In other words, while Signed_Int_32 has a size of 32 bits, Unsigned_Int_31 has a range of 31 bits, even if the base type has 32 bits.

Note that the declaration above is actually similar to the existing Natural subtype. Ada provides the following standard subtypes:

```ada
subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```

Since they're standard subtypes, you can declare variables of those subtypes directly in your implementation, in the same way as you can declare Integer variables.

As indicated in the table above, however, there is a difference in behavior for the variables we just declared, which occurs in case of overflow. Let's consider this C example:

[C]

Listing 44: main.c
```c
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[]) {
   unsigned int x = UINT_MAX + 1;
   /* Now: x == 0 */
   printf("x = %u\n", x);
   return 0;
}
```

The corresponding code in Ada raises an exception:

[Ada]

Listing 45: main.adb
```ada
with Ada.Text_Io; use Ada.Text_Io;

procedure Main is
   type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;
   X : Unsigned_Int_32 := Unsigned_Int_32'Last + 1;
```

(continues on next page)
-- Overflow: exception is raised!

begin  
  Put_Line ("X = " & Unsigned_Int_32'Image (X));
end Main;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overflow_Wraparound_Ada
MD5: ee4c3e905c59f5c8d87311e13d079836

Build output
main.adb:6:48: warning: value not in range of type "Unsigned_Int_32" defined at
line 4 [enabled by default]
main.adb:6:48: warning: Constraint_Error will be raised at run time [enabled by
default]

Runtime output
raised CONSTRAINT_ERROR : main.adb:6 range check failed

While the C uses modulo arithmetic for unsigned integer, Ada doesn't use it for the Un-
signed_Int_32 type. Ada does, however, support modular types via type definitions using
the mod keyword. In this example, we declare a 32-bit modular type:

[Ada]
Listing 46: main.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Unsigned_32 is mod 2**32;
  X : Unsigned_32 := Unsigned_32'Last + 1;
  -- Now: X = 0
begin  
  Put_Line ("X = " & Unsigned_32'Image (X));
end Main;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overflow_Wraparound_Ada
MD5: 4ed963ab372cafc8e7a19d9c3107276b

Runtime output
X = 0

In this case, the behavior is the same as in the C declaration above.

Modular types, unlike Ada's signed integers, also provide bit-wise operations, a typical ap-
application for unsigned integers in C. In Ada, you can use operators such as and, or, xor and
not. You can also use typical bit-shifting operations, such as Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left and Rotate_Right.
67.11.6 Attributes

Attributes start with a single apostrophe ("tick"), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The 'Image and 'Value attributes allow you to transform a scalar value into a String and vice-versa. For example:

[Ada]

Listing 47: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    A : Integer := 10;
begin
    Put_Line (Integer'Image (A));
    A := Integer'Value ("99");
    Put_Line (Integer'Image (A));
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Development.Perspective.Image_Attribute
MD5: 1fcfc79ec599a26e21aef7eacffcf96e

Runtime output

```
10
99
```

Important

Semantically, attributes are equivalent to subprograms. For example, Integer'Image is defined as follows:

```
function Integer'Image(Arg : Integer'Base) return String;
```

Certain attributes are provided only for certain kinds of types. For example, the 'Val and 'Pos attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]

Listing 48: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    C : Character := 'a';
begin
    Put (C);
    C := Character'Val (Character'Pos (C) + 1);
    Put (C);
end Main;
```
A more concise way to get the next value in Ada is to use the 'Succ attribute:

[Listing 49: main.adb]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  C : Character := 'a';
begin
  Put (C);
  C := Character'_succ (C);
  Put (C);
end Main;
```

You can get the previous value using the 'Pred attribute. Here is the equivalent in C:

[Listing 50: main.c]

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
  char c = 'a';
  printf("%c", c);
  c++;
  printf("%c", c);
  return 0;
}
```

Other interesting examples are the 'First and 'Last attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, `Integer'First returns -2^{31} and `Integer'Last returns 2^{31} - 1.
### 67.11.7 Arrays and Strings

C arrays are pointers with offsets, but the same is not the case for Ada. Arrays in Ada are not interchangeable with operations on pointers, and array types are considered first-class citizens. They have dedicated semantics such as the availability of the array’s boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds — the lower bound doesn’t necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here’s an example of declaring an array of 26 characters, initializing the values from 'a' to 'z':

[Ada]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Arr_Type is array (Integer range <>) of Character;
  Arr : Arr_Type (1 .. 26);
  C : Character := 'a';
begin
  for I in Arr'Range loop
    Arr (I) := C;
    C := Character'Succ (C);
    Put (Arr (I) & " ");
    if I mod 7 = 0 then
      New_Line;
    end if;
  end loop;
end Main;
```

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
  char Arr [26];
  char C = 'a';
  for (int I = 0; I < 26; ++I) {
    Arr [I] = C++;
  }
  return 0;
}
```

---

![Code block metadata](Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Range_Ada
MD5: 8e0597f6c040c740b35c79bc4706829b

Runtime output

```
a b c d e f g
h i j k l m n
o p q r s t u
v w x y z
```

---
printf("%c ", Arr [I]);

if ((I + 1) % 7 == 0) {
      printf("\n");
  }

return 0;
}

In C, only the size of the array is given during declaration. In Ada, array index ranges are specified using two values of a discrete type. In this example, the array type declaration specifies the use of \texttt{Integer} as the index type, but does not provide any constraints (use $\langle =$, pronounced box, to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called 'Range. In our example, \texttt{Arr'Range} can also be expressed as \texttt{Arr'First .. Arr'Last}; both expressions will resolve to \texttt{1 .. 26}. So the 'Range attribute supplies the bounds for our for loop. There is no risk of stating either of the bounds incorrectly, as one might do in C where \texttt{I \langle = 26} may be specified as the end-of-loop condition.

As in C, Ada \texttt{String} is an array of \texttt{Character}. Ada strings, importantly, are not delimited with the special character '0' like they are in C. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined \texttt{String} type is very straightforward to use:

\begin{verbatim}
[Ada]
Listing 53: main.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  My_String : String (1 .. 19) := "This is an example!";
begin
  Put_Line (My_String);
end Main;
\end{verbatim}

Unlike C, Ada does not offer escape sequences such as 'n'. Instead, explicit values from the ASCII package must be concatenated (via the concatenation operator, \&). Here for example, is how to initialize a line of text ending with a new line:
Listing 54: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  My_String : String := "This is a line" & ASCII.LF;
begin
  Put (My_String);
end Main;
```

Code block metadata

Project: Courses.Ada.For_Embedded_C Dev.Perspective.Constrained_String
MD5: 684bbbdf99d48ed6fd5c257183a6609f

Runtime output

This is a line

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of My_String's bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada, the actual array contents are duplicated. To get the above behavior, actual pointer types would have to be defined and used.

Listing 55: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Arr_Type is array (Integer range <>) of Integer;
  A1 : Arr_Type (1 .. 2);
  A2 : Arr_Type (1 .. 2);
begin
  A1 (1) := 0;
  A1 (2) := 1;
  A2 := A1;
  for I in A2'Range loop
    Put_Line (Integer'Image (A2 (I)));
  end loop;
end Main;
```

Code block metadata

Project: Courses.Ada.For_Embedded_C Dev.Perspective.Array_Copy_Ada
MD5: 4d4e9aa063c1f488e7cefa90083d06c2

Runtime output

0
1
Listing 56: main.c

```c
#include <stdio.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    int A1 [2];
    int A2 [2];
    A1 [0] = 0;
    A1 [1] = 1;
    memcpy (A2, A1, sizeof (int) * 2);
    for (int i = 0; i < 2; i++) {
        printf("%d\n", A2[i]);
    }
    return 0;
}
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Copy_C
MD5: 0dade800452673b7a82afe1c656f07e6

**Runtime output**

```
0
1
```

In all of the examples above, the source and destination arrays must have precisely the same number of elements. Ada allows you to easily specify a portion, or slice, of an array. So you can write the following:

[Ada]

Listing 57: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    type Arr_Type is array (Integer range <>) of Integer;
    A1 : Arr_Type (1 .. 10) := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    A2 : Arr_Type (1 .. 5) := (1, 2, 3, 4, 5);
    begin
        A2 (1 .. 3) := A1 (4 .. 6);
        for I in A2'Range loop
            Put_Line (Integer'Image (A2 (I)));
        end loop;
    end Main;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Slice
MD5: cb2a7de2cffe8ea19025363886f8821e4

**Runtime output**

```
```
This assigns the 4th, 5th, and 6th elements of A1 into the 1st, 2nd, and 3rd elements of A2. Note that only the length matters here: the values of the indexes don’t have to be equal; they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as opposed to their addresses:

Listing 58: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Arr_Type is array (Integer range <>) of Integer;
  A1 : Arr_Type (1 .. 2) := (10, 20);
  A2 : Arr_Type (1 .. 2) := (10, 20);
begin
  if A1 = A2 then
    Put_Line ("A1 = A2");
  else
    Put_Line ("A1 /= A2");
  end if;
end Main;
```

Listing 59: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
  int eq = 1;
  for (int i = 0; i < 2; ++i) {
    if (A1 [i] != A2 [i]) {
      eq = 0;
      break;
    }
  }
  if (eq) {
    printf("A1 == A2\n");
  }
  return 0;
}
```

[Ada]
You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the others expression to define a default value for all the unspecified array elements. Therefore, you can write:

[Listing 60: main.adb]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Arr_Type is array (Integer range <>) of Integer;
  A1 : Arr_Type (-2 .. 42) := (others => 0);

  -- use a slice to assign A1 elements 11 .. 19 to 1
  A1 (11 .. 19) := (others => 1);

  Put_Line ("---- A1 ----");
  for I in A1'Range loop
    Put_Line (Integer'Image (I) & " => " & 
               Integer'Image (A1 (I)));
  end loop;

end Main;
```

[Ada]
In this example, we're specifying that A1 has a range between -2 and 42. We use (others => 0) to initialize all array elements with zero. In the next example, the number of elements is determined by looking at the right-hand side:

[Ada]

Listing 61: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
begin
   A1 := (1, 2, 3, others => 10);
   Put_Line ("---- A1 ----");
   for I in A1'Range loop
      Put_Line (Integer'Image (I) & " => " & Integer'Image (A1 (I)));
   end loop;
end Main;
```

Code block metadata
Since A1 is initialized with an aggregate of 9 elements, A1 automatically has 9 elements. Also, we're not specifying any range in the declaration of A1. Therefore, the compiler uses the default range of the underlying array type Arr_Type, which has an unconstrained range based on the Integer type. The compiler selects the first element of that type (Integer'First) as the start index of A1. If you replaced Integer range <> in the declaration of the Arr_Type by Positive range <> , then A1's start index would be Positive'First — which corresponds to one.

### 67.11.8 Heterogeneous Data Structures

The structure corresponding to a C struct is an Ada record. Here are some simple records:

[Ada]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type R is record
    A, B : Integer;
    C : Float;
  end record;
  V : R;
begin
  V.A := 0;
  Put_Line ("V.A = " & Integer'Image (V.A));
end Main;
```

[C]

```
V.A = 0
```
Listing 63: main.c

```c
#include <stdio.h>

struct R {
    int A, B;
    float C;
};

int main(int argc, const char * argv[])
{
    struct R V;
    V.A = 0;
    printf("V.A = %d
", V.A);
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Struct_C
MD5: 653b65bbb6ea02a512e439d912e11d7f

Runtime output

V.A = 0

Ada allows specification of default values for fields just like C. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by others => <> to specify that fields not listed will take their default values. For example:

[Ada]

Listing 64: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    type R is record
        A, B : Integer := 0;
        C : Float := 0.0;
    end record;

    procedure Put_R (V : R; Name : String) is
    begin
        Put_Line (Name & " = (" &
                   Integer'Image (V.A) & ", " &
                   Integer'Image (V.B) & ", " &
                   Float'Image (V.C) & ")");
    end Put_R;

    V1 : constant R := (1, 2, 1.0);
    V2 : constant R := (A => 1, B => 2, C => 1.0);
    V3 : constant R := (C => 1.0, A => 1, B => 2);
    V4 : constant R := (C => 1.0, others => <>);

    begin
        Put_R (V1, "V1");
        Put_R (V2, "V2");
        Put_R (V3, "V3");
        Put_R (V4, "V4");
```

(continues on next page)
As a foreword to the topic of pointers, it's important to keep in mind the fact that most situations that would require a pointer in C do not in Ada. In the vast majority of cases, indirect memory management can be hidden from the developer and thus saves from many potential errors. However, there are situations that do require the use of pointers, or said differently that require to make memory indirection explicit. This section will present Ada access types, the equivalent of C pointers. A further section will provide more details as to how situations that require pointers in C can be done without access types in Ada.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

Listing 65: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type R is record
    A, B : Integer;
  end record;

  procedure Put_R (V : R; Name : String) is
    begin
      Put_Line (Name & " = (" & Integer'Image (V.A) & ", " & Integer'Image (V.B) & ")");
      end Put_R;

  begin
    V1, V2 : R;
    V1.A := 0;
    V2 := V1;
    V2.A := 1;
    Put_R (V1, "V1");
    Put_R (V2, "V2");
  end Main;
```

Code block metadata
There are many commonalities between the Ada and C semantics above. In Ada and C, objects are allocated on the stack and are directly accessed. \texttt{V1} and \texttt{V2} are two different objects and the assignment statement copies the value of \texttt{V1} into \texttt{V2}. \texttt{V1} and \texttt{V2} are two distinct objects.

Here’s now a similar example, but using heap allocation instead:

[C]

Listing 66: main.c

```
#include <stdio.h>

struct R {
    int A, B;
};

void print_r(const struct R *v, const char *name) {
    printf("%s = (%d, %d)\n", name, v->A, v->B);
}

int main(int argc, const char * argv[]) {
    struct R V1, V2;
    V1.A = 0;
    V2 = V1;
    V2.A = 1;
    print_r(&V1, "V1");
    print_r(&V2, "V2");
    return 0;
}
```
A, B : Integer;
end record;

type R_Access is access R;

procedure Put_R (V : R; Name : String) is
begin
  Put_Line (Name & " = (" & Integer'Image (V.A) & ", " & Integer'Image (V.B) & ")");
end Put_R;

V1 : R_Access;
V2 : R_Access;
begin
  V1 := new R;
  V1.A := 0;
  V2 := V1;
  V2.A := 1;
  Put_R (V1.all, "V1");
  Put_R (V2.all, "V2");
end Main;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Heap_Alloc_Ada
MD5: 963b48bb0a8585a9941d8fb2d0eda390

Runtime output
V1 = ( 1, 0)
V2 = ( 1, 0)

Listing 68: main.c
#include <stdio.h>
#include <stdlib.h>

struct R {
  int A, B;
};

void print_r(const struct R *v,
  const char *name)
{
  printf("%s = (%d, %d)\n", name, v->A, v->B);
}

int main(int argc, const char * argv[])
{
  struct R * V1, * V2;
  V1 = malloc(sizeof(struct R));
  V1->A = 0;
  V2 = V1;
  V2->A = 1;
  print_r(V1, "V1");
  print_r(V2, "V2");
In this example, an object of type R is allocated on the heap. The same object is then referred to through V1 and V2. As in C, there’s no garbage collector in Ada, so objects allocated by the new operator need to be expressly freed (which is not the case here).

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append .all. The equivalent of V1->A in C can be written either as V1.A or V1.all.A.

Pointers to scalar objects in Ada and C look like:

[Ada]

```ada
procedure Main is
  type A_Int is access Integer;
  Var : A_Int := new Integer;
begin
  Var.all := 0;
end Main;
```

[C]

```c
#include <stdlib.h>

int main(int argc, const char * argv[]) {
  int * Var = malloc(sizeof(int));
  *Var = 0;
  return 0;
}
```

In Ada, an initializer can be specified with the allocation by appending '(value):
When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being aliased. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either `access all` (if the referenced object needs to be assigned to) or `access constant` (if the referenced object is a constant). The `Access` attribute works like the C `&` operator to get a pointer to the object, but with a `scope accessibility` check to prevent references to objects that have gone out of scope. For example:

**[Ada]**

```
procedure Main is
  type A_Int is access Integer;
  Var : A_Int := new Integer'(0);
begin
  null;
end Main;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Access_Init
MD5: 5789253068f77100eec34919b8de66ec

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs, it's called Ada.Unchecked_Deallocation. To create your customized deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

```
procedure Main is
  type A_Int is access all Integer;
  Var : aliased Integer;
  Ptr : A_Int := Var'Access;
begin
  null;
end Main;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Access_All
MD5: 520df34083e3517876e10710530380be

```
tdef main(int argc, const char * argv[]) {
  int Var;
  int * Ptr = &Var;
  return 0;
}
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Access_All_C
MD5: a592fcf09dabe15f2aaf12fba047d74f
67.12 Functions and Procedures

67.12.1 General Form

Subroutines in C are always expressed as functions which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term subprogram to refer to both functions and procedures.

Parameters can be passed in three distinct modes:

- **in**, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram.
- **out** is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller.
- **in out** is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant pointer in C).
Ada also provides **access** and **aliased** parameters, which are in effect explicit pass-by-reference indicators.

In Ada, the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). C has the programmer specify how to pass the parameter.

**Important**

There are some exceptions to the "general" rule in Ada. For example, parameters of scalar types are always passed by copy, for all three modes.

Here's a first example:

[Ada]

```ada
procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer);
```

**Listing 76: proc.ads**

```ada
function Func (Var : Integer) return Integer;
```

**Listing 77: func.ads**

```
with Func;
```

**Listing 78: proc.adb**

```ada
procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer)
is
begin
  Var2 := Func (Var1);
  Var3 := Var3 + 1;
end Proc;
```

**Listing 79: func.adb**

```ada
function Func (Var : Integer) return Integer
is
begin
  return Var + 1;
end Func;
```

**Listing 80: main.adb**

```
with Ada.Text_IO; use Ada.Text_IO;
with Proc;
```

```ada
procedure Main is
  V1, V2 : Integer;
begin
  V2 := 2;
  Proc (5, V1, V2);
```

(continues on next page)
Learning Ada

(continued from previous page)

```
10 Put_Line ("V1: " & Integer'Image (V1));
11 Put_Line ("V2: " & Integer'Image (V2));
end Main;
```

### Code block metadata

**Project:** Courses.Ada_For_Embedded_C_Dev.Perspective.Subroutines_Ada  
**MD5:** a35fb6ae1b37325c3f39b331e4246a8

### Runtime output

```
V1: 6
V2: 3
```

[C]

#### Listing 81: proc.h

```c
void Proc (int Var1,
          int * Var2,
          int * Var3);
```

#### Listing 82: func.h

```c
int Func (int Var);
```

#### Listing 83: proc.c

```c
#include "func.h"
void Proc (int Var1,
          int * Var2,
          int * Var3)
{
  *Var2 = Func (Var1);
  *Var3 += 1;
}
```

#### Listing 84: func.c

```c
int Func (int Var)
{
  return Var + 1;
}
```

#### Listing 85: main.c

```c
#include <stdio.h>
#include "proc.h"
int main(int argc, const char * argv[])
{
  int v1, v2;
  v2 = 2;
  Proc (5, &v1, &v2);
  printf("v1: %d\n", v1);
}
```

(continues on next page)
The first two declarations for Proc and Func are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C, a function that has not yet been seen cannot be used. Here, Proc can call Func because its specification has been declared.

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax (except for the modes), including default values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.

**In Ada 202X**

Ada 202X allows for using static expression functions, which are evaluated at compile time. To achieve this, we can use an aspect — we'll discuss aspects later in this chapter (page 1411).

An expression function is static when the Static aspect is specified. For example:

```ada
procedure Main is
  X1 : constant := (if True then 37 else 42);

  function If_Then_Else (Flag : Boolean; X, Y : Integer) return Integer is
    (if Flag then X else Y) with Static;

  X2 : constant := If_Then_Else (True, 37, 42);
begin
  null;
end Main;
```

In this example, we declare X1 using an expression. In the declaration of X2, we call the static expression function If_Then_Else. Both X1 and X2 have the same constant value.
67.12.2 Overloading

In C, function names must be unique. Ada allows overloading, in which multiple subprograms can share the same name as long as the subprogram signatures (the parameter types, and function return types) are different. The compiler will be able to resolve the calls to the proper routines or it will reject the calls. For example:

[Ada]

Listing 86: machine.ads

```ada
package Machine is
  type Status is (Off, On);
  type Code is new Integer range 0 .. 3;
  type Threshold is new Float range 0.0 .. 10.0;

  function Get (S : Status) return Code;
  function Get (S : Status) return Threshold;
end Machine;
```

Listing 87: machine.adb

```ada
package body Machine is
  function Get (S : Status) return Code is
    begin
      case S is
        when Off => return 1;
        when On  => return 3;
      end case;
    end Get;

  function Get (S : Status) return Threshold is
    begin
      case S is
        when Off => return 2.0;
        when On  => return 10.0;
      end case;
    end Get;
end Machine;
```

Listing 88: main.adb

```ada
with Ada.Text_I0; use Ada.Text_I0;
with Machine;    use Machine;

procedure Main is
  S : Status;
  C : Code;
  T : Threshold;
begin
  S := On;
  C := Get (S);
  T := Get (S);
  Put_Line ("S: " & Status'Image (S));
  Put_Line ("C: " & Code'Image (C));
  Put_Line ("T: " & Threshold'Image (T));
end Main;
```
The Ada compiler knows that an assignment to C requires a Code value. So, it chooses the Get function that returns a Code to satisfy this requirement.

Operators in Ada are functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To declare an operator as a function, enclose its “name” in quotes:

[Ada]

Listing 89: machine_2.ads

```ada
package Machine_2 is
  type Status is (Off, Waiting, On);
  type Input is new Float range 0.0 .. 10.0;
  function Get (I : Input) return Status;
  function "=" (Left : Input; Right : Status) return Boolean;
end Machine_2;
```

Listing 90: machine_2.adb

```ada
package body Machine_2 is
  function Get (I : Input) return Status is
  begin
    if I >= 0.0 and I < 3.0 then
      return Off;
    elsif I >= 3.0 and I < 6.5 then
      return Waiting;
    else
      return On;
    end if;
  end Get;

  function "=" (Left : Input; Right : Status) return Boolean is
  begin
    return Get (Left) = Right;
  end "=";
end Machine_2;
```

Listing 91: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Machine_2; use Machine_2;

procedure Main is
  I : Input;
begin
  -- (continues on next page)
```

(continues on next page)
Learning Ada

I := 3.0;
if I = Off then
  Put_Line ("Machine is off.");
else
  Put_Line ("Machine is not off.");
end if;
end Main;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overloading_Eq
MD5: c5580f15c1b93f73fff3afc147cd15a1

Runtime output
Machine is not off.

67.12.3 Aspects

Aspect specifications allow you to define certain characteristics of a declaration using the with keyword after the declaration:

procedure Some_Procedure is <procedure_definition>
  with Some_Aspect => <aspect_specification>;

function Some_Function is <function_definition>
  with Some_Aspect => <aspect_specification>;

type Some_Type is <type_definition>
  with Some_Aspect => <aspect_specification>;

Obj : Some_Type with Some_Aspect => <aspect_specification>;

For example, you can inline a subprogram by specifying the Inline aspect:

[Ada]

Listing 92: float_arrays.ads

package Float_Arrays is
  type Float_Array is array (Positive range <>) of Float;
  function Average (Data : Float_Array) return Float
    with Inline;
end Float_Arrays;

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Inline_Aspect
MD5: 6e25e81e4015d907d50aa9cf4a0a3fab

We'll discuss inlining later in this course (page 1556).

Aspect specifications were introduced in Ada 2012. In previous versions of Ada, you had to use a pragma instead. The previous example would be written as follows:

[Ada]
package Float_Arrays is
  type Float_Array is array (Positive range <>) of Float;
  function Average (Data : Float_Array) return Float;
  pragma Inline (Average);
end Float_Arrays;

Aspects and attributes might refer to the same kind of information. For example, we can use the Size aspect to define the expected minimum size of objects of a certain type:

package My_Device_Types is
  type UInt10 is mod 2 ** 10
    with Size => 10;
end My_Device_Types;

In the same way, we can use the size attribute to retrieve the size of a type or of an object:

with Ada.Text_IO; use Ada.Text_IO;
with My_Device_Types; use My_Device_Types;
procedure Show_Device_Types is
  UInt10_Obj : constant UInt10 := 0;
begin
  Put_Line ("Size of UInt10 type: " & Positive'Image (UInt10'Size));
  Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_Obj'Size));
end Show_Device_Types;

Runtime output
Size of UInt10 type: 10
Size of UInt10 object: 16
We'll explain both Size aspect and Size attribute later in this course (page 1443).
68.1 Understanding the various options

Concurrent and real-time programming are standard parts of the Ada language. As such, they have the same semantics, whether executing on a native target with an OS such as Linux, on a real-time operating system (RTOS) such as VxWorks, or on a bare metal target with no OS or RTOS at all.

For resource-constrained systems, two subsets of the Ada concurrency facilities are defined, known as the Ravenscar and Jorvik profiles. Though restricted, these subsets have highly desirable properties, including: efficiency, predictability, analyzability, absence of deadlock, bounded blocking, absence of priority inversion, a real-time scheduler, and a small memory footprint. On bare metal systems, this means in effect that Ada comes with its own real-time kernel.

For further information

We'll discuss the Ravenscar profile later in this chapter (page 1426). Details about the Jorvik profile can be found elsewhere [Jorvik].

Enhanced portability and expressive power are the primary advantages of using the standard concurrency facilities, potentially resulting in considerable cost savings. For example, with little effort, it is possible to migrate from Windows to Linux to a bare machine without requiring any changes to the code. Thread management and synchronization is all done by the implementation, transparently. However, in some situations, it's critical to be able to access directly the services provided by the platform. In this case, it's almost always possible to make direct system calls from Ada code. Several targets of the GNAT compiler provide this sort of API by default, for example win32ada for Windows and Florist for POSIX systems.

On native and RTOS-based platforms GNAT typically provides the full concurrency facilities. In contrast, on bare metal platforms GNAT typically provides the two standard subsets: Ravenscar and Jorvik.

68.2 Tasks

Ada offers a high level construct called a task which is an independent thread of execution. In GNAT, tasks are either mapped to the underlying OS threads, or use a dedicated kernel when not available.

The following example will display the 26 letters of the alphabet twice, using two concurrent tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]
Listing 1: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is -- implicitly called by the environment task
  subtype A_to_Z is Character range 'A' .. 'Z';

  task My_Task;

  task body My_Task is
  begin
    for I in A_to_Z'Range loop
      New_Line;
    end loop;
  end My_Task;

begin
  for I in A_to_Z'Range loop
    New_Line;
  end loop;
end Main;
```

Code block metadata

Project: Courses.Ada For Embedded C Dev.Concurrency.My_Task
MD5: 154702197f0c02f5750838e51a99f548

Runtime output

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the begin. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

Task types can be parameterized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the 'Z' Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the underlying implementation of the task scheduling algorithm.

[Ada]

Listing 2: my_tasks.ads

```ada
package My_Tasks is
  task type My_Task (First : Character);
end My_Tasks;
```
In Ada, a task may be dynamically allocated rather than declared statically. The task will then start as soon as it has been allocated, and terminates when its work is completed.

[Ada]
68.3 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Let's consider the following example:

[Ada]

Listing 6: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

  task After is
    entry Go;
    end After;
  task body After is
    begin
      accept Go;
      Put_Line ("After");
    end After;

    begin
      Put_Line ("Before");
      After.Go;
    end Main;
```

The Go entry declared in After is the client interface to the task. In the task body, the accept statement causes the task to wait for a call on the entry. This particular entry and accept pair simply causes the task to wait until Main calls After.Go. So, even though the two tasks start simultaneously and execute independently, they can coordinate via Go. Then, they both continue execution independently after the rendezvous.

The entry/accept pair can take/pass parameters, and the accept statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

[Ada]

Listing 7: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
```
In the above example, the Put_Line is placed in the accept statement. Here’s a possible execution trace, assuming a uniprocessor:

1. At the begin of Main, task After is started and the main procedure is suspended.
2. After reaches the accept statement and is suspended, since there is no pending call on the Go entry.
3. The main procedure is awakened and executes the Put_Line invocation, displaying the string "Before".
4. The main procedure calls the Go entry. Since After is suspended on its accept statement for this entry, the call succeeds.
5. The main procedure is suspended, and the task After is awakened to execute the body of the accept statement. The actual parameter "Main" is passed to the accept statement, and the Put_Line invocation is executed. As a result, the string "After: Main" is displayed.
6. When the accept statement is completed, both the After task and the main procedure are ready to run. Suppose that the Main procedure is given the processor. It reaches its end, but the local task After has not yet terminated. The main procedure is suspended.
7. The After task continues, and terminates since it is at its end. The main procedure is resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform various optimizations to avoid unnecessary context switches.
68.4 Selective Rendezvous

The accept statement by itself can only wait for a single event (call) at a time. The select statement allows a task to listen for multiple events simultaneously, and then to deal with the first event to occur. This feature is illustrated by the task below, which maintains an integer value that is modified by other tasks that call Increment, Decrement, and Get:

[Ada]

Listing 8: counters.ads

```ada
package Counters is
  task Counter is
    entry Get (Result : out Integer);
    entry Increment;
    entry Decrement;
  end Counter;
end Counters;
```

Listing 9: counters.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Counters is
  task body Counter is
    Value : Integer := 0;
    begin
      loop
        select
          accept Increment do
            Value := Value + 1;
            end Increment;
          or
            accept Decrement do
              Value := Value - 1;
              end Decrement;
          or
            accept Get (Result : out Integer) do
              Result := Value;
              end Get;
          or
            delay 5.0;
            Put_Line ("Exiting Counter task...");
            exit;
          end select;
      end loop;
    end Counter;
end Counters;
```

Listing 10: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Counters; use Counters;
procedure Main is
  V : Integer;
  begin
```

(continues on next page)
Learning Ada

(continued from previous page)

```
Put_Line ("Main started.");
Counter.Get (V);
Put_Line ("Got value. Value = " & Integer'Image (V));
Counter.Increment;
Put_Line ("Incremented value.");
Counter.Increment;
Put_Line ("Incremented value.");
Counter.Get (V);
Put_Line ("Got value. Value = " & Integer'Image (V));
Counter.Decrement;
Put_Line ("Decremented value.");
Counter.Get (V);
Put_Line ("Got value. Value = " & Integer'Image (V));
Put_Line ("Main finished.");
end Main;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C Dev.Concurrency.Selective_Rendezvous
MD5: 619d009bcfcd8053bc132b2e32a29249

**Runtime output**

Main started.
Got value. Value = 0
Incremented value.
Got value. Value = 2
Decrementated value.
Got value. Value = 1
Main finished.
Exiting Counter task...

When the task's statement flow reaches the select, it will wait for all four events — three entries and a delay — in parallel. If the delay of five seconds is exceeded, the task will execute the statements following the delay statement (and in this case will exit the loop, in effect terminating the task). The accept bodies for the Increment, Decrement, or Get entries will be otherwise executed as they're called. These four sections of the select statement are mutually exclusive: at each iteration of the loop, only one will be invoked. This is a critical point; if the task had been written as a package, with procedures for the various operations, then a race condition could occur where multiple tasks simultaneously calling, say, Increment, cause the value to only get incremented once. In the tasking version, if multiple tasks simultaneously call Increment then only one at a time will be accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one of the entries and Counter is not ready to accept the call (i.e., if Counter is not suspended at the select statement) then the calling task is suspended, and placed in the queue of the entry that it is calling. From the perspective of the Counter task, at any iteration of the loop there are several possibilities:

- There is no call pending on any of the entries. In this case Counter is suspended. It will be awakened by the first of two events: a call on one of its entries (which will then be immediately accepted), or the expiration of the five second delay (whose effect
Learning Ada

was noted above).

• There is a call pending on exactly one of the entries. In this case control passes to the
  select branch with an accept statement for that entry.

• There are calls pending on more than one entry. In this case one of the entries with
  pending callers is chosen, and then one of the callers is chosen to be de-queued. The
  choice of which caller to accept depends on the queuing policy, which can be specified
  via a pragma defined in the Real-Time Systems Annex of the Ada standard; the default
  is First-In First-Out.

68.5 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared
data object, an alternative (and generally preferable) style is through a protected object,
an efficiently implementable mechanism that makes the effect more explicit. A protected
object has a public interface (its protected operations) for accessing and manipulating the
object's components (its private part). Mutual exclusion is enforced through a conceptual
lock on the object, and encapsulation ensures that the only external access to the compo-
nents are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by pro-
cedures or entries, and read-only operations by functions. The lock mechanism is imple-
mented so that it's possible to perform concurrent read operations but not concurrent write
or read/write operations.

Let's reimplement our earlier tasking example with a protected object called Counter:

[Ada]

Listing 11: counters.ads

```
package Counters is

protected Counter is
  function Get return Integer;
  procedure Increment;
  procedure Decrement;
private
  Value : Integer := 0;
end Counter;

end Counters;
```

Listing 12: counters.adb

```
package body Counters is

protected body Counter is
  function Get return Integer is
    begin
      return Value;
    end Get;

  procedure Increment is
    begin
      Value := Value + 1;
    end Increment;

  procedure Decrement is
```
A protected object can be accessed through prefix notation:

```
with Ada.Text_IO; use Ada.Text_IO;
with Counters; use Counters;

procedure Main is
begin
  Counter.Increment;
  Counter.Decrement;
  Put_Line (Integer'Image (Counter.Get));
end Main;
```

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```
declare
  protected type Counter is
    -- as above
  end Counter;

  protected body Counter is
    -- as above
  end Counter;

  C1 : Counter;
  C2 : Counter;
```

(continues on next page)
Protected objects and types can declare a procedure-like operation known as an *entry*. An entry is somewhat similar to a procedure but includes a so-called barrier condition that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

[Ada]

```ada
package Binary_Semaphores is

  protected type Binary_Semaphore is
    entry Wait;
    procedure Signal;
  private
    Signaled : Boolean := False;
  end Binary_Semaphore;

end Binary_Semaphores;
```

```ada
package body Binary_Semaphores is

  protected body Binary_Semaphore is
    entry Wait when Signaled is
    begin
      Signaled := False;
      end Wait;

    procedure Signal is
    begin
      Signaled := True;
      end Signal;
  end Binary_Semaphore;

end Binary_Semaphores;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Binary_Semaphores; use Binary_Semaphores;

procedure Main is
  B : Binary_Semaphore;
  task T1;
```

(continues on next page)
task T2;

task body T1 is
begin
   Put_Line ("Task T1 waiting...");
   B.Wait;
   Put_Line ("Task T1."看望);
   delay 1.0;
   Put_Line ("Task T1 will signal...");
   B.Signal;
   Put_Line ("Task T1 finished.");
end T1;

task body T2 is
begin
   Put_Line ("Task T2 waiting...");
   B.Wait;
   Put_Line ("Task T2");
   delay 1.0;
   Put_Line ("Task T2 will signal...");
   B.Signal;
   Put_Line ("Task T2 finished.");
end T2;

begin
   Put_Line ("Main started.");
   B.Signal;
   Put_Line ("Main finished.");
end Main;

Ada concurrency features provide much further generality than what's been presented here. For additional information please consult one of the works cited in the References section.
68.6 Ravenscar

The Ravenscar profile is a subset of the Ada concurrency facilities that supports determinism, schedulability analysis, constrained memory utilization, and certification to the highest integrity levels. Four distinct application domains are intended:

- hard real-time applications requiring predictability,
- safety-critical systems requiring formal, stringent certification,
- high-integrity applications requiring formal static analysis and verification,
- embedded applications requiring both a small memory footprint and low execution overhead.

Tasking constructs that preclude analysis, either technically or economically, are disallowed. You can use the \texttt{pragma Profile} (Ravenscar) to indicate that the Ravenscar restrictions must be observed in your program.

Some of the examples we've seen above will be rejected by the compiler when using the Ravenscar profile. For example:

[Ada]

Listing 17: my_tasks.ads

```ada
package My_Tasks is
  task type My_Task (First : Character);
end My_Tasks;
```

Listing 18: my_tasks.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body My_Tasks is
  task body My_Task is
    begin
      for C in First .. 'Z' loop
        Put (C);
      end loop;
      New_Line;
    end My_Task;
end My_Tasks;
```

Listing 19: main.adb

```ada
pragma Profile (Ravenscar);
with My_Tasks; use My_Tasks;
procedure Main is
  Tab : array (0 .. 3) of My_Task ('W');
begin
  null;
end Main;
```

Code block metadata
This code violates the *No_Task_Hierarchy* restriction of the Ravenscar profile. This is due to the declaration of `Tab` in the `Main` procedure. Ravenscar requires task declarations to be done at the library level. Therefore, a simple solution is to create a separate package and reference it in the main application:

```ada
with My_Tasks; use My_Tasks;

package My_Task_Inst is
   Tab : array (0 .. 3) of My_Task ('W');
end My_Task_Inst;
```

You can use, however, one entry per protected object. As an example, the declaration of the `Binary_Semaphore` type that we’ve discussed before compiles fine with Ravenscar:

```ada
protected type Binary_Semaphore is
   entry Wait;
```

(continues on next page)
procedure Signal;
private
  Signaled : Boolean := False;
end Binary_Semaphore;

We could add more procedures and functions to the declaration of Binary_Semaphore, but we wouldn't be able to add another entry when using Ravenscar.

Similar to the previous example with the task array declaration, objects of Binary_Semaphore cannot be declared in the main application:

procedure Main is
  B : Binary_Semaphore;
begin
  null;
end Main;

This violates the *No_Local_Protected_Objects* restriction. Again, Ravenscar expects this declaration to be done on a library level, so a solution to make this code compile is to have this declaration in a separate package and reference it in the Main procedure.

Ravenscar offers many additional restrictions. Covering those would exceed the scope of this chapter. You can find more examples using the Ravenscar profile on [this blog post](https://blog.adacore.com/theres-a-mini-rtos-in-my-language).
69.1 Understanding the Ada Run-Time

Ada supports a high level of abstractness and expressiveness. In some cases, the compiler translates those constructs directly into machine code. However, there are many high-level constructs for which a direct compilation would be difficult. In those cases, the compiler links to a library containing an implementation of those high-level constructs: this is the so-called run-time library.

One typical example of high-level constructs that can be cumbersome for direct machine code generation is Ada source-code using tasking. In this case, linking to a low-level implementation of multithreading support — for example, an implementation using POSIX threads — is more straightforward than trying to make the compiler generate all the machine code.

In the case of GNAT, the run-time library is implemented using both C and Ada source-code. Also, depending on the operating system, the library will interface with low-level functionality from the target operating system.

There are basically two types of run-time libraries:

- the standard run-time library: in many cases, this is the run-time library available on desktop operating systems or on some embedded platforms (such as ARM-Linux on a Raspberry-Pi).
- the configurable run-time library: this is a capability that is used to create custom run-time libraries for specific target devices.

Configurable run-time libraries are usually used for constrained target devices where support for the full library would be difficult or even impossible. In this case, configurable run-time libraries may support just a subset of the full Ada language. There are many reasons that speak for this approach:

- Some aspects of the Ada language may not translate well to limited operating systems.
- Memory constraints may require reducing the size of the run-time library, so that developers may need to replace or even remove parts of the library.
- When certification is required, those parts of the library that would require too much certification effort can be removed.

When using a configurable run-time library, the compiler checks whether the library supports certain features of the language. If a feature isn't supported, the compiler will give an error message.

You can find further information about the run-time library on this chapter of the GNAT User’s Guide Supplement for Cross Platforms.\footnote{https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx/the_gnat_configurable_run_time_facility.html}
69.2 Low Level Programming

69.2.1 Representation Clauses

We’ve seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let’s start with the following example:

[Ada]

```ada
type R is record
  V : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record
with Pack;
```

[C]

```c
struct R {
  unsigned int v:8;
  bool b1;
  bool b2;
};
```

The Ada and the C code above both represent efforts to create an object that’s as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C and Ada code are equivalent in this particular example, there’s an interesting semantic difference. In C, the number of bits required by each field needs to be specified. Here, we’re stating that v is only 8 bits, effectively representing values from 0 to 255. In Ada, it’s the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The Pack aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components. We’ll see more details about the Pack aspect in the sections about bitwise operations (page 1486) and mapping structures to bit-fields (page 1488) in chapter 6.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here’s how to specify a specific data layout based on the previous example:

[Ada]

```ada
for R use record
  -- Occupy the first bit of the first byte.
```

(continues on next page)
We omit the `with` Pack directive and instead use a record representation clause following the record declaration. The compiler is directed to spread objects of type \( R \) across two bytes. The layout we're specifying here is fairly inefficient to work with on any machine, but you can have the compiler construct the most efficient methods for access, rather than coding your own machine-dependent bit-level methods manually.

### 69.2.2 Embedded Assembly Code

When performing low-level development, such as at the kernel or hardware driver level, there can be times when it is necessary to implement functionality with assembly code.

Every Ada compiler has its own conventions for embedding assembly code, based on the hardware platform and the supported assembler(s). Our examples here will work with GNAT and GCC on the x86 architecture.

All x86 processors since the Intel Pentium offer the `rdtsc` instruction, which tells us the number of cycles since the last processor reset. It takes no inputs and places an unsigned 64-bit value split between the edx and eax registers.

GNAT provides a subprogram called `System.Machine_Code.As`m that can be used for assembly code insertion. You can specify a string to pass to the assembler as well as source-level variables to be used for input and output:

[Ada]

```ada
with Interfaces; use Interfaces;  
function Get_Processor_Cycles return Unsigned_64 is  
  Low, High : Unsigned_32;  
  Counter : Unsigned_64;  
begin  
  Asm ("rdtsc",  
    Outputs =>  
      (Unsigned_32'Asm_Output ("=a", High),  
       Unsigned_32'Asm_Output ("=d", Low)),  
    Volatile => True);  
  Counter :=  
      Unsigned_64 (High) * 2 ** 32 +  
      Unsigned_64 (Low);  
return Counter;  
end Get_Processor_Cycles;  
```

**Code block metadata**
The Unsigned_32'Asm_Output clauses above provide associations between machine registers and source-level variables to be updated. =a and =d refer to the eax and edx machine registers, respectively. The use of the Unsigned_32 and Unsigned_64 types from package Interfaces ensures correct representation of the data. We assemble the two 32-bit values to form a single 64-bit value.

We set the Volatile parameter to True to tell the compiler that invoking this instruction multiple times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the eax and edx registers to implement the High and Low variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User's Guide, and the GNAT Reference manual.

### 69.3 Interrupt Handling

Handling interrupts is an important aspect when programming embedded devices. Interrupts are used, for example, to indicate that a hardware or software event has happened. Therefore, by handling interrupts, an application can react to external events.

Ada provides built-in support for handling interrupts. We can process interrupts by attaching a handler — which must be a protected procedure — to it. In the declaration of the protected procedure, we use the Attach_Handler aspect and indicate which interrupt we want to handle.

Let's look into a code example that traps the quit interrupt (SIGQUIT) on Linux:

[Ada]

```
with System.OS_Interface;

package Signal_Handlers is

  protected type Quit_Handler is
    function Requested return Boolean;
  private
    Quit_Request : Boolean := False;
  -- Declaration of an interrupt handler for the "quit" interrupt:
  --
  procedure Handle_SigQuit
    with Attach_Handler => System.OS_Interface.SIGQUIT;

end Signal_Handlers;
```

Listing 3: signal_handlers.adb

```
with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)
The specification of the Signal_Handlers package from this example contains the declaration of Quit_Handler, which is a protected type. In the private part of this protected type, we declare the Handle_Quit_Signal procedure. By using the Attach_Handler aspect in the declaration of Handle_Quit_Signal and indicating the quit interrupt (System.OS_Interface.SIGQUIT), we’re instructing the operating system to call this procedure for any quit request. So when the user presses CTRL+\ on their keyboard, for example, the application will behave as follows:

• the operating system calls the Handle_Quit_Signal procedure, which displays a message to the user (“Quit request detected!”) and sets a Boolean variable — Quit_Request, which is declared in the Quit_Handler type;

• the main application checks the status of the quit handler by calling the Requested function as part of the while True loop;
  - This call is in the exit when Quit.Requested line.
  - The Requested function returns True in this case because the Quit_Request flag was set by the Handle_Quit_Signal procedure.

• the main applications exits the loop, displays a message and finishes.

Note that the code example above isn't portable because it makes use of interrupts from the Linux operating system. When programming embedded devices, we would use instead

69.3. Interrupt Handling
Learning Ada

the interrupts available on those specific devices.

Also note that, in the example above, we're declaring a static handler at compilation time. If you need to make use of dynamic handlers, which can be configured at runtime, you can use the subprograms from the Ada.Interrupts package. This package includes not only a version of Attach_Handler as a procedure, but also other procedures such as:

- Exchange_Handler, which lets us exchange, at runtime, the current handler associated with a specific interrupt by a different handler;
- Detach_Handler, which we can use to remove the handler currently associated with a given interrupt.

Details about the Ada.Interrupts package are out of scope for this course. We'll discuss them in a separate, more advanced course in the future. You can find some information about it in the Interrupts appendix of the Ada Reference Manual.

69.4 Dealing with Absence of FPU with Fixed Point

Many numerical applications typically use floating-point types to compute values. However, in some platforms, a floating-point unit may not be available. Other platforms may have a floating-point unit, but using it in certain numerical algorithms can be prohibitive in terms of performance. For those cases, fixed-point arithmetic can be a good alternative.

The difference between fixed-point and floating-point types might not be so obvious when looking at this code snippet:

[Ada]

```
package Fixed_Definitions is

  D : constant := 2.0 ** (-31);

  type Fixed is delta D range -1.0 .. 1.0 - D;

end Fixed_Definitions;
```

Listing 5: fixed_definitions.ads

```
with Ada.Text_IO; use Ada.Text_IO;
with Fixed_Definitions; use Fixed_Definitions;

procedure Show_Float_And_Fixed_Point is

  Float_Value : Float := 0.25;
  Fixed_Value : Fixed := 0.25;

begin

  Float_Value := Float_Value + 0.25;
  Fixed_Value := Fixed_Value + 0.25;

  Put_Line ("Float_Value = " & Float'Image (Float_Value));
  Put_Line ("Fixed_Value = " & Fixed'Image (Fixed_Value));

end Show_Float_And_Fixed_Point;
```

Listing 6: show_float_and_fixed_point.adb

Code block metadata

http://www.ada-auth.org/standards/12aarm/html/AA-C-3-2.html
In this example, the application will show the value 0.5 for both `Float_Value` and `Fixed_Value`.

The major difference between floating-point and fixed-point types is in the way the values are stored. Values of ordinary fixed-point types are, in effect, scaled integers. The scaling used for ordinary fixed-point types is defined by the type's `small`, which is derived from the specified `delta` and, by default, is a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types. In that sense, ordinary fixed-point types can be thought of being close to the actual representation on the machine. In fact, ordinary fixed-point types make use of the available integer shift instructions, for example.

Another difference between floating-point and fixed-point types is that Ada doesn't provide standard fixed-point types — except for the `Duration` type, which is used to represent an interval of time in seconds. While the Ada standard specifies floating-point types such as `Float` and `Long_Float`, we have to declare our own fixed-point types. Note that, in the previous example, we have used a fixed-point type named `Fixed`: this type isn't part of the standard, but must be declared somewhere in the source-code of our application.

The syntax for an ordinary fixed-point type is

```
type <type_name> is delta <delta_value> range <lower_bound> .. <upper_bound>;
```

By default, the compiler will choose a scale factor, or `small`, that is a power of 2 no greater than `<delta_value>`.

For example, we may define a normalized range between -1.0 and 1.0 as following:

[Ada]

Listing 7: normalized_fixed_point_type.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Normalized_Fixed_Point_Type is
  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;
begin
  Put_Line ("TQ31 requires " & Integer'Image (TQ31'Size) & " bits");
  Put_Line ("The delta value of TQ31 is " & TQ31'Delta);
  Put_Line ("The minimum value of TQ31 is " & TQ31'First);
  Put_Line ("The maximum value of TQ31 is " & TQ31'Last);
end Normalized_Fixed_Point_Type;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Normalized_Fixed_Point_Type
MD5: 2fe6e9f9bd28d2cfab959d1c0737280b

Runtime output

```
TQ31 requires 32 bits
The delta value of TQ31 is 0.0000000005
The minimum value of TQ31 is -1.0000000000
The maximum value of TQ31 is 0.9999999995
```
In this example, we are defining a 32-bit fixed-point data type for our normalized range. When running the application, we notice that the upper bound is close to one, but not exactly one. This is a typical effect of fixed-point data types — you can find more details in this discussion about the Q format\(^{328}\). We may also rewrite this code with an exact type definition:

[Ada]

Listing 8: normalized_adapted_fixed_point_type.ads

```ada
package Normalized_Adapted_Fixed_Point_Type is
  type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);
end Normalized_Adapted_Fixed_Point_Type;
```

We may also use any other range. For example:

[Ada]

Listing 9: custom_fixed_point_range.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics; use Ada.Numerics;

procedure Custom_Fixed_Point_Range is
  type Inv_Trig is delta 2.0 ** (-15) * Pi range -Pi / 2.0 .. Pi / 2.0;
begin
  Put_Line ("Inv_Trig requires ", Integer’Image (Inv_Trig’Size) & " bits");
  Put_Line ("The delta value of Inv_Trig is ", Inv_Trig’Image (Inv_Trig’Delta));
  Put_Line ("The minimum value of Inv_Trig is ", Inv_Trig’Image (Inv_Trig’First));
  Put_Line ("The maximum value of Inv_Trig is ", Inv_Trig’Image (Inv_Trig’Last));
end Custom_Fixed_Point_Range;
```

In this example, we are defining a 16-bit type called Inv_Trig, which has a range from -π/2 to π/2. All standard operations are available for fixed-point types. For example:

[Ada]

\(^{328}\) https://en.wikipedia.org/wiki/Q_(number_format)
**Listing 10: fixed_point_op.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Fixed_Point_Op is
  type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);
  A, B, R : TQ31;
begin
  A := 0.25;
  B := 0.50;
  R := A + B;
  Put_Line ("R is " & TQ31'Image (R));
end Fixed_Point_Op;
```

**Code block metadata**

MD5: 78bafd93b25da898c00cc38c9d518e2a

**Runtime output**

R is 0.7500000000

As expected, R contains 0.75 after the addition of A and B.

In the case of C, since the language doesn’t support fixed-point arithmetic, we need to emulate it using integer types and custom operations via functions. Let’s look at this very rudimentary example:

```c
#include <stdio.h>
#include <math.h>

#define SHIFT_FACTOR 32
#define TO_FIXED(x) ((int) ((x) * pow (2.0, SHIFT_FACTOR - 1)))
#define TO_FLOAT(x) ((float) ((double)(x) * (double)pow (2.0, -(SHIFT_FACTOR - 1))))

typedef int fixed;

fixed add (fixed a, fixed b)
{
    return a + b;
}

fixed mult (fixed a, fixed b)
{
    return (fixed)(((long)a * (long)b) >> (SHIFT_FACTOR - 1));
}

void display_fixed (fixed x)
{
    printf("value (integer) = %d\n", x);
    printf("value (float) = %.3f\n\n", TO_FLOAT (x));
}

int main(int argc, const char * argv[]) {
    (continues on next page)
}
```

69.4. Dealing with Absence of FPU with Fixed Point
Here, we declare the fixed-point type fixed based on `int` and two operations for it: addition (via the `add` function) and multiplication (via the `mult` function). Note that, while fixed-point addition is quite straightforward, multiplication requires right-shifting to match the correct internal representation. In Ada, since fixed-point operations are part of the language specification, they don't need to be emulated. Therefore, no extra effort is required from the programmer.

Also note that the example above is very rudimentary, so it doesn't take some of the side-effects of fixed-point arithmetic into account. In C, you have to manually take all side-effects deriving from fixed-point arithmetic into account, while in Ada, the compiler takes care of selecting the right operations for you.
69.5 Volatile and Atomic data

Ada has built-in support for handling both volatile and atomic data. Let's start by discussing volatile objects.

69.5.1 Volatile

A volatile object can be described as an object in memory whose value may change between two consecutive memory accesses of a process A — even if process A itself hasn't changed the value. This situation may arise when an object in memory is being shared by multiple threads. For example, a thread B may modify the value of that object between two read accesses of a thread A. Another typical example is the one of memory-mapped I/O, where the hardware might be constantly changing the value of an object in memory.

Because the value of a volatile object may be constantly changing, a compiler cannot generate code that stores the value of that object into a register and use the value from the register in subsequent operations. Storing into a register is avoided because, if the value is stored there, it would be outdated if another process had changed the volatile object in the meantime. Instead, the compiler generates code in such a way that the process must read the value of the volatile object from memory for each access.

Let's look at a simple example of a volatile variable in C:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[]) {
    volatile double val = 0.0;
    int i;
    for (i = 0; i < 1000; i++) {
        val += i * 2.0;
    }
    printf("val: %5.3f\n", val);
    return 0;
}
```

In this example, `val` has the modifier `volatile`, which indicates that the compiler must handle `val` as a volatile object. Therefore, each read and write access in the loop is performed by accessing the value of `val` in the memory.

This is the corresponding implementation in Ada:

---

329 https://en.wikipedia.org/wiki/Volatile_(computer_programming)
330 https://en.wikipedia.org/wiki/Memory-mapped_I/O
Listing 13: show_volatile_object.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Volatile_Object is
   Val : Long_Float with Volatile;
begin
   Val := 0.0;
   for I in 0 .. 999 loop
      Val := Val + 2.0 * Long_Float (I);
   end loop;
   Put_Line ("Val: " & Long_Float'Image (Val));
end Show_Volatile_Object;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Volatile_Object_Ada
MD5: aa1e276e64e69813bf3e3ef39f3dd47

Runtime output

Val: 9.99000000000000E+05

In this example, Val has the Volatile aspect, which makes the object volatile. We can also use the Volatile aspect in type declarations. For example:

Listing 14: show_volatile_type.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Volatile_Type is
   type Volatile_Long_Float is new Long_Float with Volatile;
   Val : Volatile_Long_Float;
begin
   Val := 0.0;
   for I in 0 .. 999 loop
      Val := Val + 2.0 * Volatile_Long_Float (I);
   end loop;
   Put_Line ("Val: " & Volatile_Long_Float'Image (Val));
end Show_Volatile_Type;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Volatile_Type
MD5: 41ecf028803a58ce244c421eaeb118e4

Runtime output

Val: 9.99000000000000E+05

Here, we’re declaring a new type Volatile_Long_Float based on the Long_Float type and using the Volatile aspect. Any object of this type is automatically volatile.

In addition to that, we can declare components of an array to be volatile. In this case, we can use the Volatile_Components aspect in the array declaration. For example:
Listing 15: show_volatile_array_components.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Volatile_Array_Components is
    Arr : array (1 .. 2) of Long_Float with Volatile_Components;
begin
    Arr := (others => 0.0);
    for I in 0 .. 999 loop
        Arr (1) := Arr (1) + 2.0 * Long_Float (I);
        Arr (2) := Arr (2) + 10.0 * Long_Float (I);
    end loop;
    Put_Line ("Arr (1): " & Long_Float'Image (Arr (1)));
    Put_Line ("Arr (2): " & Long_Float'Image (Arr (2)));
end Show_Volatile_Array_Components;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Volatile_Array_Components
MD5: 601d61dd01888c60ae1a51ec513138d5

**Runtime output**

<table>
<thead>
<tr>
<th>Arr (1)</th>
<th>9.99000000000000E+05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arr (2)</td>
<td>4.99500000000000E+06</td>
</tr>
</tbody>
</table>

Note that it's possible to use the Volatile aspect for the array declaration as well:

[Ada]

```
Arr : array (1 .. 2) of Long_Float with Volatile;
```

### 69.5.2 Atomic

An atomic object is an object that only accepts atomic reads and updates. The Ada standard specifies that "for an atomic object (including an atomic component), all reads and updates of the object as a whole are indivisible." In this case, the compiler must generate Assembly code in such a way that reads and updates of an atomic object must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes.

**In other contexts**

Generally, we can say that operations are said to be atomic when they can be completed without interruptions. This is an important requirement when we're performing operations on objects in memory that are shared between multiple processes.

This definition of atomicity above is used, for example, when implementing databases. However, for this section, we're using the term "atomic" differently. Here, it really means that reads and updates must be performed with a single Assembly instruction.

For example, if we have a 32-bit object composed of four 8-bit bytes, the compiler cannot generate code to read or update the object using four 8-bit store/load instructions, or even two 16-bit store/load instructions. In this case, in order to maintain atomicity, the compiler must generate code using one 32-bit store/load instruction.

Because of this strict definition, we might have objects for which the Atomic aspect cannot be specified. Lots of machines support integer types that are larger than the native word-
sized integer. For example, a 16-bit machine probably supports both 16-bit and 32-bit integers, but only 16-bit integer objects can be marked as atomic — or, more generally, only objects that fit into at most 16 bits.

Atomicity may be important, for example, when dealing with shared hardware registers. In fact, for certain architectures, the hardware may require that memory-mapped registers are handled atomically. In Ada, we can use the Atomic aspect to indicate that an object is atomic. This is how we can use the aspect to declare a shared hardware register:

[Ada]

Listing 16: show_shared_hw_register.adb

```ada
with System;

procedure Show_Shared_HW_Register is
  R : Integer with Atomic, Address => System'Address (16#FFFF00A0#);
begin
  null;
end Show_Shared_HW_Register;
```

**Code block metadata**

MD5: 7ef148adfd393819fc3fbc25eb45afe46

Note that the Address aspect allows for assigning a variable to a specific location in the memory. In this example, we're using this aspect to specify the address of the memory-mapped register. We'll discuss more about the Address aspect later in the section about mapping structures to bit-fields (page 1488) (in chapter 6).

In addition to atomic objects, we can declare atomic types and atomic array components — similarly to what we've seen before for volatile objects. For example:

[Ada]

Listing 17: show_shared_hw_register.adb

```ada
with System;

procedure Show_Shared_HW_Register is
  type Atomic_Integer is new Integer with Atomic;
  R : Atomic_Integer with Address => System'Address (16#FFFF00A0#);
  Arr : array (1 .. 2) of Integer with Atomic_Components;
begin
  null;
end Show_Shared_HW_Register;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Atomic_Types_Arrays
MD5: 11475b5152087eff7f36abfe2c5ae9a1

In this example, we're declaring the Atomic_Integer type, which is an atomic type. Objects of this type — such as R in this example — are automatically atomic. This example also includes the declaration of the Arr array, which has atomic components.
69.6 Interfacing with Devices

Previously, we've seen that we can use representation clauses (page 1430) to specify a particular layout for a record type. As mentioned before, this is useful when interfacing with hardware, drivers, or communication protocols. In this section, we'll extend this concept for two specific use-cases: register overlays and data streams. Before we discuss those use-cases, though, we'll first explain the Size aspect and the Size attribute.

69.6.1 Size aspect and attribute

The Size aspect indicates the minimum number of bits required to represent an object. When applied to a type, the Size aspect is telling the compiler to not make record or array components of a type T any smaller than X bits. Therefore, a common usage for this aspect is to just confirm expectations: developers specify 'Size to tell the compiler that T should fit X bits, and the compiler will tell them if they are right (or wrong).

When the specified size value is larger than necessary, it can cause objects to be bigger in memory than they would be otherwise. For example, for some enumeration types, we could say for type Enum 'Size use 32; when the number of literals would otherwise have required only a byte. That's useful for unchecked conversions because the sizes of the two types need to be the same. Likewise, it's useful for interfacing with C, where enum types are just mapped to the int type, and thus larger than Ada might otherwise require. We'll discuss unchecked conversions later in the course (page 1502).

Let's look at an example from an earlier chapter:

[Ada]

Listing 18: my_device_types.ads

```ada
package My_Device_Types is
  type UInt10 is mod 2 ** 10
    with Size => 10;
end My_Device_Types;
```

Here, we're saying that objects of type UInt10 must have at least 10 bits. In this case, if the code compiles, it is a confirmation that such values can be represented in 10 bits when packed into an enclosing record or array type.

If the size specified was larger than what the compiler would use by default, then it could affect the size of objects. For example, for UInt10, anything up to and including 16 would make no difference on a typical machine. However, anything over 16 would then push the compiler to use a larger object representation. That would be important for unchecked conversions, for example.

The Size attribute indicates the number of bits required to represent a type or an object. We can use the size attribute to retrieve the size of a type or of an object:

[Ada]
Listing 19: show_device_types.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with My_Device_Types; use My_Device_Types;

procedure Show_Device_Types is
    UInt10_Obj : constant UInt10 := 0;
begin
    Put_Line ("Size of UInt10 type: " & Positive'Image (UInt10'Size));
    Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_Obj'Size));
end Show_Device_Types;
```

Code block metadata

Project: Courses.Ada_For_EMBEDDED_C_Dev.Perspective.Size_Aspect
MD5: 4e46ad9cf54276b381b960672daa03b9

Runtime output

Size of UInt10 type: 10
Size of UInt10 object: 16

Here, we're retrieving the actual sizes of the UInt10 type and an object of that type. Note that the sizes don't necessarily need to match. For example, although the size of UInt10 type is expected to be 10 bits, the size of UInt10_Obj may be 16 bits, depending on the platform. Also, components of this type within composite types (arrays, records) will probably be 16 bits as well unless they are packed.

69.6.2 Register overlays

Register overlays make use of representation clauses to create a structure that facilitates manipulating bits from registers. Let's look at a simplified example of a power management controller containing registers such as a system clock enable register. Note that this example is based on an actual architecture:

[Ada]

Listing 20: registers.ads

```ada
with System;

package Registers is

    type Bit is mod 2 ** 1
        with Size => 1;
    type UInt5 is mod 2 ** 5
        with Size => 5;
    type UInt10 is mod 2 ** 10
        with Size => 10;

    subtype USB_Clock_Enable is Bit;

    -- System Clock Enable Register
    type PMC_SCER_Register is record
        -- Reserved bits
        Reserved_0_4 : UInt5 := 16#0#;
        -- Write-only. Enable USB FS Clock
        USBCLK : USB_Clock_Enable := 16#0#;
    end Registers;
```

(continues on next page)
-- Reserved bits
Reserved_6_15 : UInt10 := 16#'0#;
end record
with
Volatile,
Size => 16,
Bit_Order => System.Low_Order_First;
for PMC_SCER_Register use record
Reserved_0_4 at 0 range 0 .. 4;
USBCLK at 0 range 5 .. 5;
Reserved_6_15 at 0 range 6 .. 15;
end record;

-- Power Management Controller

type PMC_Peripheral is record
-- System Clock Enable Register
PMC_SCER : aliased PMC_SCER_Register;
-- System Clock Disable Register
PMC_SCDR : aliased PMC_SCER_Register;
end record
with Volatile;
for PMC_Peripheral use record
-- 16-bit register at byte 0
PMC_SCER at 16#'0# range 0 .. 15;
-- 16-bit register at byte 2
PMC_SCDR at 16#'2# range 0 .. 15;
end record;

-- Power Management Controller
PMC_Periph : aliased PMC_Peripheral
with Import, Address => System'Address (16#'400E0600#);
end Registers;

First, we declare the system clock enable register — this is PMC_SCER_Register type in the code example. Most of the bits in that register are reserved. However, we're interested in bit #5, which is used to activate or deactivate the system clock. To achieve a correct representation of this bit, we do the following:

- We declare the USBCLK component of this record using the USB_Clock_Enable type, which has a size of one bit; and
- We use a representation clause to indicate that the USBCLK component is specifically at bit #5 of byte #0.

After declaring the system clock enable register and specifying its individual bits as components of a record type, we declare the power management controller type — PMC_Peripheral record type in the code example. Here, we declare two 16-bit registers as record components of PMC_Peripheral. These registers are used to enable or disable the system clock. The strategy we use in the declaration is similar to the one we've just seen above:

- We declare these registers as components of the PMC_Peripheral record type;
- We use a representation clause to specify that the PMC_SCER register is at byte #0 and the PMC_SCDR register is at byte #2.
Since these registers have 16 bits, we use a range of bits from 0 to 15.

The actual power management controller becomes accessible by the declaration of the PMC_Periph object of PMC_Peripheral type. Here, we specify the actual address of the memory-mapped registers (400E0600 in hexadecimal) using the Address aspect in the declaration. When we use the Address aspect in an object declaration, we're indicating the address in memory of that object.

Because we specify the address of the memory-mapped registers in the declaration of PMC_Periph, this object is now an overlay for those registers. This also means that any operation on this object corresponds to an actual operation on the registers of the power management controller. We'll discuss more details about overlays in the section about mapping structures to bit-fields (page 1488) (in chapter 6).

Finally, in a test application, we can access any bit of any register of the power management controller with simple record component selection. For example, we can set the USBCLK bit of the PMC_SCER register by using PMC_Periph.PMC_SCER.USBCLK:

```ada
with Registers;

procedure Enable_USB_Clock is
begin
  Registers.PMC_Periph.PMC_SCER.USBCLK := 1;
end Enable_USB_Clock;
```

This code example makes use of many aspects and keywords of the Ada language. One of them is the Volatile aspect, which we've discussed in the section about volatile and atomic objects (page 1439). Using the Volatile aspect for the PMC_SCER_Register type ensures that objects of this type won't be stored in a register.

In the declaration of the PMC_SCER_Register record type of the example, we use the Bit_Order aspect to specify the bit ordering of the record type. Here, we can select one of these options:

- High_Order_First: first bit of the record is the most significant bit;
- Low_Order_First: first bit of the record is the least significant bit.

The declarations from the Registers package also makes use of the Import, which is sometimes necessary when creating overlays. When used in the context of object declarations, it avoids default initialization (for data types that have it.). Aspect Import will be discussed in the section that explains how to map structures to bit-fields (page 1488) in chapter 6. Please refer to that chapter for more details.

### Details about 'Size

In the example above, we're using the Size aspect in the declaration of the PMC_SCER_Register type. In this case, the effect is that it has the compiler confirm that the record type will fit into the expected 16 bits.

That's what the aspect does for type PMC_SCER_Register in the example above, as well as for the types Bit, UInt5 and UInt10. For example, we may declare a stand-alone object of type Bit:
Listing 22: show_bit_declaration.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Bit_Declaration is

  type Bit is mod 2 ** 1
    with Size => 1;

  B : constant Bit := 0;
  -- ^ Although Bit'Size is 1, B'Size is almost certainly 8

begin
  Put_Line ("Bit'Size = " & Positive'Image (Bit'Size));
  Put_Line ("B'Size = " & Positive'Image (B'Size));
end Show_Bit_Declaration;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Bit_Declaration
MD5: 1778bb96b4b772928b5bdedefee7c596

Runtime output

Bit'Size = 1
B'Size = 8

In this case, B is almost certainly going to be 8-bits wide on a typical machine, even though the language requires that Bit'Size is 1 by default.

In the declaration of the components of the PMC_Peripheral record type, we use the aliased keyword to specify that those record components are accessible via other paths besides the component name. Therefore, the compiler won't store them in registers. This makes sense because we want to ensure that we're accessing specific memory-mapped registers, and not registers assigned by the compiler. Note that, for the same reason, we also use the aliased keyword in the declaration of the PMC_Periph object.

69.6.3 Data streams

Creating data streams — in the context of interfacing with devices — means the serialization of arbitrary information and its transmission over a communication channel. For example, we might want to transmit the content of memory-mapped registers as byte streams using a serial port. To do this, we first need to get a serialized representation of those registers as an array of bytes, which we can then transmit over the serial port.

Serialization of arbitrary record types — including register overlays — can be achieved by declaring an array of bytes as an overlay. By doing this, we're basically interpreting the information from those record types as bytes while ignoring their actual structure — i.e. their components and representation clause. We'll discuss details about overlays in the section about mapping structures to bit-fields (page 1488) (in chapter 6).

Let's look at a simple example of serialization of an arbitrary record type:

[Ada]

Listing 23: arbitrary_types.ads

package Arbitrary_Types is

  type Arbitrary_Record is record

(continues on next page)
A : Integer;
B : Integer;
C : Integer;
end record;
end Arbitrary_Types;

with Arbitrary_Types;
procedure Serialize_Data (Some_Object : Arbitrary_Types.Arbitrary_Record);

with Arbitrary_Types;
with Serialize_Data;
procedure Data_Stream_Declaration is
dummy_Object : Arbitrary_Types.Arbitrary_Record;
begin
Serialize_Data (dummy_Object);
end Data_Stream_Declaration;

The most important part of this example is the implementation of the Serialize_Data procedure, where we declare Raw_TX as an overlay for our arbitrary object (Some_Object of Arbitrary_Record type). In simple terms, by writing with Address => Some_Object'Address; in the declaration of Raw_TX, we're specifying that Raw_TX and Some_Object have the same address in memory. Here, we are:
• taking the address of Some_Object — using the Address attribute —, and then
• using it as the address of Raw_TX — which is specified with the Address aspect.

By doing this, we're essentially saying that both Raw_TX and Some_Object are different representations of the same object in memory.

Because the Raw_TX overlay is completely agnostic about the actual structure of the record type, the Arbitrary_Record type could really be anything. By declaring Raw_TX, we create an array of bytes that we can use to stream the information from Some_Object.

We can use this approach and create a data stream for the register overlay example that we've seen before. This is the corresponding implementation:

[Ada]

Listing 27: registers.ads

```ada
with System;

package Registers is

  type Bit is mod 2 ** 1
    with Size => 1;
  type UInt5 is mod 2 ** 5
    with Size => 5;
  type UInt10 is mod 2 ** 10
    with Size => 10;

  subtype USB_Clock_Enable is Bit;

  -- System Clock Register
  type PMC_SCER_Register is record
    -- Reserved bits
    Reserved_0_4 : UInt5 := 16#0#
    -- Write-only. Enable USB FS Clock
    USBClk : USB_Clock_Enable := 16#0#
    -- Reserved bits
    Reserved_6_15 : UInt10 := 16#0#
  end record
    with
    Volatile,
    Size => 16,
    Bit_Order => System.Low_Order_First;

  for PMC_SCER_Register use record
    Reserved_0_4 at 0 range 0 .. 4;
    USBClk at 0 range 5 .. 5;
    Reserved_6_15 at 0 range 6 .. 15;
  end record;

  -- Power Management Controller
  type PMC_Peripheral is record
    -- System Clock Enable Register
    PMC_SCER : aliased PMC_SCER_Register;
    -- System Clock Disable Register
    PMC_SCDR : aliased PMC_SCER_Register;
  end record
    with Volatile;

  for PMC_Peripheral use record
    -- 16-bit register at byte 0
    PMC_SCER at 16#0# range 0 .. 15;
    -- 16-bit register at byte 2
```

(continues on next page)
PMC_SCDR at 16#2# range 0 .. 15;
end record;

-- Power Management Controller
PMC_Periph : aliased PMC_Peripheral;
-- with Import, Address => System'To_Address (16#400E0600#);
end Registers;

Listing 28: serial_ports.ads

package Serial_Ports is
  type UByte is new Natural range 0 .. 255
  with Size => 8;
  type UByte_Array is array (Positive range <>) of UByte;
  type Serial_Port is null record;
  procedure Read (Port : in out Serial_Port;
                  Data : out UByte_Array);
  procedure Write (Port : in out Serial_Port;
                  Data : UByte_Array);
end Serial_Ports;

Listing 29: serial_ports.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Serial_Ports is
  procedure Display (Data : UByte_Array) is
  begin
    Put_Line ("---- Data ----");
    for E of Data loop
      Put_Line (UByte'Image (E));
    end loop;
    Put_Line ("--------------");
  end Display;

  procedure Read (Port : in out Serial_Port;
                 Data : out UByte_Array) is
    pragma Unreferenced (Port);
  begin
    Put_Line ("Reading data...");
    Data := (0, 0, 32, 0);
  end Read;

  procedure Write (Port : in out Serial_Port;
                  Data : UByte_Array) is
    pragma Unreferenced (Port);
  begin
    Put_Line ("Writing data...");
    Display (Data);
  end Write;
end Serial_Ports;
Listing 30: data_stream.ads

```ada
with Serial_Ports; use Serial_Ports;
with Registers; use Registers;

package Data_Stream is

  procedure Send (Port : in out Serial_Port;
                  PMC : PMC_Peripheral);

  procedure Receive (Port : in out Serial_Port;
                     PMC : out PMC_Peripheral);

end Data_Stream;
```

Listing 31: data_stream.adb

```ada
package body Data_Stream is

  procedure Send (Port : in out Serial_Port;
                   PMC : PMC_Peripheral)
  is
    Raw_TX : UByte_Array (1 .. PMC'Size / 8)
    with Address => PMC'Address;
  begin
    Write (Port => Port,
            Data => Raw_TX);
  end Send;

  procedure Receive (Port : in out Serial_Port;
                     PMC : out PMC_Peripheral)
  is
    Raw_TX : UByte_Array (1 .. PMC'Size / 8)
    with Address => PMC'Address;
  begin
    Read (Port => Port,
           Data => Raw_TX);
  end Receive;

end Data_Stream;
```

Listing 32: test_data_stream.adb

```ada
with Ada.Text_IO;
with Registers;
with Data_Stream;
with Serial_Ports;

procedure Test_Data_Stream is

  procedure Display_Registers is
    use Ada.Text_IO;
  begin
    Put_Line ("---- Registers ----");
    Put_Line ("PMC_SCER.USBCCLK: 
              & Registers.PMC_Periph.PMC_SCER.USBCCLK'Image);
    Put_Line ("PMC_SCDR.USBCCLK: 
              & Registers.PMC_Periph.PMC_SCDR.USBCCLK'Image);
    Put_Line ("-------------- ----");
  end Display_Registers;
```
(continues on next page)
Port : Serial_Ports.Serial_Port;

begin
  Registers.PMC_Periph.PMC_SCER.USBCLOCK := 1;
  Registers.PMC_Periph.PMC_SCDR.USBCLOCK := 1;
  Display_Registers;
  Data_Stream.Send (Port => Port,
                    PMC => Registers.PMC_Periph);
  Data_Stream.Receive (Port => Port,
                        PMC => Registers.PMC_Periph);
  Display_Registers;
end Test_Data_Stream;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Data_Stream
MD5: 3f4e1a184e52a83b1b9de9e3d5cb43bf

Runtime output

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PMC_SCER.USBCLOCK:</td>
<td>1</td>
</tr>
<tr>
<td>PMC_SCDR.USBCLOCK:</td>
<td>1</td>
</tr>
</tbody>
</table>

Writing data...

<table>
<thead>
<tr>
<th>Data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Reading data...

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PMC_SCER.USBCLOCK:</td>
<td>0</td>
</tr>
<tr>
<td>PMC_SCDR.USBCLOCK:</td>
<td>1</td>
</tr>
</tbody>
</table>

In this example, we can find the overlay in the implementation of the Send and Receive procedures from the Data_Stream package. Because the overlay doesn't need to know the internals of the PMC_Peripheral type, we're declaring it in the same way as in the previous example (where we created an overlay for Some_Object). In this case, we're creating an overlay for the PMC parameter.

Note that, for this section, we're not really interested in the details about the serial port. Thus, package Serial_Ports in this example is just a stub. However, because the Serial_Port type in that package only sees arrays of bytes, after implementing an actual serial port interface for a specific device, we could create data streams for any type.
69.7 ARM and svd2ada

As we’ve seen in the previous section about *interfacing with devices* (page 1443), Ada offers powerful features to describe low-level details about the hardware architecture without giving up its strong typing capabilities. However, it can be cumbersome to create a specification for all those low-level details when you have a complex architecture. Fortunately, for ARM Cortex-M devices, the GNAT toolchain offers an Ada binding generator called *svd2ada*, which takes CMSIS-SVD descriptions for those devices and creates Ada specifications that match the architecture. CMSIS-SVD description files are based on the Cortex Microcontroller Software Interface Standard (CMSIS), which is a hardware abstraction layer for ARM Cortex microcontrollers.

Please refer to the svd2ada project page[^331] for details about this tool.

[^331]: https://github.com/AdaCore/svd2ada
70.1 Understanding Exceptions and Dynamic Checks

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering "exceptions" that interrupt the normal flow of execution. For example, an exception is raised by an attempt to access an array component via an index that is out of bounds. This simple check precludes exploits based on buffer overflow. Several other cases also raise language-defined exceptions, such as scalar range constraint violations and null pointer dereferences. Developers may declare and raise their own application-specific exceptions too. (Exceptions are software artifacts, although an implementation may map hardware events to exceptions.)

Exceptions are raised during execution of what we will loosely define as a "frame." A frame is a language construct that has a call stack entry when called, for example a procedure or function body. There are a few other constructs that are also pertinent but this definition will suffice for now.

Frames have a sequence of statements implementing their functionality. They can also have optional "exception handlers" that specify the response when exceptions are "raised" by those statements. These exceptions could be raised directly within the statements, or indirectly via calls to other procedures and functions.

For example, the frame below is a procedure including three exceptions handlers:

```
procedure P is
begin
  Statements_That_Might_Raise_Exceptions;
exception
  when A =>
    Handle_A;
  when B =>
    Handle_B;
  when C =>
    Handle_C;
end P;
```

The three exception handlers each start with the word `when` (lines 5, 7, and 9). Next comes one or more exception identifiers, followed by the so-called "arrow." In Ada, the arrow always associates something on the left side with something on the right side. In this case, the left side is the exception name and the right side is the handler's code for that exception.
Each handler's code consists of an arbitrary sequence of statements, in this case specific procedures called in response to those specific exceptions. If exception A is raised we call procedure Handle_A (line 6), dedicated to doing the actual work of handling that exception. The other two exceptions are dealt with similarly, on lines 8 and 10.

Structurally, the exception handlers are grouped together and textually separated from the rest of the code in a frame. As a result, the sequence of statements representing the normal flow of execution is distinct from the section representing the error handling. The reserved word exception separates these two sections (line 4 above). This separation helps simplify the overall flow, increasing understandability. In particular, status result codes are not required so there is no mixture of error checking and normal processing. If no exception is raised the exception handler section is automatically skipped when the frame exits.

Note how the syntactic structure of the exception handling section resembles that of an Ada case statement. The resemblance is intentional, to suggest similar behavior. When something in the statements of the normal execution raises an exception, the corresponding exception handler for that specific exception is executed. After that, the routine completes. The handlers do not "fall through" to the handlers below. For example, if exception B is raised, procedure Handle_B is called but Handle_C is not called. There's no need for a break statement, just as there is no need for it in a case statement. (There's no break statement in Ada anyway.)

So far, we've seen a frame with three specific exceptions handled. What happens if a frame has no handler for the actual exception raised? In that case the run-time library code goes "looking" for one.

Specifically, the active exception is propagated up the dynamic call chain. At each point in the chain, normal execution in that caller is abandoned and the handlers are examined. If that caller has a handler for the exception, the handler is executed. That caller then returns normally to its caller and execution continues from there. Otherwise, propagation goes up one level in the call chain and the process repeats. The search continues until a matching handler is found or no callers remain. If a handler is never found the application terminates abnormally. If the search reaches the main procedure and it has a matching handler it will execute the handler, but, as always, the routine completes so once again the application terminates.

For a concrete example, consider the following:

### Listing 2: arrays.ads

```ada
package Arrays is
  type List is array (Natural range <>) of Integer;
  function Value (A : List; X, Y : Integer) return Integer;
end Arrays;
```

### Listing 3: arrays.adb

```ada
package body Arrays is
  function Value (A : List; X, Y : Integer) return Integer is
    begin
    return A (X + Y * 10);
  end Value;
```

(continues on next page)
Listing 4: some_process.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Arrays; use Arrays;

procedure Some_Process is
  L : constant List (1 .. 100) := (others => 42);
begin
  Put_Line (Integer'Image (Value (L, 1, 10)));
exception
  when Constraint_Error =>
    Put_Line ("Constraint_Error caught in Some_Process");
    Put_Line ("Some_Process completes normally");
end Some_Process;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Exceptions
MD5: 7733854601db37eb53f4c4094fe5ca0d

Listing 5: main.adb

```ada
with Some_Process;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
  Some_Process;
  Put_Line ("Main completes normally");
end Main;
```

Procedure Main calls Some_Process, which in turn calls function Value (line 7). Some_Process declares the array object L of type List on line 5, with bounds 1 through 100. The call to Value has arguments, including variable L, leading to an attempt to access an array component via an out-of-bounds index (1 + 10 * 10 = 101, beyond the last index of L). This attempt will trigger an exception in Value prior to actually accessing the array object's memory. Function Value doesn't have any exception handlers so the exception is propagated up to the caller Some_Process. Procedure Some_Process has an exception handler for Constraint_Error and it so happens that Constraint_Error is the exception raised in this case. As a result, the code for that handler will be executed, printing some messages on the screen. Then procedure Some_Process will return to Main normally. Main then continues to execute normally after the call to Some_Process and prints its completion message.

If procedure Some_Process had also not had a handler for Constraint_Error, that procedure call would also have returned abnormally and the exception would have been propagated further up the call chain to procedure Main. Normal execution in Main would likewise be abandoned in search of a handler. But Main does not have any handlers so Main would have completed abnormally, immediately, without printing its closing message.

This semantic model is the same as with many other programming languages, in which the execution of a frame's sequence of statements is unavoidably abandoned when an exception becomes active. The model is a direct reaction to the use of status codes returned from functions as in C, where it is all too easy to forget (intentionally or otherwise) to check the status values returned. With the exception model errors cannot be ignored.

70.1. Understanding Exceptions and Dynamic Checks
However, full exception propagation as described above is not the norm for embedded applications when the highest levels of integrity are required. The run-time library code implementing exception propagation can be rather complex and expensive to certify. Those problems apply to the application code too, because exception propagation is a form of control flow without any explicit construct in the source. Instead of the full exception model, designers of high-integrity applications often take alternative approaches.

One alternative consists of deactivating exceptions altogether, or more precisely, deactivating language-defined checks, which means that the compiler will not generate code checking for conditions giving rise to exceptions. Of course, this makes the code vulnerable to attacks, such as buffer overflow, unless otherwise verified (e.g. through static analysis). Deactivation can be applied at the unit level, through the -gnatp compiler switch, or locally within a unit via the pragma Suppress. (Refer to the GNAT User’s Guide for Native Platforms for more details about the switch.)

For example, we can write the following. Note the pragma on line 4 of arrays.adb within function Value:

---

**Listing 6: arrays.ads**

```ada
package Arrays is

    type List is array (Natural range <>) of Integer;

    function Value (A : List; X, Y : Integer) return Integer;

end Arrays;
```

**Listing 7: arrays.adb**

```ada
package body Arrays is

    function Value (A : List; X, Y : Integer) return Integer is
        pragma Suppress (All_Checks);
        begin
            return A (X + Y * 10);
        end Value;

end Arrays;
```

---

**Listing 8: some_process.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Arrays; use Arrays;

procedure Some_Process is
    L : constant List (1 .. 100) := (others => 42);
    begin
        Put_Line (Integer’Image (Value (L, 1, 10)));
        exception
            when Constraint_Error =>
                Put_Line ("FAILURE");
    end Some_Process;
```

This placement of the pragma will only suppress checks in the function body. However, [332](https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html)
that is where the exception would otherwise have been raised, leading to incorrect and unpredictable execution. (Run the program more than once. If it prints the right answer (42), or even the same value each time, it's just a coincidence.) As you can see, suppressing checks negates the guarantee of errors being detected and addressed at run-time.

Another alternative is to leave checks enabled but not retain the dynamic call-chain propagation. There are a couple of approaches available in this alternative.

The first approach is for the run-time library to invoke a global "last chance handler" (LCH) when any exception is raised. Instead of the sequence of statements of an ordinary exception handler, the LCH is actually a procedure intended to perform "last-wishes" before the program terminates. No exception handlers are allowed. In this scheme "propagation" is simply a direct call to the LCH procedure. The default LCH implementation provided by GNAT does nothing other than loop infinitely. Users may define their own replacement implementation.

The availability of this approach depends on the run-time library. Typically, Zero Footprint and Ravenscar SFP run-times will provide this mechanism because they are intended for certification.

A user-defined LCH handler can be provided either in C or in Ada, with the following profiles:

[Ada]

```ada
procedure Last_Chance_Handler (Source_Location : System.Address; Line : Integer);
pragma Export (C,
   Last_Chance_Handler,
   "__gnat_last_chance_handler");
```

[C]

```c
void __gnat_last_chance_handler (char *source_location,
   int line);
```

We'll go into the details of the pragma Export in a further section on language interfacing. For now, just know that the symbol __gnat_last_chance_handler is what the run-time uses to branch immediately to the last-chance handler. Pragma Export associates that symbol with this replacement procedure so it will be invoked instead of the default routine. As a consequence, the actual procedure name in Ada is immaterial.

Here is an example implementation that simply blinks an LED forever on the target:

```ada
procedure Last_Chance_Handler (Msg : System.Address; Line : Integer) is
   pragma Unreferenced (Msg, Line);

   Next_Release : Time := Clock;
   Period : constant Time_Span := Milliseconds (500);
begin
   Initialize_LEDs;
   All_LEDS_Off;
   loop
      Toggle (LCH_LED);
      Next_Release := Next_Release + Period;
      delay until Next_Release;
   end loop;
end Last_Chance_Handler;
```

The LCH_LED is a constant referencing the LED used by the last-chance handler, declared elsewhere. The infinite loop is necessary because a last-chance handler must never return to the caller (hence the term "last-chance"). The LED changes state every half-second.

Unlike the approach in which there is only the last-chance handler routine, the other approach allows exception handlers, but in a specific, restricted manner. Whenever an ex-
Learning Ada

exception is raised, the only handler that can apply is a matching handler located in the same frame in which the exception is raised. Propagation in this context is simply an immediate branch instruction issued by the compiler, going directly to the matching handler’s sequence of statements. If there is no matching local handler the last chance handler is invoked. For example consider the body of function Value in the body of package Arrays:

Listing 9: arrays.ads

```ada
package Arrays is

  type List is array (Natural range <>) of Integer;
  function Value (A : List; X, Y : Integer) return Integer;

end Arrays;
```

Listing 10: arrays.adb

```ada
package body Arrays is

  function Value (A : List; X, Y : Integer) return Integer is
  begin
    return A (X + Y * 10);
  exception
    when Constraint_Error =>
      return 0;
  end Value;

end Arrays;
```

Code block metadata

Project: Courses.Ada_For_EMBEDDED_C_DEV.SPARK.Exception_Return
MD5: 1f63b92739deb03529884ab0d25dad8

Listing 11: some_process.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Arrays; use Arrays;

procedure Some_Process is
  L : constant List (1 .. 100) := (others => 42);
begin
  Put_Line (Integer'Image (Value (L, 1, 10)));
  exception
  when Constraint_Error =>
    Put_Line ("FAILURE");
end Some_Process;
```

In both procedure Some_Process and function Value we have an exception handler for Constraint_Error. In this example the exception is raised in Value because the index check fails there. A local handler for that exception is present so the handler applies and the function returns zero, normally. Because the call to the function returns normally, the execution of Some_Process prints zero and then completes normally.

Let’s imagine, however, that function Value did not have a handler for Constraint_Error. In the context of full exception propagation, the function call would return to the caller, i.e., Some_Process, and would be handled in that procedure’s handler. But only local handlers are allowed under the second alternative so the lack of a local handler in Value would result in the last-chance handler being invoked. The handler for Constraint_Error in Some_Process under this alternative approach.
So far we've only illustrated handling the Constraint_Error exception. It's possible to handle other language-defined and user-defined exceptions as well, of course. It is even possible to define a single handler for all other exceptions that might be encountered in the handled sequence of statements, beyond those explicitly named. The "name" for this otherwise anonymous exception is the Ada reserved word others. As in case statements, it covers all other choices not explicitly mentioned, and so must come last. For example:

Listing 12: arrays.ads

```ada
package Arrays is
  type List is array (Natural range <>) of Integer;
  function Value (A : List; X, Y : Integer) return Integer;
end Arrays;
```

Listing 13: arrays.adb

```ada
package body Arrays is
  function Value (A : List; X, Y : Integer) return Integer is
  begin
    return A (X + Y * 10);
  when Constraint_Error =>
    return 0;
  when others =>
    return -1;
  end Value;
end Arrays;
```

In the code above, the Value function has a handler specifically for Constraint_Error as before, but also now has a handler for all other exceptions. For any exception other than Constraint_Error, function Value returns -1. If you remove the function's handler for Constraint_Error (lines 7 and 8) then the other "anonymous" handler will catch the exception and -1 will be returned instead of zero.

There are additional capabilities for exceptions, but for now you have a good basic understanding of how exceptions work, especially their dynamic nature at run-time.
70.2 Understanding Dynamic Checks versus Formal Proof

So far, we have discussed language-defined checks inserted by the compiler for verification at run-time, leading to exceptions being raised. We saw that these dynamic checks verified semantic conditions ensuring proper execution, such as preventing writing past the end of a buffer, or exceeding an application-specific integer range constraint, and so on. These checks are defined by the language because they apply generally and can be expressed in language-defined terms.

Developers can also define dynamic checks. These checks specify component-specific or application-specific conditions, expressed in terms defined by the component or application. We will refer to these checks as "user-defined" for convenience. (Be sure you understand that we are not talking about user-defined exceptions here.)

Like the language-defined checks, user-defined checks must be true at run-time. All checks consist of Boolean conditions, which is why we can refer to them as assertions: their conditions are asserted to be true by the compiler or developer.

Assertions come in several forms, some relatively low-level, such as a simple pragma Assert, and some high-level, such as type invariants and contracts. These forms will be presented in detail in a later section, but we will illustrate some of them here.

User-defined checks can be enabled at run-time in GNAT with the -gnata switch, as well as with pragma Assertion_Policy. The switch enables all forms of these assertions, whereas the pragma can be used to control specific forms. The switch is typically used but there are reasonable use-cases in which some user-defined checks are enabled, and others, although defined, are disabled.

By default in GNAT, language-defined checks are enabled but user-defined checks are disabled. Here's an example of a simple program employing a low-level assertion. We can use it to show the effects of the switches, including the defaults:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  X : Positive := 10;
begin
  X := X * 5;
  pragma Assert (X > 99);
  X := X - 99;
  Put_Line (Integer'Image (X));
end Main;
```

If we compiled this code we would get a warning about the assignment on line 8 after the pragma Assert, but not one about the Assert itself on line 7.

```
gprbuild -q -P main.gpr
main.adb:8:11: warning: value not in range of type "Standard Positive"
main.adb:8:11: warning: "Constraint_Error" will be raised at run time
```

No code is generated for the user-defined check expressed via pragma Assert but the language-defined check is emitted. In this case the range constraint on X excludes zero and negative numbers, but X * 5 = 50, X - 99 = -49. As a result, the check for the last
assignment would fail, raising **Constraint_Error** when the program runs. These results are the expected behavior for the default switch settings.

But now let's enable user-defined checks and build it. Different compiler output will appear.

Listing 16: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    X : Positive := 10;
    begin
        X := X * 5;
        pragma Assert (X > 99);
        X := X - 99;
        Put_Line (Integer'Image (X));
    end Main;
```

**Code block metadata**

- **Project**: Courses.Ada_For_Embedded_C_Dev.SPARK.Assert
- **MD5**: 2eb5e1879740cc3914acb8a362995b31

**Build output**

- main.adb:7:19: warning: assertion will fail at run time [-gnatw.a]
- main.adb:8:11: warning: value not in range of type "Standard.Positive" [enabled by default]
- main.adb:8:11: warning: Constraint_Error will be raised at run time [enabled by default]

**Runtime output**

-raised ADA ASSERTIONS ASSERTION_ERROR : main.adb:7

Now we also get the compiler warning about the pragma **Assert** condition. When run, the failure of pragma **Assert** on line 7 raises the exception **Ada.Assertions.Assertion_Error**. According to the expression in the assertion, **X** is expected (incorrectly) to be above 99 after the multiplication. (The exception name in the error message, **SYSTEM ASSERTIONS ASSERTION_FAILURE**, is a GNAT-specific alias for **Ada.Assertions.Assertion_Error**.)

It's interesting to see in the output that the compiler can detect some violations at compile-time:

- main.adb:7:19: warning: assertion will fail at run time
- main.adb:7:21: warning: condition can only be **True if invalid values present**
- main.adb:8:11: warning: value **not in range of type "Standard.Positive"**

Generally speaking, a complete analysis is beyond the scope of compilers and they may not find all errors prior to execution, even those we might detect ourselves by inspection. More errors can be found by tools dedicated to that purpose, known as static analyzers. But even an automated static analysis tool cannot guarantee it will find all potential problems.

A much more powerful alternative is formal proof, a form of static analysis that can (when possible) give strong guarantees about the checks, for all possible conditions and all possible inputs. Proof can be applied to both language-defined and user-defined checks.

Be sure you understand that formal proof, as a form of static analysis, verifies conditions prior to execution, even prior to compilation. That earliness provides significant cost benefits. Removing bugs earlier is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after
Learning Ada

deployment. Preventing bug introduction into the deployed system is the least expensive approach of all. Furthermore, cost savings during the initial development will be possible as well, for reasons specific to proof. We will revisit this topic later in this section.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the gnatprove verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof. As a disclaimer, this course is not aimed at providing a full introduction to proof and the SPARK language, but rather to present in a few examples what it is about and what it can do for us.

As it turns out, our procedure Main is already SPARK compliant so we can start verifying it.

Listing 17: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    X : Positive := 10;
begin
    X := X * 5;
    pragma Assert (X > 99);
    X := X - 99;
    Put_Line (Integer'Image (X));
end Main;
```

Code block metadata

Project: Courses.Ada For Embedded C Dev.SPARK.Assert
MD5: 98cad2c7e7b7a12740db013727f01d45

Build output

main.adb:7:20: warning: assertion will fail at run time [-gnatw.a]
main.adb:8:12: warning: value not in range of type "Standard.Positive" [enabled by _default]
main.adb:8:12: warning: Constraint_Error will be raised at run time [enabled by _default]

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
main.adb:7:20: medium: assertion might fail
gnatprove: unproved check messages considered as errors

Runtime output

raised ADA.ASSERTIONS ASSERTION_ERROR : main.adb:7

The "Prove" button invokes gnatprove on main.adb. You can ignore the parameters to the invocation. For the purpose of this demonstration, the interesting output is this message:

main.adb:7:19: medium: assertion might fail, cannot prove X > 99 (e.g. when X = 50)

gnatprove can tell that the assertion X > 99 may have a problem. There's indeed a bug here, and gnatprove even gives us the counterexample (when X is 50). As a result the code is not proven and we know we have an error to correct.

Notice that the message says the assertion "might fail" even though clearly gnatprove has an example for when failure is certain. That wording is a reflection of the fact that SPARK gives strong guarantees when the assertions are proven to hold, but does not guarantee that flagged problems are indeed problems. In other words, gnatprove does not give false
positives but false negatives are possible. The result is that if \texttt{gnatprove} does not indicate a problem for the code under analysis we can be sure there is no problem, but if \texttt{gnatprove} does indicate a problem the tool may be wrong.

### 70.3 Initialization and Correct Data Flow

An immediate benefit from having our code compatible with the SPARK subset is that we can ask \texttt{gnatprove} to verify initialization and correct data flow, as indicated by the absence of messages during SPARK “flow analysis.” Flow analysis detects programming errors such as reading uninitialized data, problematic aliasing between formal parameters, and data races between concurrent tasks.

In addition, \texttt{gnatprove} checks unit specifications for the actual data read or written, and the flow of information from inputs to outputs. As you can imagine, this verification provides significant benefits, and it can be reached with comparatively low cost.

For example, the following illustrates an initialization failure:

```
Listing 18: main.adb
1. with Increment;
2. with Ada.Text_IO; use Ada.Text_IO;

3. procedure Main is
4.     B : Integer;
5. begin
6.     Increment (B);
7.     Put_Line (B'Image);
8. end Main;
```

```
Listing 19: increment.adb
1. procedure Increment (Value : in out Integer) is
2. begin
3.     Value := Value + 1;
4. end Increment;
```

Consider this next routine, which contains a serious coding error. Flow analysis will find it for us.

```
with Increment;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    B : Integer;
begin
    Increment (B);
    Put_Line (B'Image);
end Main;
```

procedure Compute_Offset (K : Float; Z : out Integer; Flag : out Boolean) is
  X : constant Float := Sin (K);
begin
  if X < 0.0 then
    Z := 0;
    Flag := True;
  elsif X > 0.0 then
    Z := 1;
    Flag := True;
  else
    Flag := False;
  end if;
end Compute_Offset;

70.4 Contract-Based Programming

So far, we’ve seen assertions in a routine’s sequence of statements, either through implicit language-defined checks (is the index in the right range?) or explicit user-defined checks. These checks are already useful by themselves but they have an important limitation: the assertions are in the implementation, hidden from the callers of the routine. For example, a call’s success or failure may depend upon certain input values but the caller doesn’t have that information.

Generally speaking, Ada and SPARK put a lot of emphasis on strong, complete specifications for the sake of abstraction and analysis. Callers need not examine the implementations to determine whether the arguments passed to it are changed, for example. It is possible to go beyond that, however, to specify implementation constraints and functional requirements. We use contracts to do so.
At the language level, contracts are higher-level forms of assertions associated with specifications and declarations rather than sequences of statements. Like other assertions they can be activated or deactivated at run-time, and can be statically proven. We'll concentrate here on two kinds of contracts, both associated especially (but not exclusively) with procedures and functions:

- **Preconditions**, those Boolean conditions required to be true prior to a call of the corresponding subprogram
- **Postconditions**, those Boolean conditions required to be true after a call, as a result of the corresponding subprogram’s execution

In particular, preconditions specify the initial conditions, if any, required for the called routine to correctly execute. Postconditions, on the other hand, specify what the called routine's execution must have done, at least, on normal completion. Therefore, preconditions are obligations on callers (referred to as "clients") and postconditions are obligations on implementers. By the same token, preconditions are guarantees to the implementers, and postconditions are guarantees to clients.

Contract-based programming, then, is the specification and rigorous enforcement of these obligations and guarantees. Enforcement is rigorous because it is not manual, but tool-based: dynamically at run-time with exceptions, or, with SPARK, statically, prior to build.

Preconditions are specified via the "Pre" aspect. Postconditions are specified via the "Post" aspect. Usually subprograms have separate declarations and these aspects appear with those declarations, even though they are about the bodies. Placement on the declarations allows the obligations and guarantees to be visible to all parties. For example:

```ada
function Mid (X, Y : Integer) return Integer with
  Pre => X + Y /= 0,
  Post => Mid′Result > X;
```

```
with Mid;
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is
  A, B, C : Integer;
begin
  A := Mid (1, 2);
  B := Mid (1, -1);
  C := Mid (A, B);
  Put_Line (C′Image);
end Demo;
```

The precondition on line 2 specifies that, for any given call, the sum of the values passed to parameters X and Y must not be zero. (Perhaps we're dividing by \(X + Y\) in the body.) The declaration also provides a guarantee about the function call's result, via the postcondition on line 3: for any given call, the value returned will be greater than the value passed to X.

Consider a client calling this function:
Learning Ada

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_2
MD5: 3e0617d4b1c14b37a81377456bf73eb5

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
demo.adb:8:09: medium: precondition might fail
gnatprove: unproved check messages considered as errors

gnatprove indicates that the assignment to B (line 8) might fail because of the precondition, i.e., the sum of the inputs shouldn’t be 0, yet \(-1 + 1 = 0\). (We will address the other output message elsewhere.)

Let’s change the argument passed to Y in the second call (line 8). Instead of -1 we will pass -2:

Listing 23: demo.adb

```ada
with Mid;
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is
    A, B, C : Integer;
begin
    A := Mid (1, 2);
    B := Mid (1, -2);
    C := Mid (A, B);
    Put_Line (C'Image);
end Demo;
```

Listing 24: mid.ads

```ada
function Mid (X, Y : Integer) return Integer with
    Pre => X + Y /= 0,
    Post => Mid'Result > X;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_3
MD5: 496937d76e16ba524f98f5a94398e929

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
warning: no bodies have been analyzed by GNATprove
enable analysis of a non-generic body using SPARK_Mode

The second call will no longer be flagged for the precondition. In addition, gnatprove will know from the postcondition that A has to be greater than 1, as does B, because in both calls 1 was passed to X. Therefore, gnatprove can deduce that the precondition will hold for the third call \(C := \text{Mid} (A, B)\); because the sum of two numbers greater than 1 will never be zero.

Postconditions can also compare the state prior to a call with the state after a call, using the 'Old attribute. For example:
Listing 25: increment.ads

```ada
procedure Increment (Value : in out Integer) with
  Pre => Value < Integer'Last,
  Post => Value = Value'Old + 1;
```

Listing 26: increment.adb

```ada
procedure Increment (Value : in out Integer) is
begin
  Value := Value + 1;
end Increment;
```

Code block metadata

Project: Courses.Ada For Embedded C Dev.SPARK.Contracts_4
MD5: b879dcff91cb4fbce5501474b7f2e732

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

The postcondition specifies that, on return, the argument passed to the parameter Value will be one greater than it was immediately prior to the call (Value'Old).

70.5 Replacing Defensive Code

One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations. For example, the Push operation for a stack type would need to ensure that the given stack is not already full. The body of the routine would first check that, explicitly, and perhaps raise an exception or set a status code. With preconditions we can make the requirement explicit and gnatprove will verify that the requirement holds at all call sites.

This reduction has a number of advantages:

- The implementation is simpler, removing validation code that is often difficult to test, makes the code more complex and leads to behaviors that are difficult to define.
- The precondition documents the conditions under which it's correct to call the subprogram, moving from an implementer responsibility to mitigate invalid input to a user responsibility to fulfill the expected interface.
- Provides the means to verify that this interface is properly respected, through code review, dynamic checking at run-time, or formal static proof.

As an example, consider a procedure Read that returns a component value from an array. Both the Data and Index are objects visible to the procedure so they are not formal parameters.

Listing 27: p.ads

```ada
package P is
  type List is array (Integer range <>) of Character;
  Data : List (1 .. 100);
  Index : Integer := Data'First;
```

(continues on next page)
In addition to procedure Read we would also have a way to load the array components in the first place, but we can ignore that for the purpose of this discussion.

Procedure Read is responsible for reading an element of the array and then incrementing the index. What should it do in case of an invalid index? In this implementation there is defensive code that returns a value arbitrarily chosen. We could also redesign the code to return a status in this case, or — better — raise an exception.

An even more robust approach would be instead to ensure that this subprogram is only called when Index is within the indexing boundaries of Data. We can express that requirement with a precondition (line 9).

In addition to procedure Read we would also have a way to load the array components in the first place, but we can ignore that for the purpose of this discussion.

Procedure Read is responsible for reading an element of the array and then incrementing the index. What should it do in case of an invalid index? In this implementation there is defensive code that returns a value arbitrarily chosen. We could also redesign the code to return a status in this case, or — better — raise an exception.

An even more robust approach would be instead to ensure that this subprogram is only called when Index is within the indexing boundaries of Data. We can express that requirement with a precondition (line 9).
procedure Read (V : out Character) is
begin
  V := Data (Index);
  Index := Index + 1;
end Read;
end P;

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Defensive
MD5: 9646614c34d191be51b4522c972538aa

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

Now we don't need the defensive code in the procedure body. That's safe because SPARK will attempt to prove statically that the check will not fail at the point of each call.

Assuming that procedure Read is intended to be the only way to get values from the array, in a real application (where the principles of software engineering apply) we would take advantage of the compile-time visibility controls that packages offer. Specifically, we would move all the variables' declarations to the private part of the package, or even the package body, so that client code could not possibly access the array directly. Only procedure Read would remain visible to clients, thus remaining the only means of accessing the array. However, that change would entail others, and in this chapter we are only concerned with introducing the capabilities of SPARK. Therefore, we keep the examples as simple as possible.

## 70.6 Proving Absence of Run-Time Errors

Earlier we said that gnatprove will verify both language-defined and user-defined checks. Proving that the language-defined checks will not raise exceptions at run-time is known as proving "Absence of Run-Time Errors" or AoRTE for short. Successful proof of these checks is highly significant in itself.

One of the major resulting benefits is that we can deploy the final executable with checks disabled. That has obvious performance benefits, but it is also a safety issue. If we disable the checks we also disable the run-time library support for them, but in that case the language does not define what happens if indeed an exception is raised. Formally speaking, anything could happen. We must have good reason for thinking that exceptions cannot be raised.

This is such an important issue that proof of AoRTE can be used to comply with the objectives of certification standards in various high-integrity domains (for example, DO-178B/C in avionics, EN 50128 in railway, IEC 61508 in many safety-related industries, ECSS-Q-ST-80C in space, IEC 60880 in nuclear, IEC 62304 in medical, and ISO 26262 in automotive).

As a result, the quality of the program can be guaranteed to achieve higher levels of integrity than would be possible in other programming languages.

However, successful proof of AoRTE may require additional assertions, especially preconditions. We can see that with procedure Increment, the procedure that takes an Integer argument and increments it by one. But of course, if the incoming value of the argument is the largest possible positive value, the attempt to increment it would overflow, raising Constraint_Error. (As you have likely already concluded, Constraint_Error is the most...
common exception you will have to deal with.) We added a precondition to allow only the integer values up to, but not including, the largest positive value:

Listing 31: increment.ads

```ada
procedure Increment (Value : in out Integer) with
  Pre => Value < Integer'Last,
  Post => Value = Value'Old + 1;
```

Listing 32: increment.adb

```ada
procedure Increment (Value : in out Integer) is
begin
  Value := Value + 1;
end Increment;
```

70.7 Proving Abstract Properties

The postcondition on Increment expresses what is, in fact, a unit-level requirement. Successfully proving such requirements is another significant robustness and cost benefit. Together with the proofs for initialization and AoRTE, these proofs ensure program integrity, that is, the program executes within safe boundaries: the control flow of the program is correctly programmed and cannot be circumvented through run-time errors, and data cannot be corrupted.

We can go even further. We can use contracts to express arbitrary abstract properties when such exist. Safety and security properties, for instance, could be expressed as postconditions and then proven by gnatprove.

For example, imagine we have a procedure to move a train to a new position on the track, and we want to do so safely, without leading to a collision with another train. Procedure Move, therefore, takes two inputs: a train identifier specifying which train to move, and the intended new position. The procedure's output is a value indicating a motion command to be given to the train in order to go to that new position. If the train cannot go to that new position safely the output command is to stop the train. Otherwise the command is for the train to continue at an indicated speed:

```ada
type Move_Result is (Full_Speed, Slow_Down, Keep_Going, Stop);

procedure Move
  (Train : in Train_Id;
   New_Position : in Train_Position;
   Result : out Move_Result)
```

(continues on next page)
The preconditions specify that, given a safe initial state and a valid move, the result of the call will also be a safe state: there will be at most one train per track section and the track signaling system will not allow any unsafe movements.

### 70.8 Final Comments

Make sure you understand that `gnatprove` does not attempt to prove the program correct as a whole. It attempts to prove language-defined and user-defined assertions about parts of the program, especially individual routines and calls to those routines. Furthermore, `gnatprove` proves the routines correct only to the extent that the user-defined assertions correctly and sufficiently describe and constrain the implementation of the corresponding routines.

Although we are not proving whole program correctness, as you will have seen — and done — we can prove properties than make our software far more robust and bug-free than is possible otherwise. But in addition, consider what proving the unit-level requirements for your procedures and functions would do for the cost of unit testing and system integration. The tests would pass the first time.

However, within the scope of what SPARK can do, not everything can be proven. In some cases that is because the software behavior is not amenable to expression as boolean conditions (for example, a mouse driver). In other cases the source code is beyond the capabilities of the analyzers that actually do the mathematical proof. In these cases the combination of proof and actual test is appropriate, and still less expensive that testing alone.

There is, of course, much more to be said about what can be done with SPARK and `gnatprove`. Those topics are reserved for the *Introduction to SPARK* (page 973) course.
71.1 Naming conventions and casing considerations

One question that may arise relatively soon when converting from C to Ada is the style of source code presentation. The Ada language doesn't impose any particular style and for many reasons, it may seem attractive to keep a C-like style — for example, camel casing — to the Ada program.

However, the code in the Ada language standard, most third-party code, and the libraries provided by GNAT follow a specific style for identifiers and reserved words. Using a different style for the rest of the program leads to inconsistencies, thereby decreasing readability and confusing automatic style checkers. For those reasons, it's usually advisable to adopt the Ada style — in which each identifier starts with an upper case letter, followed by lower case letters (or digits), with an underscore separating two "distinct" words within the identifier. Acronyms within identifiers are in upper case. For example, there is a language-defined package named Ada.Text_IO. Reserved words are all lower case.

Following this scheme doesn't preclude adding additional, project-specific rules.

71.2 Manually interfacing C and Ada

Before even considering translating code from C to Ada, it's worthwhile to evaluate the possibility of keeping a portion of the C code intact, and only translating selected modules to Ada. This is a necessary evil when introducing Ada to an existing large C codebase, where re-writing the entire code upfront is not practical nor cost-effective.

Fortunately, Ada has a dedicated set of features for interfacing with other languages. The Interfaces package hierarchy and the pragmas Convention, Import, and Export allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

Listing 1: call.c

```c
#include <stdio.h>

struct my struct {
    int A, B;
};

void call (struct my struct *p) {
    printf ("%d", p->A);
}
```
To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C struct `my_struct` is represented, we can use the following record along with a `pragma Convention`. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

[Ada]

Listing 2: use_my_struct.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Interfaces.C;

procedure Use_My_Struct is
  type my_struct is record
    A : Interfaces.C.int;
    B : Interfaces.C.int;
  end record;
  pragma Convention (C, my_struct);

  V : my_struct := (A => 1, B => 2);

begin
  Put_Line ('V = ('
    & Interfaces.C.int' Image (V.A)
    & Interfaces.C.int' Image (V.B)
    & ')');

end Use_My_Struct;
```

Describing a foreign subprogram call to Ada code is called binding and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a void function maps to an Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a `pragma Import`:

```ada
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

The Import pragma specifies that whenever Call is invoked by Ada code, it should invoke the Call function with the C calling convention.

And that's all that's necessary. Here's an example of a call to Call:

[Ada]

Listing 3: use_my_struct.adb

```ada
with Interfaces.C;

(continues on next page)```
procedure Use_My_Struct is
  type my_struct is record
    A : Interfaces.C.int;
    B : Interfaces.C.int;
  end record;
  pragma Convention (C, my_struct);

procedure Call (V : my_struct);
  pragma Import (C, Call, "call"); -- Third argument optional

V : my_struct := (A => 1, B => 2);
begin
  Call (V);
end Use_My_Struct;

Code block metadata
Project: Courses.Ada_For_EMBEDDED_C_DEV.Translation.My_Struct
MD5: 9b54edadd406c7f5a2b9f8b8f82a4a88

71.3 Building and Debugging mixed language code

The easiest way to build an application using mixed C / Ada code is to create a simple project file for gprbuild and specify C as an additional language. By default, when using gprbuild we only compile Ada source files. To compile C code files as well, we use the Languages attribute and specify c as an option, as in the following example of a project file named default.gpr:

project Default is
  for Languages use ("ada", "c");
  for Main use ("main.adb");
end Default;

Then, we use this project file to build the application by simply calling gprbuild. Alternatively, we can specify the project file on the command-line with the -P option — for example, gprbuild -P default.gpr. In both cases, gprbuild compiles all C source-code file found in the directory and links the corresponding object files to build the executable.

In order to include debug information, you can use gprbuild -cargs -g. This option adds debug information based on both C and Ada code to the executable. Alternatively, you can specify a Builder package in the project file and include global compilation switches for each language using the GlobalCompilationSwitches attribute. For example:

project Default is
  for Languages use ("ada", "c");
  for Main use ("main.adb");

  package Builder is
    for GlobalCompilationSwitches ("Ada") use ("-g");
    for GlobalCompilationSwitches ("C") use ("-g");
  end Builder;

end Default;
In this case, you can simply run `gprbuild -P default.gpr` to build the executable.

To debug the executable, you can use programs such as `gdb` or `ddd`, which are suitable for debugging both C and Ada source-code. If you prefer a complete IDE, you may want to look into GNAT Studio, which supports building and debugging an application within a single environment, and remotely running applications loaded to various embedded devices. You can find more information about `gprbuild` and GNAT Studio in the Introduction to GNAT Toolchain (page 1681) course.

### 71.4 Automatic interfacing

It may be useful to start interfacing Ada and C by using automatic binding generators. These can be done either by invoking `gcc -fdump-ada-spec` option (to generate an Ada binding to a C header file) or `-gnatceg` option (to generate a C binding to an Ada specification file). For example:

```
gcc -c -fdump-ada-spec my_header.h
gcc -c -gnatceg spec.ads
```

The level of interfacing is very low level and typically requires either massaging (changing the generated files) or wrapping (calling the generated files from a higher level interface). For example, numbers bound from C to Ada are only standard numbers where user-defined types may be desirable. C uses a lot of by-pointer parameters which may be better replaced by other parameter modes, etc.

However, the automatic binding generator helps having a starting point which ensures compatibility of the Ada and the C code.

### 71.5 Using Arrays in C interfaces

It is relatively straightforward to pass an array from Ada to C. In particular, with the GNAT compiler, passing an array is equivalent to passing a pointer to its first element. Of course, as there's no notion of boundaries in C, the length of the array needs to be passed explicitly. For example:

[C]

Listing 4: p.h

```c
void p (int * a, int length);
```

[Ada]

Listing 5: main.adb

```ada
procedure Main is
  type Arr is array (Integer range <>) of Integer;
  procedure P (V : Arr; Length : Integer);
pragma Import (C, P);
```

(continues on next page)
Learning Ada

(continued from previous page)

X : Arr (5 .. 15);
begin
  P (X, X'Length);
end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_1
MD5: 9bfbc0f31da4554a1e1de1a1ba2b1d305

The other way around — that is, retrieving an array that has been creating on the C side —
is more difficult. Because C doesn't explicitly carry boundaries, they need to be recreated
in some way.

The first option is to actually create an Ada array without boundaries. This is the most
flexible, but also the least safe option. It involves creating an array with indices over the
full range of Integer without ever creating it from Ada, but instead retrieving it as an access
from C. For example:

[C]

Listing 6: f.h

int * f ();

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_2
MD5: 19e33efb6d7d46778b88eaa270911ee5

[Ada]

Listing 7: main.adb

procedure Main is
  type Arr is array (Integer) of Integer;
  type Arr_A is access all Arr;
  function F return Arr_A;
  pragma Import (C, F);
begin
  null;
end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_2
MD5: b52213bcd8db5e8abfcb8e448d84df

Note that Arr is a constrained type (it doesn't have the range <> notation for indices). For
that reason, as it would be for C, it's possible to iterate over the whole range of integer,
beyond the memory actually allocated for the array.

A somewhat safer way is to overlay an Ada array over the C one. This requires having
access to the length of the array. This time, let's consider two cases, one with an array
and its size accessible through functions, another one on global variables. This time, as
we're using an overlay, the function will be directly mapped to an Ada function returning
an address:

[C]
Listing 8: fg.h

```c
int * f_arr (void);
int f_size (void);
int * g_arr;
int g_size;
```

Listing 9: fg.ads

```ada
with System;
package Fg is
  type Arr is array (Integer range <>) of Integer;
  function F_Arr return System.Address;
  pragma Import (C, F_Arr, "f_arr");
  function F_Size return Integer;
  pragma Import (C, F_Size, "f_size");
  F : Arr (0 .. F_Size - 1) with Address => F_Arr;
  G_Size : Integer;
  pragma Import (C, G_Size, "g_size");
  G_Arr : Arr (0 .. G_Size - 1);
  pragma Import (C, G_Arr, "g_arr");
end Fg;
```
With all solutions though, importing an array from C is a relatively unsafe pattern, as there's only so much information on the array as there would be on the C side in the first place. These are good places for careful peer reviews.

### 71.6 By-value vs. by-reference types

When interfacing Ada and C, the rules of parameter passing are a bit different with regards to what's a reference and what's a copy. Scalar types and pointers are passed by value, whereas record and arrays are (almost) always passed by reference. However, there may be cases where the C interface also passes values and not pointers to objects. Here's a slightly modified version of a previous example to illustrate this point:

[C]

#### Listing 11: call.c

```c
#include <stdio.h>

struct my_struct {
    int A, B;
};

void call (struct my_struct p) {
    printf("%d", p.A);
}
```

#### Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Param_By_Value_C
MD5: 42b6e329c5dbfcae368078ca7635341f

In Ada, a type can be modified so that parameters of this type can always be passed by copy.

[Ada]

#### Listing 12: main.adb

```ada
with Interfaces.C;

procedure Main is
    type my_struct is record
        A : Interfaces.C.int;
        B : Interfaces.C.int;
    end record
```

(continues on next page)
with Convention => C_Pass_By_Copy;

procedure Call (V : my_struct);
pragma Import (C, Call, "call");
begin
  null;
end Main;

71.7 Naming and prefixes

Because of the absence of namespaces, any global name in C tends to be very long. And because of the absence of overloading, they can even encode type names in their type.

In Ada, the package is a namespace — two entities declared in two different packages are clearly identified and can always be specifically designated. The C names are usually a good indication of the names of the future packages and should be stripped — it is possible to use the full name if useful. For example, here’s how the following declaration and call could be translated:

[C]

Listing 13: reg_interface.h

void registerInterface_Initilize (int size);

Listing 14: reg_interface_test.c

#include "reg_interface.h"

int main(int argc, const char * argv[])
{
  registerInterface_Initilize(15);
  return 0;
}

[Ada]

Listing 15: register_interface.ads

package Register_Interface is
  procedure Initialize (Size : Integer)
    with Import => True,
    Convention => C,
    External_Name => "registerInterface_Initilize";
  end Initialize;
end Register_Interface;
71.8 Pointers

The first thing to ask when translating pointers from C to Ada is: are they needed in the first place? In Ada, pointers (or access types) should only be used with complex structures that cannot be allocated at run-time — think of a linked list or a graph for example. There are many other situations that would need a pointer in C, but do not in Ada, in particular:

- Arrays, even when dynamically allocated
- Results of functions
- Passing large structures as parameters
- Access to registers
- ... others

This is not to say that pointers aren't used in these cases but, more often than not, the pointer is hidden from the user and automatically handled by the code generated by the compiler; thus avoiding possible mistakes from being made. Generally speaking, when looking at C code, it's good practice to start by analyzing how many pointers are used and to translate as many as possible into pointerless Ada structures.

Here are a few examples of such patterns — additional examples can be found throughout this document.

Dynamically allocated arrays can be directly allocated on the stack:

[C]

```
#include <stdlib.h>

int main() {
    int *a = malloc(sizeof(int) * 10);
    return 0;
}
```

Note that in the above example, a `use` clause on Register_Interface could allow us to omit the prefix.
Learning Ada

Listing 18: main.adb

```ada
procedure Main is
  type Arr is array (Integer range <>) of Integer;
  A : Arr (0 .. 9);
begin
  null;
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Array_Stack_Alloc_Ada
MD5: 2e4196c2a2016244a48de153beaa2b49

Build output

main.adb:3:04: warning: variable "A" is never read and never assigned [-gnatwv]

It's even possible to create a such an array within a structure, provided that the size of the array is known when instantiating this object, using a type discriminant:

[C]

Listing 19: array_decl.c

```c
#include <stdlib.h>

typedef struct {
  int * a;
} S;

int main(int argc, const char * argv[])
{
  S v;
  v.a = malloc(sizeof(int) * 10);
  return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Struct_Array_Stack_Alloc_C
MD5: f8e5a877977387986b3e2353834a2989

[Ada]

Listing 20: main.adb

```ada
procedure Main is
  type Arr is array (Integer range <>) of Integer;
  type S (Last : Integer) is record
    A : Arr (0 .. Last);
  end record;
  V : S (9);
```

(continues on next page)
begin
  null;
end Main;

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Struct_Array_Stack_Alloc_Ada
MD5: 955c704bde4b2b788e4a790ade12df7

**Build output**

main.adb:8:04: warning: variable "V" is never read and never assigned [-gnatwv]

With regards to parameter passing, usage mode (input / output) should be preferred to implementation mode (by copy or by reference). The Ada compiler will automatically pass a reference when needed. This works also for smaller objects, so that the compiler will copy in an out when needed. One of the advantages of this approach is that it clarifies the nature of the object: in particular, it differentiates between arrays and scalars. For example:

[C]

Listing 21: p.h

```c
void p (int * a, int * b);
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Array_In_Out_C
MD5: c2c936dd3afc4850c5869e4db73bb36b

[A]

Listing 22: array_types.ads

```ada
package Array_Types is
  type Arr is array (Integer range <>) of Integer;
  procedure P (A : in out Integer; B : in out Arr);
end Array_Types;
```

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Array_In_Out_Ada
MD5: cf8e51391c9fd8608183c9dae2aa2802

Most of the time, access to registers end up in some specific structures being mapped onto a specific location in memory. In Ada, this can be achieved through an Address clause associated to a variable, for example:

[C]

Listing 23: test_c.c

```c
int main(int argc, const char * argv[])
{
  int * r = (int *)0xFFFF00A0;
  return 0;
}
```

**Code block metadata**

71.8. Pointers
[Ada]

Listing 24: test.adb

```ada
with System;

procedure Test is
  R : Integer with Address => System'To_Address (16#FFFF00A0#);

begin
  null;
end Test;
```

71.9 Bitwise Operations

Bitwise operations such as masks and shifts in Ada should be relatively rarely needed, and, when translating C code, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other.

Consider the case of using a C primitive type as a container for single bit boolean flags. In C, this would be done through masks, e.g.:

[C]

Listing 25: flags.c

```c
#define FLAG_1 0b0001
#define FLAG_2 0b0010
#define FLAG_3 0b0100
#define FLAG_4 0b1000

int main(int argc, const char * argv[]) {
  int value = 0;
  value |= FLAG_2 | FLAG_4;
  return 0;
}
```

In Ada, the above can be represented through a Boolean array of enumerate values:

[Ada]
Listing 26: main.adb

```
procedure Main is
  type Values is (Flag_1, Flag_2, Flag_3, Flag_4);
  type Value_Array is array (Values) of Boolean
      with Pack;
  Value : Value_Array :=
      (Flag_2 => True,
       Flag_4 => True,
       others => False);
begin
  null;
end Main;
```

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Translation.Flags_Ada
MD5: c92c8532763469f5e4d1027df2bd6a6b

Note the Pack directive for the array, which requests that the array takes as little space as possible.

It is also possible to map records on memory when additional control over the representation is needed or more complex data are used:

[C]

Listing 27: struct_map.c

```
int main(int argc, const char * argv[]) {
    int value = 0;
    value = (2 << 1) | 1;
    return 0;
}
```

Code block metadata
Project: Courses.Ada_For_Embedded_C_Dev.Translation.Rec_Map_C
MD5: 16606f11ab3e9c86d3e1d88ac9c3f37f

[Ada]

Listing 28: main.adb

```
procedure Main is
  type Value_Rec is record
    V1 : Boolean;
    V2 : Integer range 0 .. 3;
  end record;
  for Value_Rec use record
    V1 at 0 range 0 .. 0;
    V2 at 0 range 1 .. 2;
  end record;
  Value : Value_Rec := (V1 => True, V2 => 2);
begin
```

(continues on next page)
The benefit of using Ada structure instead of bitwise operations is threefold:

- The code is simpler to read / write and less error-prone
- Individual fields are named
- The compiler can run consistency checks (for example, check that the value indeed fit in the expected size).

Note that, in cases where bitwise operators are needed, Ada provides modular types with `and`, `or` and `xor` operators. Further shift operators can also be provided upon request through a `pragma`. So the above could also be literally translated to:

[Ada]

Listing 29: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Value_Type is mod 2 ** 32;
  pragma Provide_Shift_Operators (Value_Type);
  Value : Value_Type;
begin
  Value := Shift_Left (2, 1) or 1;
  Put_Line ("Value = " & Value_Type'image (Value));
end Main;
```

71.10 Mapping Structures to Bit-Fields

In the previous section, we've seen how to perform bitwise operations. In this section, we look at how to interpret a data type as a bit-field and perform low-level operations on it.

In general, you can create a bit-field from any arbitrary data type. First, we declare a bit-field type like this:

[Ada]

```
type Bit_Field is array (Natural range <>) of Boolean with Pack;
```

As we've seen previously, the Pack aspect declared at the end of the type declaration indicates that the compiler should optimize for size. We must use this aspect to be able to interpret data types as a bit-field.
Then, we can use the **Size** and the **Address** attributes of an object of any type to declare a bit-field for this object. We've discussed the **Size** attribute *earlier in this course* (page 1443).

The **Address** attribute indicates the address in memory of that object. For example, assuming we've declare a variable V, we can declare an actual bit-field object by referring to the **Address** attribute of V and using it in the declaration of the bit-field, as shown here:

[Ada]

\[
B : \text{Bit\_Field (0 .. V'\text{Size} - 1) with Address => V'Address;}
\]

Note that, in this declaration, we're using the **Address** attribute of V for the **Address** aspect of B.

This technique is called overlays for serialization. Now, any operation that we perform on B will have a direct impact on V, since both are using the same memory location.

The approach that we use in this section relies on the **Address** aspect. Another approach would be to use unchecked conversions, which we'll discuss in the *next section* (page 1502).

We should add the **Volatile** aspect to the declaration to cover the case when both objects can still be changed independently — they need to be volatile, otherwise one change might be missed. This is the updated declaration:

[Ada]

\[
B : \text{Bit\_Field (0 .. V'\text{Size} - 1) with Address => V'Address, Volatile;}
\]

Using the **Volatile** aspect is important at high level of optimizations. You can find further details about this aspect in the section about the **Volatile and Atomic aspects** (page 1439).

Another important aspect that should be added is **Import**. When used in the context of object declarations, it'll avoid default initialization which could overwrite the existing content while creating the overlay — see an example in the admonition below. The declaration now becomes:

[Ada]

\[
B : \text{Bit\_Field (0 .. V'\text{Size} - 1) with Address => V'Address, Import, Volatile;}
\]

Let's look at a simple example:

[Ada]

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Simple_Bitfield is
  type Bit_Field is array (Natural range <>) of Boolean with Pack;
  V : Integer := 0;
  B : Bit_Field (0 .. V'\text{Size} - 1)
    with Address => V'Address, Import, Volatile;
begin
  B (2) := True;
  Put_Line ("V = " & Integer'\text{Image} (V));
end Simple_Bitfield;
```

**Code block metadata**

Project: Courses.Ada_For_EMBEDDED_C_DEV.Translation.Bitfield_Ada  
MD5: 193a2db91619426a145cd267f873145f

71.10. Mapping Structures to Bit-Fields 1489
Learning Ada

Runtime output

\[
V = 4
\]

In this example, we first initialize \( V \) with zero. Then, we use the bit-field \( B \) and set the third element (\( B(2) \)) to \texttt{True}. This automatically sets bit #3 of \( V \) to 1. Therefore, as expected, the application displays the message \( V = 4 \), which corresponds to \( 2^2 = 4 \).

Note that, in the declaration of the bit-field type above, we could also have used a positive range. For example:

\[
\text{type Bit_Field is array (Positive range <>)} \text{ of Boolean with Pack;}
\]

\[
B : \text{Bit_Field (1 .. V’Size)}
\]

\[
\text{with Address} => V’Address,Import,Volatile;
\]

The only difference in this case is that the first bit is \( B(1) \) instead of \( B(0) \).

In C, we would rely on bit-shifting and masking to set that specific bit:

[C]

Listing 31: bitfield.c

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int v = 0;
    v = v | (1 << 2);
    printf("v = %d\n", v);
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_C
MD5: 98557f80ea3bc1b081ae2688f844cbe1

Runtime output

\[ v = 4 \]

Important

Ada has the concept of default initialization. For example, you may set the default value of record components:

[Ada]

Listing 32: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    type Rec is record
        X : Integer := 10;
        Y : Integer := 11;
    end record;
```
In the code above, we don't explicitly initialize the components of R, so they still have the default values 10 and 11, which are displayed by the application.

Likewise, the Default_Value aspect can be used to specify the default value in other kinds of type declarations. For example:

[Ada]

Listing 33: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

  type Percentage is range 0 .. 100
  with Default_Value => 10;

  P : Percentage;

begin
  Put_Line ("P = " & Percentage'Image (P));
end Main;
```

When declaring an object whose type has a default value, the object will automatically be initialized with the default value. In the example above, P is automatically initialized with 10, which is the default value of the Percentage type.

Some types have an implicit default value. For example, access types have a default value of null.

As we've just seen, when declaring objects for types with associated default values, automatic initialization will happen. This can also happens when creating an overlay with the Address aspect. The default value is then used to overwrite the content at the memory location indicated by the address. However, in most situations, this isn't the behavior we expect, since overlays are usually created to analyze and manipulate existing values. Let's look at an example where this happens:

[Ada]
Learning Ada

Listing 34: p.ads

package P is

  type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;

  type Byte_Field is array (Natural range <>) of Unsigned_8;

  procedure Display.Bytes_Increment (V : in out Integer);

end P;

Listing 35: p.adb

with Ada.Text_IO; use Ada.Text_IO;

package body P is

  procedure Display.Bytes_Increment (V : in out Integer)
  is
    BF : Byte_Field (1 .. V'Size / 8)
      with Address => V'Address, Volatile;
  begin
    for B of BF loop
      Put_Line ("Byte = " & Unsigned_8'Image (B));
    end loop;
    Put_Line ("Now incrementing...");
    V := V + 1;
  end Display.Bytes_Increment;

end P;

Listing 36: main.adb

with Ada.Text_IO; use Ada.Text_IO;

with P; use P;

procedure Main is
  V : Integer := 10;
begin
  Put_Line ("V = " & Integer'Image (V));
  Display.Bytes_Increment (V);
  Put_Line ("V = " & Integer'Image (V));
end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Overlay_Default_Init_Overwrite
MD5: 04994b2b4c98e9232a155515dc0c365a

Build output

p.adb:7:14: warning: default initialization of "Bf" may modify "V" [enabled by default]
p.adb:7:14: warning: use pragma Import for "Bf" to suppress initialization (RM B. 1(24)) [enabled by default]

Runtime output

V = 10
Byte = 0
Byte = 0

(continues on next page)
Byte = 0
Byte = 0
Now incrementing...
V = 1

In this example, we expect Display_Bytes_Increment to display each byte of the V parameter and then increment it by one. Initially, V is set to 10, and the call to Display_Bytes_Increment should change it to 11. However, due to the default value associated to the Unsigned_8 type — which is set to 0 — the value of V is overwritten in the declaration of BF (in Display_Bytes_Increment). Therefore, the value of V is 1 after the call to Display_Bytes_Increment. Of course, this is not the behavior that we originally intended.

Using the Import aspect solves this problem. This aspect tells the compiler to not apply default initialization in the declaration because the object is imported. Let’s look at the corrected example:

[Ada]

Listing 37: p.ads

```ada
package P is

  type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;
  type Byte_Field is array (Natural range <>) of Unsigned_8;
  procedure Display_Bytes_Increment (V : in out Integer);

end P;
```

Listing 38: p.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body P is

  procedure Display_Bytes_Increment (V : in out Integer) is
    BF : Byte_Field (1 .. V'Size / 8)
      with Address => V'Address, Import, Volatile;
  begin
    for B of BF loop
      Put_Line ("Byte = " & Unsigned_8'image (B));
    end loop;
    Put_Line ("Now incrementing...");
    V := V + 1;
  end Display_Bytes_Increment;

end P;
```

Listing 39: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with P; use P;

procedure Main is
  V : Integer := 10;
begin
  Put_Line ("V = " & Integer'image (V));
  Display_Bytes_Increment (V);
```

(continues on next page)
Put_Line ("V = " & Integer'Mimage (V));
end Main;

This unwanted side-effect of the initialization by the Default_Value aspect that we've just seen can also happen in these cases:

- when we set a default value for components of a record type declaration,
- when we use the Default_Component_Value aspect for array types, or
- when we set use the Initialize_Scalars pragma for a package.

Again, using the Import aspect when declaring the overlay eliminates this side-effect.

We can use this pattern for objects of more complex data types like arrays or records. For example:

Listing 40: int_array_bitfield.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Int_Array_Bitfield is
  type Bit_Field is array (Natural range <>) of Boolean with Pack;

  A : array (1 .. 2) of Integer := (others => 0);
  B : Bit_Field (0 .. A'Size - 1)
      with Address => A'Address, Import, Volatile;

begin
  B (2) := True;
  for I in A'Range loop
    Put_Line ("A (" & Integer'Mimage (I)
                & ")= " & Integer'Mimage (A (I)))
  end loop;
end Int_Array_Bitfield;
```

```
In the Ada example above, we're using the bit-field to set bit #3 of the first element of the array (A (1)). We could set bit #4 of the second element by using the size of the data type (in this case, `Integer` 'Size'):

[Ada]

```
B (Integer'Size + 3) := True;
```

In C, we would select the specific array position and, again, rely on bit-shifting and masking to set that specific bit:

[C]

```
Listing 41: bitfield_int_array.c
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int i;
    int a[2] = {0, 0};
    a[0] = a[0] | (1 << 2);
    for (i = 0; i < 2; i++)
    {
        printf("a[%d] = %d\n", i, a[i]);
    }
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Int_Array_C
MD5: 4dc3fe77e8260ff3b449c8779745a63c

Runtime output

```
a[0] = 4
a[1] = 0
```

Since we can use this pattern for any arbitrary data type, this allows us to easily create a subprogram to serialize data types and, for example, transmit complex data structures as a bitstream. For example:

[Ada]

```
Listing 42: serializer.ads
package Serializer is
    type Bit_Field is array (Natural range <>) of Boolean with Pack;
    procedure Transmit (B : Bit_Field);
end Serializer;
```

```
Listing 43: serializer.adb
with Ada.Text_IO; use Ada.Text_IO;
package body Serializer is
```
(continues on next page)
procedure 

procedure 

begin
  when False => Put ("0");
  when True  => Put ("1");
end case;
end Show_Bit;

begin
  Put ("Bits: ");
  for E of B loop
    Show_Bit (E);
  end loop;
  New_Line;
end Transmit;
end Serializer;

package My_Recs is
  type Rec is record
    V : Integer;
    S : String (1 .. 3);
  end record;
end My_Recs;

with Serializer; use Serializer;
with My_Recs; use My_Recs;

procedure Main is
  R : Rec := (5, "abc");
  B : Bit_Field (0 .. R'Size - 1)
    with Address => R'Address, Import, Volatile;
begin
  Transmit (B);
end Main;

Code block metadata
Project: Courses.Ada_For_Embedded_C_DEV_Translation.Bitfield_Serialization_ada
MD5: 5c9c2d18bab7c78456d1d795c6334cd9

Build output
main.adb:9:14: warning: volatile actual passed by copy (RM C.6(19)) [enabled by _default]

Runtime output
Bits: 1010000000000000000000000000000010000110010001101100011000000000

In this example, the Transmit procedure from Serializer package displays the individual bits of a bit-field. We could have used this strategy to actually transmit the information as
a bitstream. In the main application, we call Transmit for the object \( R \) of record type Rec. Since Transmit has the bit-field type as a parameter, we can use it for any type, as long as we have a corresponding bit-field representation.

In C, we interpret the input pointer as an array of bytes, and then use shifting and masking to access the bits of that byte. Here, we use the \texttt{char} type because it has a size of one byte in most platforms.

[C]

Listing 46: my_recs.h

```c
typedef struct {
    int v;
    char s[4];
} rec;
```

Listing 47: serializer.h

```c
void transmit (void *bits, int len);
```

Listing 48: serializer.c

```c
#include "serializer.h"
#include <stdio.h>
#include <assert.h>

void transmit (void *bits, int len)
{
    int i, j;
    char *c = (char *)bits;
    assert(sizeof(char) == 1);
    printf("Bits: ");
    for (i = 0; i < len / (sizeof(char) * 8); i++)
    {
        for (j = 0; j < sizeof(char) * 8; j++)
        {
            printf("%d", c[i] >> j & 1);
        }
    }
    printf("\n");
} 
```
#include <stdio.h>

#include "my_recs.h"
#include "serializer.h"

int main(int argc, const char * argv[]) {
    rec r = {5, "abc"};
    transmit(&r, sizeof(r) * 8);
    return 0;
}

Similarly, we can write a subprogram that converts a bit-field — which may have been received as a bitstream — to a specific type. We can add a ToRec subprogram to the My_Recs package to convert a bit-field to the Rec type. This can be used to convert a bitstream that we received into the actual data type representation.

As you know, we may write the ToRec subprogram as a procedure or as a function. Since we need to use slightly different strategies for the implementation, the following example has both versions of ToRec.

This is the updated code for the My_Recs package and the Main procedure:

[Ada]

package Serializer is

    type Bit_Field is array (Natural range <>) of Boolean with Pack;

    procedure Transmit (B : Bit_Field);

end Serializer;

with Ada.Text_IO; use Ada.Text_IO;

package body Serializer is

    procedure Transmit (B : Bit_Field) is

        procedure Show_Bit (V : Boolean) is begin
            case V is
                when False => Put ("0");
                when True  => Put ("1");
            end case;

(continues on next page)
end Show_Bit;

begin
    Put ("Bits: ");
    for E of B loop
        Show_Bit (E);
    end loop;
    New_Line;
end Transmit;
end Serializer;

Listing 52: my_recs.ads

with Serializer; use Serializer;

package My_Recs is

    type Rec is record
        V : Integer;
        S : String (1 .. 3);
    end record;

    procedure To_Rec (B : Bit_Field;
                       R : out Rec);
    function To_Rec (B : Bit_Field) return Rec;
    procedure Display (R : Rec);

end My_Recs;

Listing 53: my_recs.adb

with Ada.Text_IO; use Ada.Text_IO;

package body My_Recs is

    procedure To_Rec (B : Bit_Field;
                       R : out Rec) is
        B_R : Rec
        with Address => B'Address, Import, Volatile;
    begin
        -- Assigning data from overlayed record B_R to output parameter R.
        R := B_R;
    end To_Rec;

    function To_Rec (B : Bit_Field) return Rec is
        R : Rec;
        B_R : Rec
        with Address => B'Address, Import, Volatile;
    begin
        -- Assigning data from overlayed record B_R to local record R.
        R := B_R;
        return R;
    end To_Rec;

    procedure Display (R : Rec) is
    begin
(continues on next page)
Listing 54: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Serializer; use Serializer;
with My_Recs; use My_Recs;

procedure Main is
    R1 : Rec := (5, "abc");
    R2 : Rec := (0, "zzz");
    B1 : Bit_Field (0 .. R1'Size - 1)
        with Address => R1'Address, Import, Volatile;
    begin
        Put ("R2 = ");
        Display (R2);
        New_Line;

        -- Getting Rec type using data from B1, which is a bit-field representation of R1.
        To_Rec (B1, R2);

        -- We could use the function version of To_Rec:
        -- R2 := To_Rec (B1);
        Put_Line ("New bitstream received!");
        Put ("R2 = ");
        Display (R2);
        New_Line;
    end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Deserialization_Ada
MD5: bf5cb5e048ed1f95dba8e85275f6e32

Build output

```
main.adb:18:12: warning: volatile actual passed by copy (RM C.6(19)) [enabled by default]
```

Runtime output

```
R2 = ( 0, zzz)
New bitstream received!
R2 = ( 5, abc)
```

In both versions of To_Rec, we declare the record object B_R as an overlay of the input bit-field. In the procedure version of To_Rec, we then simply copy the data from B_R to the output parameter R. In the function version of To_Rec, however, we need to declare a local record object R, which we return after the assignment.

In C, we can interpret the input pointer as an array of bytes, and copy the individual bytes. For example:

[C]
Listing 55: my_recs.h

typedef struct {
    int v;
    char s[3];
} rec;

void to_r (void *bits, int len, rec *r);

void display_r (rec *r);

Listing 56: my_recs.c

#include "my_recs.h"

#include <stdio.h>
#include <assert.h>

void to_r (void *bits, int len, rec *r)
{
    int i;
    char *c1 = (char *)bits;
    char *c2 = (char *)r;

    assert(len == sizeof(rec) * 8);

    for (i = 0; i < len / (sizeof(char) * 8); i++)
    {
        c2[i] = c1[i];
    }
}

void display_r (rec *r)
{
    printf("%d, %c%c%c", r->v, r->s[0], r->s[1], r->s[2]);
}

Listing 57: bitfield_serialization.c

#include <stdio.h>
#include "my_recs.h"

int main(int argc, const char *argv[])
{
    rec r1 = {5, "abc"};
    rec r2 = {0, "zzz"};

    printf("r2 = \n");
    display_r (&r2);
    printf("\n");

    to_r(&r1, sizeof(r1) * 8, &r2);

    printf("New bitstream received!\n");
    printf("r2 = \n");
    display_r (&r2);
    printf("\n");

    return 0;
}
Here, to_r casts both pointer parameters to pointers to char to get a byte-aligned pointer. Then, it simply copies the data byte-by-byte.

### 71.10.1 Overlays vs. Unchecked Conversions

Unchecked conversions are another way of converting between unrelated data types. This conversion is done by instantiating the generic Unchecked_Conversions function for the types you want to convert. Let’s look at a simple example:

[Ada]

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Conversion;

procedure Simple_Unchecked_Conversion is
  type State is (Off, State_1, State_2)
    with Size => Integer'Size;
  for State use (Off => 0, State_1 => 32, State_2 => 64);
  function As_Integer is new Ada.Unchecked_Conversion (Source => State, Target => Integer);

  I : Integer;
begin
  I := As_Integer (State_2);
  Put_Line ("I = " & Integer'Image (I));
end Simple_Unchecked_Conversion;
```

In this example, As_Integer is an instantiation of Unchecked_Conversion to convert between the State enumeration and the Integer type. Note that, in order to ensure safe conversion, we’re declaring State to have the same size as the Integer type we want to convert to.

This is the corresponding implementation using overlays:

[Ada]
Listing 59: simple_overlay.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Simple_Overlay is
  type State is (Off, State_1, State_2)
    with Size => Integer'Size;
  for State use (Off => 0, State_1 => 32, State_2 => 64);
  S : State;
  I : Integer
    with Address => S'Address, Import, Volatile;
begin
  S := State_2;
  Put_Line ("I = " & Integer'Image (I));
end Simple_Overlay;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Simple_Overlay
MD5: 932135a47c36c406e70b22e075afeaf2

Runtime output

```
I = 64
```

Let's look at another example of converting between different numeric formats. In this case, we want to convert between a 16-bit fixed-point and a 16-bit integer data type. This is how we can do it using Unchecked_Conversion:

[Ada]

Listing 60: fixed_intUnchecked_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Conversion;

procedure Fixed_Int_Unchecked_Conversion is
  Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
  Max_16   : constant := 2 ** 15;
  type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
    with Size => 16;
  type Int_16 is range -Max_16 .. Max_16 - 1
    with Size => 16;
  function As_Int_16 is new Ada.Unchecked_Conversion (Source => Fixed_16,
    Target => Int_16);
  function As_Fixed_16 is new Ada.Unchecked_Conversion (Source => Int_16,
    Target => Fixed_16);
  I : Int_16 := 0;
  F : Fixed_16 := 0.0;
begin
  F := Fixed_16'Last;
  I := As_Int_16 (F);
  Put_Line ("F = " & Fixed_16'Image (F));
  Put_Line ("I = " & Int_16'Image (I));
end Fixed_Int_Unchecked_Conversion;
```

71.10. Mapping Structures to Bit-Fields
Here, we instantiate Unchecked_Conversion for the Int_16 and Fixed_16 types, and we call the instantiated functions explicitly. In this case, we call As_Int_16 to get the integer value corresponding to Fixed_16'Last.

This is how we can rewrite the implementation above using overlays:

[Listing 61: fixed_int_overlay.adb]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Fixed_Int_Overlay is
  Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
  Max_16   : constant := 2 ** 15;

  type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
    with Size => 16;
  type Int_16  is range -Max_16 .. Max_16 - 1
    with Size => 16;

  I : Int_16 := 0;
  F : Fixed_16 with Address => I'Address, Import, Volatile;
begin
  F := Fixed_16'Last;
  Put_Line ('F = ' & Fixed_16'Image (F));
  Put_Line ('I = ' & Int_16'Image (I));
end Fixed_Int_Overlay;
```

Here, the conversion to the integer value is implicit, so we don’t need to call a conversion function.

Using Unchecked_Conversion has the advantage of making it clear that a conversion is happening, since the conversion is written explicitly in the code. With overlays, that conversion is automatic and therefore implicit. In that sense, using an unchecked conversion is a cleaner and safer approach. On the other hand, an unchecked conversion requires a copy, so it’s less efficient than overlays, where no copy is performed — because one change in the source object is automatically reflected in the target object (and vice-versa). In the end, the choice between unchecked conversions and overlays depends on the level of performance that you want to achieve.
Also note that an unchecked conversion only has defined behavior when instantiated for constrained types. For example, we shouldn’t use this kind of conversion:

```
Ada.Unchecked_Conversion (Source => String, Target => Integer);
```

Although this compiles, the behavior will only be well-defined in those cases when Source’Size = Target’Size. Therefore, instead of using an unconstrained type for Source, we should use a subtype that matches this expectation:

```
subtype Integer_String is String (1 .. Integer’Size / Character’Size);
```

function As_Integer is new
    Ada.Unchecked_Conversion (Source => Integer_String, Target => Integer);

Similarly, in order to rewrite the examples using bit-fields that we’ve seen in the previous section, we cannot simply instantiate Unchecked_Conversion with the Target indicating the unconstrained bit-field, such as:

```
Ada.Unchecked_Conversion (Source => Integer, Target => Bit_Field);
```

Instead, we have to declare a subtype for the specific range we’re interested in. This is how we can rewrite one of the previous examples:

[Ada]

Listing 62: simple_bitfield_conversion.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Conversion;

procedure Simple_Bitfield_Conversion is
    type Bit_Field is array (Natural range <>) of Boolean with Pack;
    V : Integer := 4;
    -- Declaring subtype that takes the size of V into account.
    subtype Integer_Bit_Field is Bit_Field (0 .. V’Size - 1);
    -- NOTE: we could also use the Integer type in the declaration:
    -- subtype Integer_Bit_Field is Bit_Field (0 .. Integer’Size - 1);
    --
    -- Using the Integer_Bit_Field subtype as the target
    function As_Bit_Field is new
        Ada.Unchecked_Conversion (Source => Integer, Target => Integer_Bit_Field);
    B : Integer_Bit_Field;
    begin
        B := As_Bit_Field (V);
        Put_Line ("V = " & Integer’Image (V));
    end Simple_Bitfield_Conversion;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Conversion
MD5: 46ead7e5f3da8f261770811d450453e7
Runtime output

V = 4

In this example, we first declare the subtype Integer_Bit_Field as a bit-field with a length that fits the V variable we want to convert to. Then, we can use that subtype in the instantiation of Unchecked_Conversion.
72.1 Understanding static and dynamic variability

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. For example, the same application may need to be portable between two different architectures (ARM and x86), or two different platforms with different set of devices available. Maybe the same application is used for two different generations of the product, so it needs to account for absence or presence of new features, or it's used for different projects which may select different components or configurations. All these cases, and many others, require variability in the software in order to ensure its reusability.

In C, variability is usually achieved through macros and function pointers, the former being tied to static variability (variability in different builds) the latter to dynamic variability (variability within the same build decided at run-time).

Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. When Ada isn't enough, the GNAT compilation system also provides a layer of capabilities, in particular selection of alternate bodies.

If you're familiar with object-oriented programming (OOP) — supported in languages such as C++ and Java —, you might also be interested in knowing that OOP is supported by Ada and can be used to implement variability. This should, however, be used with care, as OOP brings its own set of problems, such as loss of efficiency — dispatching calls can't be inlined and require one level of indirection — or loss of analyzability — the target of a dispatching call isn't known at run time. As a rule of thumb, OOP should be considered only for cases of dynamic variability, where several versions of the same object need to exist concurrently in the same application.

72.2 Handling variability & reusability statically

72.2.1 Genericity

One usage of C macros involves the creation of functions that works regardless of the type they're being called upon. For example, a swap macro may look like:

[C]

Listing 1: main.c

```c
#include <stdio.h>
#include <stdlib.h>

#define SWAP(t, a, b) {{
```

(continues on next page)
t tmp = a; \\
  a = b;  \\
  b = tmp; \\
});

int main()
{
    int a = 10;
    int b = 42;
    printf("a = %d, b = %d\n", a, b);
    SWAP (int, a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Swap_C
MD5: 96d0e8ce9ae985e4de9ed64a0f961f5

Runtime output

a = 10, b = 42
a = 42, b = 10

Ada offers a way to declare this kind of functions as a generic, that is, a function that is written after static arguments, such as a parameter:

[Ada]

Listing 2: main.adb

with Ada.Text_I0; use Ada.Text_I0;

procedure Main is

  generic
    type A_Type is private;
  procedure Swap (Left, Right : in out A_Type);

  procedure Swap (Left, Right : in out A_Type) is
    Temp : constant A_Type := Left;
  begin
    Left := Right;
    Right := Temp;
  end Swap;

  procedure Swap_I is new Swap (Integer);

  A : Integer := 10;
  B : Integer := 42;

begin
  Put_Line ("A = 
    & Integer'Image (A)
    & ", B = 
    & Integer'Image (B));

(continues on next page)
Learning Ada

(continued from previous page)

```ada
Swap_I (A, B);
Put_Line ("A = "
 & Integer'Image (A)
 & ", B = "
 & Integer'Image (B));
end Main;
```

### Code block metadata

**Project:** Courses.Ada_For_Embedded_C_Dev.Reusability.Swap_Ada  
**MD5:** 13f3527b4e3258ebd43be827ad0fcd14

### Runtime output

A = 10, B = 42  
A = 42, B = 10

There are a few key differences between the C and the Ada version here. In C, the macro can be used directly and essentially get expanded by the preprocessor without any kind of checks. In Ada, the generic will first be checked for internal consistency. It then needs to be explicitly instantiated for a concrete type. From there, it's exactly as if there was an actual version of this Swap function, which is going to be called as any other function. All rules for parameter modes and control will apply to this instance.

In many respects, an Ada generic is a way to provide a safe specification and implementation of such macros, through both the validation of the generic itself and its usage.

Subprograms aren't the only entities that can be made generic. As a matter of fact, it's much more common to render an entire package generic. In this case the instantiation creates a new version of all the entities present in the generic, including global variables. For example:

*[Ada]*

**Listing 3: gen.ads**

```ada
generic
  type T is private;
package Gen is
  type C is tagged record
    V : T;
  end record;
  G : Integer;
end Gen;
```

### Code block metadata

**Project:** Courses.Ada_For_Embedded_C_Dev.Reusability.Gen_Pkg_1  
**MD5:** 721f9954561b7e0d2964ba0d226c748b

The above can be instantiated and used the following way:

**Listing 4: main.adb**

```ada
with Gen;
procedure Main is
  package I1 is new Gen (Integer);
```

(continues on next page)
package \texttt{I2} is new \texttt{Gen} (Integer);

subtype \texttt{Str10} is String (1 .. 10);

package \texttt{I3} is new \texttt{Gen} (Str10);

begin
\texttt{I1.G} := 0;
\texttt{I2.G} := 1;
\texttt{I3.G} := 2;
end \texttt{Main};

Here, \texttt{I1.G}, \texttt{I2.G} and \texttt{I3.G} are three distinct variables.

So far, we've only looked at generics with one kind of parameter: a so-called private type. There's actually much more that can be described in this section, such as variables, subprograms or package instantiations with certain properties. For example, the following provides a sort algorithm for any kind of structurally compatible array type:

[Ada]

\begin{verbatim}
generic
  type Component is private;
  type Index is (<>);
  with function "<" (Left, Right : Component) return Boolean;
  type Array_Type is array (Index range <>) of Component;

  procedure Sort (A : in out Array_Type);
\end{verbatim}

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

- \texttt{type T is private; }\texttt{ -- T is a constrained type, such as Integer}
- \texttt{type T (<> is private; \texttt{ -- T can be an unconstrained type e.g. String}}
- \texttt{type T is tagged private; \texttt{ -- T is a tagged type}}
- \texttt{type T is new T2 with private; \texttt{ -- T is an extension of T2}}
- \texttt{type T is (<>); \texttt{ -- T is a discrete type}}
- \texttt{type T is range <>; \texttt{ -- T is an integer type}}
- \texttt{type T is digits <>; \texttt{ -- T is a floating point type}}
- \texttt{type T is access T2; \texttt{ -- T is an access type to T2}}

For a more complete list please reference the Generic Formal Types in the \textit{Appendix of the Introduction to Ada course} (page 269).
### 72.2.2 Simple derivation

Let's take a case where a codebase needs to handle small variations of a given device, or maybe different generations of a device, depending on the platform it's running on. In this example, we're assuming that each platform will lead to a different binary, so the code can statically resolve which set of services are available. However, we want an easy way to implement a new device based on a previous one, saying "this new device is the same as this previous device, with these new services and these changes in existing services".

We can implement such patterns using Ada's simple derivation — as opposed to tagged derivation, which is OOP-related and discussed in a later section.

Let's start from the following example:

[Ada]

```ada
package Drivers_1 is

  type Device_1 is null record;
  procedure Startup (Device : Device_1);
  procedure Send (Device : Device_1; Data : Integer);
  procedure Send_Fast (Device : Device_1; Data : Integer);
  procedure Receive (Device : Device_1; Data : out Integer);

end Drivers_1;
```

```ada
package body Drivers_1 is

  -- NOTE: unimplemented procedures: Startup, Send, Send_Fast
  -- mock-up implementation: Receive

  procedure Startup (Device : Device_1) is null;
  procedure Send (Device : Device_1; Data : Integer) is null;
  procedure Send_Fast (Device : Device_1; Data : Integer) is null;
  procedure Receive (Device : Device_1; Data : out Integer) is
  begin
    Data := 42;
  end Receive;

end Drivers_1;
```

### Code block metadata

**Project:** Courses.Ada_For_EMBEDDED_C_DEV.Reusability.Derived_Drivers

**MD5:** 4f9d7e29b64cda8664438a1d7eed9049

In the above example, `Device_1` is an empty record type. It may also have some fields if required, or be a different type such as a scalar. Then the four procedures `Startup`, `Send`, `Send_Fast` and `Receive` are primitives of this type. A primitive is essentially a subprogram that has a parameter or return type directly referencing this type and declared in the same scope. At this stage, there's nothing special with this type: we're using it as we would use any other type. For example:
Let's now assume that we need to implement a new generation of device, Device_2. This new device works exactly like the first one, except for the startup code that has to be done differently. We can create a new type that operates exactly like the previous one, but modifies only the behavior of Startup:

[Ada]

Listing 9: drivers_2.ads

```ada
with Drivers_1; use Drivers_1;

package Drivers_2 is

   type Device_2 is new Device_1;

   overriding
   procedure Startup (Device : Device_2);

end Drivers_2;
```

Listing 10: drivers_2.adb

```ada
package body Drivers_2 is

   overriding
   procedure Startup (Device : Device_2) is null;

end Drivers_2;
```

Here, Device_2 is derived from Device_1. It contains all the exact same properties and primitives, in particular, Startup, Send, Send_Fast and Receive. However, here, we decided to change the Startup function and to provide a different implementation. We over-
ride this function. The main subprogram doesn't change much, except for the fact that it now relies on a different type:

[Ada]

Listing 11: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Drivers_2; use Drivers_2;

procedure Main is
  D : Device_2;
  I : Integer;
begin
  Startup (D);
  Send_Fast (D, 999);
  Receive (D, I);
  Put_Line (Integer'Image (I));
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 31e7105a99771ce6c1602af117e2e8a6

Runtime output

42

We can continue with this approach and introduce a new generation of devices. This new device doesn't implement the Send_Fast service so we want to remove it from the list of available services. Furthermore, for the purpose of our example, let's assume that the hardware team went back to the Device_1 way of implementing Startup. We can write this new device the following way:

[Ada]

Listing 12: drivers_3.ads

```ada
with Drivers_1; use Drivers_1;

package Drivers_3 is
  type Device_3 is new Device_1;
  overriding
  procedure Startup (Device : Device_3);
  procedure Send_Fast (Device : Device_3; Data : Integer)
    is abstract;
end Drivers_3;
```

Listing 13: drivers_3.adb

```ada
package body Drivers_3 is
  overriding
  procedure Startup (Device : Device_3) is null;
end Drivers_3;
```

Code block metadata
The *is abstract* definition makes illegal any call to a function, so calls to `Send_Fast` on `Device_3` will be flagged as being illegal. To then implement `Startup` of `Device_3` as being the same as the `Startup` of `Device_1`, we can convert the type in the implementation:

```ada
package body Drivers_3 is
    overriding
    procedure Startup (Device : Device_3) is
    begin
        Drivers_1.Startup (Device_1 (Device));
    end Startup;
end Drivers_3;
```

Our Main now looks like:

```ada
procedure Main is
    D : Device_3;
    I : Integer;
begin
    Startup (D);
    Send_Fast (D, 999);
    Receive (D, I);
    Put_Line (Integer’Image (I));
end Main;
```

Here, the call to `Send_Fast` will get flagged by the compiler.

Note that the fact that the code of `Main` has to be changed for every implementation isn’t necessarily satisfactory. We may want to go one step further, and isolate the selection of the device kind to be used for the whole application in one unique file. One way to do this is to use the same name for all types, and use a renaming to select which package to use. Here’s a simplified example to illustrate that:
Learning Ada

Listing 16: drivers_1.ads

```ada
package Drivers_1 is
  type Transceiver is null record;
  procedure Send (Device : Transceiver; Data : Integer);
  procedure Receive (Device : Transceiver; Data : out Integer);
end Drivers_1;
```

Listing 17: drivers_1.adb

```ada
package body Drivers_1 is
  procedure Send (Device : Transceiver; Data : Integer) is null;
  procedure Receive (Device : Transceiver; Data : out Integer) is
    pragma Unreferenced (Device);
    begin
      Data := 42;
    end Receive;
end Drivers_1;
```

Listing 18: drivers_2.ads

```ada
with Drivers_1;

package Drivers_2 is
  type Transceiver is new Drivers_1.Transceiver;
  procedure Send (Device : Transceiver; Data : Integer);
  procedure Receive (Device : Transceiver; Data : out Integer);
end Drivers_2;
```

Listing 19: drivers_2.adb

```ada
package body Drivers_2 is
  procedure Send (Device : Transceiver; Data : Integer) is null;
  procedure Receive (Device : Transceiver; Data : out Integer) is
    pragma Unreferenced (Device);
    begin
      Data := 42;
    end Receive;
end Drivers_2;
```

Listing 20: drivers.ads

```ada
with Drivers_1;

package Drivers renames Drivers_1;
```
In the above example, the whole code can rely on `drivers.ads`, instead of relying on the specific driver. Here, `Drivers` is another name for `Driver_1`. In order to switch to `Driver_2`, the project only has to replace that one `drivers.ads` file.

In the following section, we'll go one step further and demonstrate that this selection can be done through a configuration switch selected at build time instead of a manual code modification.

### 72.2.3 Configuration pragma files

Configuration pragmas are a set of pragmas that modify the compilation of source-code files. You may use them to either relax or strengthen requirements. For example:

```ada
pragma Suppress (Overflow_Check);
```

In this example, we're suppressing the overflow check, thereby relaxing a requirement. Normally, the following program would raise a constraint error due to a failed overflow check:

[Ada]

```ada
package P is
  function Add_Max (A : Integer) return Integer;
end P;
```

```ada
package body P is
  function Add_Max (A : Integer) return Integer is
  begin
    return A + Integer'Last;
  end Add_Max;
end P;
```
Listing 24: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with P; use P;

procedure Main is
  I : Integer := Integer'Last;
begin
  I := Add_Max (I);
  Put_Line ("I = " & Integer'Image (I));
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Constraint_Error_Detection
MD5: d6960fe8ae2af1d66b617bb92d3d47b6

Runtime output

raised CONSTRAINT_ERROR : p.adb:4 overflow check failed

When suppressing the overflow check, however, the program doesn't raise an exception, and the value that Add_Max returns is -2, which is a wraparound of the sum of the maximum integer values (Integer'Last + Integer'Last).

We could also strengthen requirements, as in this example:

```ada
pragma Restrictions (No_Floating_Point);
```

Here, the restriction forbids the use of floating-point types and objects. The following program would violate this restriction, so the compiler isn't able to compile the program when the restriction is used:

```ada
procedure Main is
  F : Float := 0.0;
  -- Declaration is not possible with No_Floating_Point restriction.
begin
  null;
end Main;
```

Restrictions are especially useful for high-integrity applications. In fact, the Ada Reference Manual has a separate section for them.

When creating a project, it is practical to list all configuration pragmas in a separate file. This is called a configuration pragma file, and it usually has an .adc file extension. If you use GPRbuild for building Ada applications, you can specify the configuration pragma file in the corresponding project file. For example, here we indicate that gnat.adc is the configuration pragma file for our project:

```ada
project Default is

  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Main use ("main.adb");

  package Compiler is
    for Local_Configuration_Pragmas use "gnat.adc";
  end Compiler;

(continues on next page)
```

end Default;

72.2.4 Configuration packages

In C, preprocessing flags are used to create blocks of code that are only compiled under certain circumstances. For example, we could have a block that is only used for debugging:

[C]

Listing 25: main.c

```c
#include <stdio.h>
#include <stdlib.h>

int func(int x)
{
    return x % 4;
}

int main()
{
    int a, b;
    a = 10;
    b = func(a);
    #ifdef DEBUG
    printf("func(%d) => %d\n", a, b);
    #endif
    return 0;
}
```

Here, the block indicated by the DEBUG flag is only included in the build if we define this preprocessing flag, which is what we expect for a debug version of the build. In the release version, however, we want to keep debug information out of the build, so we don't use this flag during the build process.

Ada doesn't define a preprocessor as part of the language. Some Ada toolchains — like the GNAT toolchain — do have a preprocessor that could create code similar to the one we've just seen. When programming in Ada, however, the recommendation is to use configuration packages to select code blocks that are meant to be included in the application.

When using a configuration package, the example above can be written as:

[Ada]

Listing 26: config.ads

```ada
package Config is
    Debug : constant Boolean := False;
end Config;
```
Listing 27: func.ads

```
function Func (X : Integer) return Integer;
```

Listing 28: func.adb

```
function Func (X : Integer) return Integer is
begin
  return X mod 4;
end Func;
```

Listing 29: main.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Config;
with Func;

procedure Main is
  A, B : Integer;
begin
  A := 10;
  B := Func (A);
  if Config.Debug then
    Put_Line ("Func(" & Integer'Image (A) & ") => " & Integer'Image (B));
  end if;
end Main;
```

In this example, Config is a configuration package. The version of Config we're seeing here is the release version. The debug version of the Config package looks like this:

```
package Config is
  Debug : constant Boolean := True;
end Config;
```

The compiler makes sure to remove dead code. In the case of the release version, since Config.Debug is constant and set to False, the compiler is smart enough to remove the call to Put_Line from the build.

As you can see, both versions of Config are very similar to each other. The general idea is to create packages that declare the same constants, but using different values.

In C, we differentiate between the debug and release versions by selecting the appropriate preprocessing flags, but in Ada, we select the appropriate configuration package during the build process. Since the file name is usually the same (config.ads for the example above), we may want to store them in distinct directories. For the example above, we could have:

- src/debug/config.ads for the debug version, and
- src/release/config.ads for the release version.

Then, we simply select the appropriate configuration package for each version of the build by indicating the correct path to it. When using GPRbuild, we can select the appropriate
directory where the config.ads file is located. We can use scenario variables in our project, which allow for creating different versions of a build. For example:

```ada
project Default is
    type Mode_Type is ("debug", "release");
    Mode : Mode_Type := external ("mode", "debug");
    for Source_Dirs use ("src", "src/" & Mode);
    for Object_Dir use "obj";
    for Main use ("main.adb");
end Default;
```

In this example, we're defining a scenario type called Mode_Type. Then, we're declaring the scenario variable Mode and using it in the Source_Dirs declaration to complete the path to the subdirectory containing the config.ads file. The expression "src/" & Mode concatenates the user-specified mode to select the appropriate subdirectory.

We can then set the mode on the command-line. For example:

```bash
gprbuild -P default.gpr -Xmode=release
```

In addition to selecting code blocks for the build, we could also specify values that depend on the target build. For our example above, we may want to create two versions of the application, each one having a different version of a MOD_VALUE that is used in the implementation of func(). In C, we can achieve this by using preprocessing flags and defining the corresponding version in APP_VERSION. Then, depending on the value of APP_VERSION, we define the corresponding value of MOD_VALUE.

```c
Listing 30: defs.h

```ifndef APP_VERSION
#define APP_VERSION 1
#endif

```if APP_VERSION == 1
#define MOD_VALUE 4
#endif

```if APP_VERSION == 2
#define MOD_VALUE 5
#endif

```

```c
Listing 31: main.c

```include <stdio.h>
#include <stdlib.h>
#include "defs.h"

int func(int x)
{
    return x % MOD_VALUE;
}

int main()
{
    int a, b;

```
Learning Ada

(continued from previous page)

```ada
a := 10;
b := func(a);
return 0;
}
```

**Code block metadata**

MD5: 9f204dcc65b70618324c48be0dbffbe

If not defined outside, the code above will compile version #1 of the application. We can change this by specifying a value for APP_VERSION during the build (e.g. as a Makefile switch).

For the Ada version of this code, we can create two configuration packages for each version of the application. For example:

[Ada]

Listing 32: app_defs.ads

```ada
-- ./src/app_1/app_defs.ads
package AppDefs is
  Mod_Value : constant Integer := 4;
end AppDefs;
```

Listing 33: func.ads

```ada
function Func (X : Integer) return Integer;
```

Listing 34: func.adb

```ada
with AppDefs;
function Func (X : Integer) return Integer is
begin
  return X mod AppDefs.Mod_Value;
end Func;
```

Listing 35: main.adb

```ada
with Func;
procedure Main is
  A, B : Integer;
begin
  A := 10;
  B := Func (A);
end Main;
```

**Code block metadata**

MD5: 7c8e4280e74c04ab51073b25e8f53995

The code above shows the version #1 of the configuration package. The corresponding

72.2. Handling variability & reusability statically
Learning Ada

implementation for version #2 looks like this:

```ada
-- ./src/app_2/app_defs.ads

package AppDefs is
  Mod_Value : constant Integer := 5;
end AppDefs;
```

Again, we just need to select the appropriate configuration package for each version of the build, which we can easily do when using **GPRbuild**.

### 72.3 Handling variability & reusability dynamically

#### 72.3.1 Records with discriminants

In basic terms, records with discriminants are records that include "parameters" in their type definitions. This allows for adding more flexibility to the type definition. In the section about **pointers** (page 1483), we've seen this example:

[Ada]

Listing 36: main.adb

```
procedure Main is
  type Arr is array (Integer range <>) of Integer;

  type S (Last : Positive) is record
    A : Arr (0 .. Last);
  end record;

  V : S (9);
begin
  null;
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Rec_Disc_Ada
MD5: 02fa8fa7832a262b99ae139a1b5b7a6

Build output

main.adb:8:04: warning: variable "V" is never read and never assigned [-gnatwv]

Here, Last is the discriminant for type S. When declaring the variable V as S (9), we specify the actual index of the last position of the array component A by setting the Last discriminant to 9.

We can create an equivalent implementation in C by declaring a **struct** with a pointer to an array:

[C]

Listing 37: main.c

```
#include <stdio.h>
#include <stdlib.h>
(continues on next page)
typedef struct {
    int * a;
    const int last;
} S;

S init_s (int last)
{
    S v = malloc (sizeof(int) * last + 1), last);
    return v;
}

int main(int argc, const char * argv[])
{
    S v = init_s (9);
    return 0;
}

-- COMPILATION ERROR!

In the C version, we declare the last field constant to get the same behavior.

-- COMPILATION ERROR!

Note that the information provided as discriminants is visible. In the example above, we could display Last by writing:

-- COMPILATION ERROR!

Also note that, even if a type is private, we can still access the information of the discriminants if they are visible in the public part of the type declaration. Let's rewrite the example above:

-- COMPILATION ERROR!

Listing 38: array_definition.ads

package Array_Definition is
    type Arr is array (Integer range <>) of Integer;
    type S (Last : Integer) is private;
private

(continues on next page)
Learning Ada

7

<table>
<thead>
<tr>
<th>type S (Last : Integer) is record</th>
</tr>
</thead>
<tbody>
<tr>
<td>A : Arr (0 .. Last);</td>
</tr>
<tr>
<td>end record;</td>
</tr>
<tr>
<td>end Array_Definition;</td>
</tr>
</tbody>
</table>

Listing 39: main.adb

1

with Ada.Text_IO; use Ada.Text_IO;
with Array_Definition; use Array_Definition;

2

procedure Main is
3

V : S (9);
4

begin
5

Put_Line ("Last : " & Integer'Image (V.Last));
6

end Main;

7

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Rec_Disc_Ada_Private
MD5: fa0158c3c61dd9ec7e4000416672f9e9

Build output

main.adb:5:04: warning: variable "V" is read but never assigned [-gnatwv]

Runtime output

Last : 9

Even though the S type is now private, we can still display Last because this discriminant is visible in the non-private part of package Array_Definition.

72.3.2 Variant records

In simple terms, a variant record is a record with discriminants that allows for changing its structure. Basically, it's a record containing a case. This is the general structure:

[Ada]

type Var_Rec (V : F) is record

  case V is
  when Opt_1 => F1 : Type_1;
  when Opt_2 => F2 : Type_2;
  end case;

end record;

Let's look at this example:

[Ada]

Listing 40: main.adb

1

with Ada.Text_IO; use Ada.Text_IO;

2

procedure Main is

3

(continues on next page)
type Float_Int (Use_Float : Boolean) is record
  case Use_Float is
  when True => F : Float;
  when False => I : Integer;
  end case;
end record;

procedure Display (V : Float_Int) is
begin
  if V.Use_Float then
    Put_Line ("Float value: " & Float'Image (V.F));
  else
    Put_Line ("Integer value: " & Integer'Image (V.I));
  end if;
end Display;

F : constant Float_Int := (Use_Float => True, F => 10.0);
I : constant Float_Int := (Use_Float => False, I => 9);

begin
  Display (F);
  Display (I);
end Main;

Here, we declare F containing a floating-point value, and I containing an integer value. In the Display procedure, we present the correct information to the user according to the Use_Float discriminant of the Float_Int type.

We can implement this example in C by using unions:

[C]

Listing 41: main.c

#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int use_float;
  union {
    float f;
    int i;
  };
} float_int;

float_int init_float (float f)
{
  float_int v;
  v.use_float = 1;
  v.f = f;
  return v;
}
Learning Ada

(float_int init_int (int i)
{
    float_int v;
    v.use_float = 0;
    v.i = i;
    return v;
}

void display (float_int v)
{
    if (v.use_float) {
        printf("Float value : %f\n", v.f);
    } else {
        printf("Integer value : %d\n", v.i);
    }
}

int main(int argc, const char * argv[])
{
    float_int f = init_float (10.0);
    float_int i = init_int (9);
    display (f);
    display (i);
    return 0;
}

Code block metadata

MD5: ac0ad1e6ff72154e9dbb6838999a62e

Runtime output

Float value  : 10.000000
Integer value  : 9

Similar to the Ada code, we declare f containing a floating-point value, and i containing an
integer value. One difference is that we use the init_float() and init_int() functions
to initialize the float_int struct. These functions initialize the correct field of the union
and set the use_float field accordingly.

Variant records and unions

There is, however, a difference in accessibility between variant records in Ada and unions
in C. In C, we're allowed to access any field of the union regardless of the initialization:

[C]

float_int v = init_float (10.0);
printf("Integer value : %d\n", v.i);

This feature is useful to create overlays. In this specific example, however, the information
displayed to the user doesn't make sense, since the union was initialized with a floating-
point value (v.f) and, by accessing the integer field (v.i), we're displaying it as if it was an integer value.

In Ada, accessing the wrong component would raise an exception at run-time ("discriminant check failed"), since the component is checked before being accessed:

[Ada]

```ada
V : constant Float_Int := (Use_Float => True, F => 10.0);
begin
    Put_Line ("Integer value: " & Integer'Image (V.I));
    -- ^ Constraint_Error is raised!
```

Using this method prevents wrong information being used in other parts of the program.
To get the same behavior in Ada as we do in C, we need to explicitly use the Unchecked_Union aspect in the type declaration. This is the modified example:

[Ada]

```ada
Listing 42: main.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    type Float_Int_Union (Use_Float : Boolean) is record
        case Use_Float is
            when True => F : Float;
            when False => I : Integer;
        end case;
    end record
    with Unchecked_Union;

    V : constant Float_Int_Union := (Use_Float => True, F => 10.0);

begin
    Put_Line ("Integer value: " & Integer'Image (V.I));
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Unchecked_Union_Ada
MD5: f6c5eacbd96c23531d02bb47a9668ac5

Runtime output

Integer value: 1092616192

Now, we can display the integer component (V.I) even though we initialized the floating-point component (V.F). As expected, the information displayed by the test application in this case doesn't make sense.

Note that, when using the Unchecked_Union aspect in the declaration of a variant record, the reference discriminant is not available anymore, since it isn't stored as part of the record. Therefore, we cannot access the Use_Float discriminant as in the following code:

[Ada]

```ada
V : constant Float_Int_Union := (Use_Float => True, F => 10.0);
begin
    if V.Use_Float then
        -- Do something... -- COMPILATION ERROR!
    end if;
```

72.3. Handling variability & reusability dynamically 1527
Unchecked unions are particularly useful in Ada when creating bindings for C code.

**Optional components**

We can also use variant records to specify optional components of a record. For example:

[Ada]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Arr is array (Integer range <>) of Integer;
  type Extra_Info is (No, Yes);
  type S_Var (Last : Integer; Has_Extra_Info : Extra_Info) is record
    case Has_Extra_Info is
      when No => null;
      when Yes => B : Arr (0 .. Last);
    end case;
  end record;

V1 : S_Var (Last => 9, Has_Extra_Info => Yes);
V2 : S_Var (Last => 9, Has_Extra_Info => No);
begin
  Put_Line ("Size of V1 is: " & Integer'Image (V1'Size));
  Put_Line ("Size of V2 is: " & Integer'Image (V2'Size));
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_DEV.Reusability.Var_Rec_Null_Ada
MD5: 548235fa8458302ba025c8fa49e61777

Build output

main.adb:17:04: warning: variable "V1" is read but never assigned [-gnatwv]
main.adb:18:04: warning: variable "V2" is read but never assigned [-gnatwv]

Runtime output

Size of V1 is: 704
Size of V2 is: 384

Here, in the declaration of S_Var, we don't have any component in case Has_Extra_Info is false. The component is simply set to null in this case.

When running the example above, we see that the size of V1 is greater than the size of V2 due to the extra B component — which is only included when Has_Extra_Info is true.
Optional output information

We can use optional components to prevent subprograms from generating invalid information that could be misused by the caller. Consider the following example:

[C]

Listing 44: main.c

```c
#include <stdio.h>
#include <stdlib.h>

float calculate (float f1,
                 float f2,
                 int *success)
{
    if (f1 < f2) {
        *success = 1;
        return f2 - f1;
    } else {
        *success = 0;
        return 0.0;
    }
}

void display (float v,
              int success)
{
    if (success) {
        printf("Value = %f\n", v);
    } else {
        printf("Calculation error!\n");
    }
}

int main(int argc, const char * argv[])
{
    float f;
    int success;
    
    f = calculate (1.0, 0.5, &success);
    display (f, success);
    
    f = calculate (0.5, 1.0, &success);
    display (f, success);
    
    return 0;
}
```

In this code, we're using the output parameter success of the calculate() function to indicate whether the calculation was successful or not. This approach has a major problem:
there's no way to prevent that the invalid value returned by calculate() in case of an error is misused in another computation. For example:

[C]

```c
int main(int argc, const char * argv[])
{
    float f;
    int success;

    f = calculate (1.0, 0.5, &success);
    f = f * 0.25; // Using f in another computation even though
                  // calculate() returned a dummy value due to error!
                  // We should have evaluated "success", but we didn't.

    return 0;
}
```

We cannot prevent access to the returned value or, at least, force the caller to evaluate success before using the returned value.

This is the corresponding code in Ada:

[Ada]

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    function Calculate (F1, F2 : Float;
                        Success : out Boolean) return Float is
    begin
        if F1 < F2 then
            Success := True;
            return F2 - F1;
        else
            Success := False;
            return 0.0;
        end if;
    end Calculate;

    procedure Display (V : Float; Success : Boolean) is
    begin
        if Success then
            Put_Line ("Value = " & Float'Image (V));
        else
            Put_Line ("Calculation error!");
        end if;
    end Display;

    F : Float;
    Success : Boolean;

begin
    F := Calculate (1.0, 0.5, Success);
    Display (F, Success);
    F := Calculate (0.5, 1.0, Success);
    Display (F, Success);
end Main;
```

Code block metadata

Chapter 72. Handling Variability and Re-usability
The Ada code above suffers from the same drawbacks as the C code. Again, there’s no way to prevent misuse of the invalid value returned by `Calculate` in case of errors.

However, in Ada, we can use variant records to make the component unavailable and therefore prevent misuse of this information. Let’s rewrite the original example and wrap the returned value in a variant record:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

  type Opt_Float (Success : Boolean) is record
    case Success is
    when False => null;
    when True  => F : Float;
    end case;
  end record;

  function Calculate (F1, F2 : Float) return Opt_Float is
  begin
    if F1 < F2 then
      return (Success => True, F => F2 - F1);
    else
      return (Success => False);
    end if;
  end Calculate;

  procedure Display (V : Opt_Float) is
  begin
    if V.Success then
      Put_Line ("Value = " & Float'Image (V.F));
    else
      Put_Line ("Calculation error!");
    end if;
  end Display;

begin
  Display (Calculate (1.0, 0.5));
  Display (Calculate (0.5, 1.0));
end Main;
```

Runtime output

Calculation error!
Value =  5.00000E-01
In this example, we can determine whether the calculation was successful or not by evaluating the Success component of the Opt_Float. If the calculation wasn't successful, we won't be able to access the F component of the Opt_Float. As mentioned before, trying to access the component in this case would raise an exception. Therefore, in case of errors, we can ensure that no information is misused after the call to Calculate.

### 72.3.3 Object orientation

In the previous section (page 1522), we've seen that we can add variability to records by using discriminants. Another approach is to use tagged records, which are the base for object-oriented programming in Ada.

#### Type extension

A tagged record type is declared by adding the tagged keyword. For example:

[Ada]

```ada
procedure Main is
    type Rec is record
        V : Integer;
    end record;

    type Tagged_Rec is tagged record
        V : Integer;
    end record;

    R1 : Rec;
    R2 : Tagged_Rec;

begin
    R1 := (V => 0);
    R2 := (V => 0);
end Main;
```

**Code block metadata**

Project: Courses.Ada For Embedded C Dev.Reusability.Tagged_Type_Decl
MD5: 53810d3bb5aa7e7b1483270d974eb025

In this simple example, there isn't much difference between the Rec and Tagged_Rec type. However, tagged types can be derived and extended. For example:

[Ada]

```ada
procedure Main is
    type Rec is record
        V : Integer;
    end record;

    -- We cannot declare this:
    --
```

(continues on next page)
-- type Ext_Rec is new Rec with record
--  V : Integer;
-- end record;

type Tagged_Rec is tagged record
  V : Integer;
end record;

-- But we can declare this:
--
-- type Ext_Tagged_Rec is new Tagged_Rec with record
--  V2 : Integer;
end record;

R1 : Rec;
R2 : Tagged_Rec;
R3 : Ext_Tagged_Rec;

pragma Unreferenced (R1, R2, R3);

begin
  R1 := (V => 0);
  R2 := (V => 0);
  R3 := (V => 0, V2 => 0);
end Main;

As indicated in the example, a type derived from an untagged type cannot have an extension. The compiler indicates this error if you uncomment the declaration of the Ext_Rec type above. In contrast, we can extend a tagged type, as we did in the declaration of Ext_Tagged_Rec. In this case, Ext_Tagged_Rec has all the components of the Tagged_Rec type (V, in this case) plus the additional components from its own type declaration (V2, in this case).

Override subprograms

Previously, we've seen that subprograms can be overridden. For example, if we had implemented a Reset and a Display procedure for the Rec type that we declared above, these procedures would be available for an Ext_Rec type derived from Rec. Also, we could override these procedures for the Ext_Rec type. In Ada, we don't need object-oriented programming features to do that: simple (untagged) records can be used to derive types, inherit operations and override them. However, in applications where the actual subprogram to be called is determined dynamically at run-time, we need dispatching calls. In this case, we must use tagged types to implement this.
Comparing untagged and tagged types

Let's discuss the similarities and differences between untagged and tagged types based on this example:

[Ada]

```ada
package P is
    type Rec is record
        V : Integer;
    end record;

    procedure Display (R : Rec);
    procedure Reset (R : out Rec);

    type New_Rec is new Rec;
    overriding procedure Display (R : New_Rec);
    not overriding procedure New_Op (R : in out New_Rec);

    type Tagged_Rec is tagged record
        V : Integer;
    end record;

    procedure Display (R : Tagged_Rec);
    procedure Reset (R : out Tagged_Rec);

    type Ext_Tagged_Rec is new Tagged_Rec with record
        V2 : Integer;
    end record;

    overriding procedure Display (R : Ext_Tagged_Rec);
    overriding procedure Reset (R : out Ext_Tagged_Rec);
    not overriding procedure New_Op (R : in out Ext_Tagged_Rec);

end P;
```

Listing 50: p.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body P is

    procedure Display (R : Rec) is
    begin
        Put_Line ("TYPE: REC");
        Put_Line ("Rec.V = " & Integer'Image (R.V));
        New_Line;
    end Display;

    procedure Reset (R : out Rec) is
    begin
        R.V := 0;
    end Reset;

    procedure Display (R : New_Rec) is
    begin
        Put_Line ("TYPE: NEW REC");
        Put_Line ("New_Rec.V = " & Integer'Image (R.V));
    end Display;

    procedure Display (R : Tagged_Rec) is
    begin
        Put_Line ("TYPE: TAGGED REC");
        Put_Line ("Tagged_Rec.V = " & Integer'Image (R.V));
    end Display;

    procedure Display (R : Ext_Tagged_Rec) is
    begin
        Put_Line ("TYPE: EXT_TAGGED REC");
        Put_Line ("Ext_Tagged_Rec.V = " & Integer'Image (R.V));
    end Display;

end P;
```
New_Line;
end Display;

procedure New_Op (R : in out New_Rec) is
begin
  R.V := R.V + 1;
end New_Op;

procedure Display (R : Tagged_Rec) is
begin
  -- Using External_Tag attribute to retrieve the tag as a string
  Put_Line ("TYPE: " & Tagged_Rec'External_Tag);
  Put_Line ("Tagged_Rec.V = " & Integer'Image (R.V));
  New_Line;
end Display;

procedure Reset (R : out Tagged_Rec) is
begin
  R.V := 0;
end Reset;

procedure Display (R : Ext_Tagged_Rec) is
begin
  -- Using External_Tag attribute to retrieve the tag as a string
  Put_Line ("TYPE: " & Ext_Tagged_Rec'External_Tag);
  Put_Line ("Ext_Tagged_Rec.V = " & Integer'Image (R.V));
  Put_Line ("Ext_Tagged_Rec.V2 = " & Integer'Image (R.V2));
  New_Line;
end Display;

procedure Reset (R : out Ext_Tagged_Rec) is
begin
  Tagged_Rec (R).Reset;
  R.V2 := 0;
end Reset;

procedure New_Op (R : in out Ext_Tagged_Rec) is
begin
  R.V := R.V + 1;
end New_Op;
end P;

Listing 51: main.adb

with Ada.Text_IO; use Ada.Text_IO;
with P; use P;

procedure Main is
  X_Rec : Rec;
  X_New_Rec : New_Rec;
  X_Tagged_Rec : aliased Tagged_Rec;
  X_Ext_Tagged_Rec : aliased Ext_Tagged_Rec;
  X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
begin
-- Reset all objects
-- (continues on next page)
```
17 Reset (X_Rec);
18 Reset (X_New_Rec);
19 X_Tagged_Rec.Reset;  --  we could write "Reset (X_Tagged_Rec)" as well
20 X_Ext_Tagged_Rec.Reset;

22 --  Use new operations when available
23 New_Op (X_New_Rec);
24 X_Ext_Tagged_Rec.New_Op;

28 --  Display all objects
29 Display (X_Rec);
30 Display (X_New_Rec);
31 X_Tagged_Rec.Display;  --  we could write "Display (X_Tagged_Rec)" as well
32 X_Ext_Tagged_Rec.Display;

37 --  Resetting and display objects of Tagged_Rec'Class
38 Put_Line ("Operations on Tagged_Rec'Class");
39 Put_Line ("-----------------------------");
41 for E of X_Tagged_Rec_Array loop
42  E.Reset;
43  E.Display;
44 end loop;
45 end Main;
```

**Code block metadata**

- **Project:** Courses.Ada_For_Embedded_C.Dev.Reusability.Tagged_Type_Extension_Decl
- **MD5:** 29412b74db6b80f8a8986b6e5284cf7

**Runtime output**

- **TYPE:** REC
  - Rec.V = 0

- **TYPE:** NEW_REC
  - New_Rec.V = 1

- **TYPE:** P.TAGGED_REC
  - Tagged_Rec.V = 0

- **TYPE:** P.EXT_TAGGED_REC
  - Ext_Tagged_Rec.V = 1
  - Ext_Tagged_Rec.V2 = 0

**Operations on Tagged_Rec'Class**

- **TYPE:** P.TAGGED_REC
  - Tagged_Rec.V = 0

- **TYPE:** P.EXT_TAGGED_REC
  - Ext_Tagged_Rec.V = 0
  - Ext_Tagged_Rec.V2 = 0

These are the similarities between untagged and tagged types:
• We can derive types and inherit operations in both cases.
  - Both X_New_Rec and X_Ext_Tagged_Rec inherit the Display and Reset procedures from their respective ancestors.
• We can override operations in both cases.
• We can implement new operations in both cases.
  - Both X_New_Rec and X_Ext_Tagged_Rec implement a procedure called New_Op, which is not available for their respective ancestors.

Now, let's look at the differences between untagged and tagged types:
• We can dispatch calls for a given type class.
  - This is what we do when we iterate over objects of the Tagged_Rec class — in the loop over X_Tagged_Rec_Array at the last part of the Main procedure.
• We can use the dot notation.
  - We can write both E.Reset or Reset (E) forms: they're equivalent.

**Dispatching calls**

Let's look more closely at the dispatching calls implemented above. First, we declare the X_Tagged_Rec_Array array and initialize it with the access to objects of both parent and derived tagged types:

[Ada]

```ada
X_Tagged_Rec : aliased Tagged_Rec;
X_Ext_Tagged_Rec : aliased Ext_Tagged_Rec;
X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
```

Here, we use the `aliased` keyword to be able to get access to the objects (via the `Access` attribute).

Then, we loop over this array and call the Reset and Display procedures:

[Ada]

```ada
for E of X_Tagged_Rec_Array loop
  E.Reset;
  E.Display;
end loop;
```

Since we're using dispatching calls, the actual procedure that is selected depends on the type of the object. For the first element (X_Tagged_Rec_Array (1)), this is Tagged_Rec, while for the second element (X_Tagged_Rec_Array (2)), this is Ext_Tagged_Rec.

Dispatching calls are only possible for a type class — for example, the Tagged_Rec'Class. When the type of an object is known at compile time, the calls won't dispatch at runtime. For example, the call to the Reset procedure of the X_Ext_Tagged_Rec object (X_Ext_Tagged_Rec.Reset) will always take the overridden Reset procedure of the Ext_Tagged_Rec type. Similarly, if we perform a view conversion by writing Tagged_Rec (A_Ext_Tagged_Rec).Display, we're instructing the compiler to interpret A_Ext_Tagged_Rec as an object of type Tagged_Rec, so that the compiler selects the Display procedure of the Tagged_Rec type.
Another useful feature of object-oriented programming is the use of interfaces. In this case, we can define abstract operations, and implement them in the derived tagged types. We declare an interface by simply writing `type T is interface`. For example:

```
[Ada]
type My_Interface is interface;
procedure Op (Obj : My_Interface) is abstract;
-- We cannot declare actual objects of an interface:
-- Obj : My_Interface; -- ERROR!
```

All operations on an interface type are abstract, so we need to write `is abstract` in the signature — as we did in the declaration of `Op` above. Also, since interfaces are abstract types and don't have an actual implementation, we cannot declare objects for it.

We can derive tagged types from an interface and implement the actual operations of that interface:

```
[Ada]
type My_Derived is new My_Interface with null record;
procedure Op (Obj : My_Derived);
```

Note that we're not using the `tagged` keyword in the declaration because any type derived from an interface is automatically tagged.

Let's look at an example with an interface and two derived tagged types:

```
Listing 52: p.ads
package P is
  type Display_Interface is interface;
  procedure Display (D : Display_Interface) is abstract;

  type Small_Display_Type is new Display_Interface with null record;
  procedure Display (D : Small_Display_Type);

  type Big_Display_Type is new Display_Interface with null record;
  procedure Display (D : Big_Display_Type);
end P;
```

```
Listing 53: p.adb
with Ada.Text_IO; use Ada.Text_IO;
package body P is
  procedure Display (D : Small_Display_Type) is
    pragma Unreferenced (D);
  begin
    Put_Line ("Using Small_Display_Type");
  end Display;
end P;
```

(continues on next page)
procedure Display (D : Big_Display_Type) is
pragma Unreferenced (D);
begin
Put_Line ("Using Big_Display_Type");
end Display;
end P;

Listing 54: main.adb

with P; use P;

procedure Main is
  D_Small : Small_Display_Type;
  D_Big : Big_Display_Type;

  procedure Dispatching_Display (D : Display_Interface'Class) is
  begin
    D.Display;
  end Dispatching_Display;

begin
  Dispatching_Display (D_Small);
  Dispatching_Display (D_Big);
end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Interfaces_1
MD5: 564eba158b2f8fc3efea9e892a21ca9

Runtime output

Using Small_Display_Type
Using Big_Display_Type

In this example, we have an interface type Display_Interface and two tagged types that are derived from Display_Interface: Small_Display_Type and Big_Display_Type.

Both types (Small_Display_Type and Big_Display_Type) implement the interface by overriding the Display procedure. Then, in the inner procedure Dispatching_Display of the Main procedure, we perform a dispatching call depending on the actual type of D.

Deriving from multiple interfaces

We may derive a type from multiple interfaces by simply writing type Derived_T is new T1 and T2 with null record. For example:

[Ada]

package Transceivers is
  type Send_Interface is interface;
  procedure Send (Obj : in out Send_Interface) is abstract;
  type Receive_Interface is interface;
end Transceivers;

(continues on next page)
procedure Receive (Obj : in out Receive_Interface) is abstract;

type Transceiver is new Send_Interface and Receive_Interface
  with null record;

  procedure Send (D : in out Transceiver);
  procedure Receive (D : in out Transceiver);
end Transceivers;

Listing 56: transceivers.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Transceivers is
  procedure Send (D : in out Transceiver) is
    pragma Unreferenced (D);
    begin
      Put_Line ("Sending data...");
      end Send;

  procedure Receive (D : in out Transceiver) is
    pragma Unreferenced (D);
    begin
      Put_Line ("Receiving data...");
      end Receive;
end Transceivers;

Listing 57: main.adb

with Transceivers; use Transceivers;

procedure Main is
  D : Transceiver;
begin
  D.Send;
  D.Receive;
end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Multiple_Interfaces
MD5: c81813941bd3458eaf7b1fd39b010a03

Runtime output

Sending data...
Receiving data...

In this example, we're declaring two interfaces (Send_Interface and Receive_Interface) and the tagged type Transceiver that derives from both interfaces. Since we need to implement the interfaces, we implement both Send and Receive for Transceiver.
Abstract tagged types

We may also declare abstract tagged types. Note that, because the type is abstract, we cannot use it to declare objects for it — this is the same as for interfaces. We can only use it to derive other types. Let’s look at the abstract tagged type declared in the Abstract_Transceivers package:

[Ada]

Listing 58: abstract_transceivers.ads

```ada
with Transceivers; use Transceivers;
package Abstract_Transceivers is
  type Abstract_Transceiver is abstract new Send_Interface and Receive_Interface with null record;
  procedure Send (D : in out Abstract_Transceiver);
  -- We don't implement Receive for Abstract_Transceiver!
end Abstract_Transceivers;
```

Listing 59: abstract_transceivers.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Abstract_Transceivers is
  procedure Send (D : in out Abstract_Transceiver) is
    pragma Unreferenced (D);
  begin
    Put_Line ("Sending data...");
  end Send;
end Abstract_Transceivers;
```

Listing 60: main.adb

```ada
with Abstract_Transceivers; use Abstract_Transceivers;
procedure Main is
  D : Abstract_Transceiver;
begin
  D.Send;
  D.Receive;
end Main;
```

In this example, we declare the abstract tagged type Abstract_Transceiver. Here, we’re only partially implementing the interfaces from which this type is derived: we’re implementing Send, but we’re skipping the implementation of Receive. Therefore, Receive is
Learning Ada

an abstract operation of Abstract_Transceiver. Since any tagged type that has abstract operations is abstract, we must indicate this by adding the abstract keyword in type declaration.

Also, when compiling this example, we get an error because we're trying to declare an object of Abstract_Transceiver (in the Main procedure), which is not possible. Naturally, if we derive another type from Abstract_Transceiver and implement Receive as well, then we can declare objects of this derived type. This is what we do in the Full_Transceivers below:

[Ada]

Listing 61: full_transceivers.ads

```ada
with Abstract_Transceivers; use Abstract_Transceivers;

package Full_Transceivers is
  type Full_Transceiver is new Abstract_Transceiver with null record;
  procedure Receive (D : in out Full_Transceiver);
end Full_Transceivers;
```

Listing 62: full_transceivers.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Full_Transceivers is
  procedure Receive (D : in out Full_Transceiver) is
    pragma Unreferenced (D);
  begin
    Put_Line ("Receiving data...");
  end Receive;
end Full_Transceivers;
```

Listing 63: main.adb

```ada
with Full_Transceivers; use Full_Transceivers;

procedure Main is
  D : Full_Transceiver;
begin
  D.Send;
  D.Receive;
end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Multiple_Interfaces
MD5: 77a86a6d917547d306a89422e7522111

Runtime output

Sending data...
Receiving data...

Here, we implement the Receive procedure for the Full_Transceiver. Therefore, the type doesn't have any abstract operation, so we can use it to declare objects.
Learning Ada

From simple derivation to OOP

In the section about simple derivation (page 1511), we've seen an example where the actual selection was done at implementation time by renaming one of the packages:

[Ada]

```ada
with Drivers_1;
package Drivers renames Drivers_1;
```

Although this approach is useful in many cases, there might be situations where we need to select the actual driver dynamically at runtime. Let's look at how we could rewrite that example using interfaces, tagged types and dispatching calls:

[Ada]

Listing 64: drivers_base.ads

```ada
package Drivers_Base is

  type Transceiver is interface;

  procedure Send (Device : Transceiver; Data : Integer) is abstract;
  procedure Receive (Device : Transceiver; Data : out Integer) is abstract;
  procedure Display (Device : Transceiver) is abstract;

end Drivers_Base;
```

Listing 65: drivers_1.ads

```ada
with Drivers_Base;

package Drivers_1 is

  type Transceiver is new Drivers_Base.Transceiver with null record;

  procedure Send (Device : Transceiver; Data : Integer);
  procedure Receive (Device : Transceiver; Data : out Integer);
  procedure Display (Device : Transceiver);

end Drivers_1;
```

Listing 66: drivers_1.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Drivers_1 is

  procedure Send (Device : Transceiver; Data : Integer) is null;
  procedure Receive (Device : Transceiver; Data : out Integer) is
    pragma Unreferenced (Device);
    begin
      Data := 42;
    end Receive;

  procedure Display (Device : Transceiver) is
    pragma Unreferenced (Device);
    begin
      Put_Line ("Using Drivers_1");
    end Display;
```

(continues on next page)

72.3. Handling variability & reusability dynamically 1543
Learning Ada

with Drivers_Base;

package Drivers_2 is

  type Transceiver is new Drivers_Base.Transceiver with null record;

  procedure Send (Device : Transceiver; Data : Integer);
  procedure Receive (Device : Transceiver; Data : out Integer);
  procedure Display (Device : Transceiver);

end Drivers_2;

with Ada.Text_IO; use Ada.Text_IO;

package body Drivers_2 is

  procedure Send (Device : Transceiver; Data : Integer) is null;
  procedure Receive (Device : Transceiver; Data : out Integer) is
    pragma Unreferenced (Device);
    begin
      Data := 7;
      end Receive;
  procedure Display (Device : Transceiver) is
    pragma Unreferenced (Device);
    begin
      Put_Line ("Using Drivers_2");
      end Display;

end Drivers_2;

with Ada.Text_IO; use Ada.Text_IO;

with Drivers_Base;
with Drivers_1;
with Drivers_2;

procedure Main is
  D1 : aliased Drivers_1.Transceiver;
  D2 : aliased Drivers_2.Transceiver;
  D : access Drivers_Base.Transceiver'Class;
  I : Integer;

  type Driver_Number is range 1 .. 2;

  procedure Select_Driver (N : Driver_Number) is
    begin
      if N = 1 then
        D := D1'Access;
      end if;

end Main;
In this example, we declare the Transceiver interface in the Drivers_Base package. This interface is then used to derive the tagged types Transceiver from both Drivers_1 and Drivers_2 packages.

In the Main procedure, we use the access to Transceiver'Class — from the interface declared in the Drivers_Base package — to declare D. This object D contains the access to the actual driver loaded at any specific time. We select the driver at runtime in the inner Select_Driver procedure, which initializes D (with the access to the selected driver). Then, any operation on D triggers a dispatching call to the selected driver.

Further resources

In the appendices, we have a step-by-step hands-on overview of object-oriented programming (page 1577) that discusses how to translate a simple system written in C to an equivalent system in Ada using object-oriented programming.

72.3.4 Pointer to subprograms

Pointers to subprograms allow us to dynamically select an appropriate subprogram at runtime. This selection might be triggered by an external event, or simply by the user. This can be useful when multiple versions of a routine exist, and the decision about which one to use cannot be made at compilation time.

This is an example on how to declare and use pointers to functions in C:

[C]
Learning Ada

Listing 70: main.c

```
#include <stdio.h>
#include <stdlib.h>

void show_msg_v1 (char *msg)
{
    printf("Using version #1: %s\n", msg);
}

void show_msg_v2 (char *msg)
{
    printf("Using version #2: \n %s\n", msg);
}

int main()
{
    int selection = 1;
    void (*current_show_msg) (char *);

    switch (selection)
    {
    case 1: current_show_msg = &show_msg_v1; break;
    case 2: current_show_msg = &show_msg_v2; break;
    default: current_show_msg = NULL; break;
    }

    if (current_show_msg != NULL)
    {
        current_show_msg ("Hello there!");
    }
    else
    {
        printf("ERROR: no version of show_msg() selected!\n");
    }

    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embded_C_Dev.Reusability.Selecting_Subprogram_C
MD5: 414c99fca2490611d20d031f8549ff59

Runtime output

Using version #1: Hello there!

The example above contains two versions of the show_msg() function: show_msg_v1() and show_msg_v2(). The function is selected depending on the value of selection, which initializes the function pointer current_show_msg. If there's no corresponding value, current_show_msg is set to null — alternatively, we could have selected a default version of show_msg() function. By calling current_show_msg ("Hello there!"), we're calling the function that current_show_msg is pointing to.

This is the corresponding implementation in Ada:

[Ada]
Listing 71: show_subprogram_selection.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Subprogram_Selection is

  procedure Show_Msg_V1 (Msg : String) is
  begin
    Put_Line ("Using version #1: " & Msg);
  end Show_Msg_V1;

  procedure Show_Msg_V2 (Msg : String) is
  begin
    Put_Line ("Using version #2: ");
    Put_Line (Msg);
  end Show_Msg_V2;

  type Show_Msg_Proc is access procedure (Msg : String);
  Current_Show_Msg : Show_Msg_Proc;
  Selection       : Natural;

begin
  Selection := 1;

  case Selection is
    when 1 => Current_Show_Msg := Show_Msg_V1'Access;
    when 2 => Current_Show_Msg := Show_Msg_V2'Access;
    when others => Current_Show_Msg := null;
  end case;

  if Current_Show_Msg /= null then
    Current_Show_Msg ("Hello there!");
  else
    Put_Line ("ERROR: no version of Show_Msg selected!");
  end if;

end Show_Subprogram_Selection;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Selecting_Subprogram_Ada
MD5: ee41e042e3b879b4a2671bf66d8072aa

Runtime output

Using version #1: Hello there!

The structure of the code above is very similar to the one used in the C code. Again, we have two version of Show_Msg: Show_Msg_V1 and Show_Msg_V2. We set Current_Show_Msg according to the value of Selection. Here, we use 'Access to get access to the corresponding procedure. If no version of Show_Msg is available, we set Current_Show_Msg to null.

Pointers to subprograms are also typically used as callback functions. This approach is extensively used in systems that process events, for example. Here, we could have a two-layered system:

- A layer of the system (an event manager) triggers events depending on information from sensors.
  - For each event, callback functions can be registered.

72.3. Handling variability & reusability dynamically 1547
- The event manager calls registered callback functions when an event is triggered.
  - Another layer of the system registers callback functions for specific events and decides what to do when those events are triggered.

This approach promotes information hiding and component decoupling because:
  - the layer of the system responsible for managing events doesn't need to know what the callback function actually does, while
  - the layer of the system that implements callback functions remains agnostic to implementation details of the event manager — for example, how events are implemented in the event manager.

Let's see an example in C where we have a `process_values()` function that calls a callback function (`process_one`) to process a list of values:

[C]

Listing 72: process_values.h

```c
typedef int (*process_one_callback) (int);
void process_values (int *values, int len, process_one_callback process_one);
```

Listing 73: process_values.c

```c
#include "process_values.h"
#include <stdio.h>
#include <stdlib.h>
void process_values (int *values, int len, process_one_callback process_one)
{
    int i;
    assert (process_one != NULL);
    for (i = 0; i < len; i++)
    {
        values[i] = process_one (values[i]);
    }
}
```

Listing 74: main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "process_values.h"

int proc_10 (int val)
{
    return val + 10;
}

#define LEN_VALUES 5

int main()
```

(continues on next page)
As mentioned previously, `process_values()` doesn't have any knowledge about what `process_one()` does with the integer value it receives as a parameter. Also, we could replace `proc_10()` by another function without having to change the implementation of `process_values()`.

Note that `process_values()` calls an `assert()` for the function pointer to compare it against null. Here, instead of checking the validity of the function pointer, we're expecting the caller of `process_values()` to provide a valid pointer.

This is the corresponding implementation in Ada:

[Ada]

Listing 75: values_processing.ads

```ada
package Values_Processing is

  type Integer_Array is array (Positive range <>) of Integer;

  type Process_One_Callback is not null access
  function (Value : Integer) return Integer;

  procedure Process_Values (Values : in out Integer_Array;
    Process_One : Process_One_Callback);

end Values_Processing;
```

Listing 76: values_processing.adb

```ada
package body Values_Processing is

  procedure Process_Values (Values : in out Integer_Array;
    Process_One : Process_One_Callback) is
    (continues on next page)
```
begin
  for I in Values'Range loop
    Values (I) := Process_One (Values (I));
  end loop;
end Process_Values;
end Values_Processing;

Listing 77: proc_10.ads

function Proc_10 (Value : Integer) return Integer;

Listing 78: proc_10.adb

function Proc_10 (Value : Integer) return Integer is
begin
  return Value + 10;
end Proc_10;

Listing 79: show_callback.adb

with Ada.Text_IO; use Ada.Text_IO;
with Values_Processing; use Values_Processing;
with Proc_10;

procedure Show_Callback is
  Values : Integer_Array := (1, 2, 3, 4, 5);
begin
  Process_Values (Values, Proc_10'Access);
  for I in Values'Range loop
    Put_Line ("Value [" & Positive'Image (I) & "] = 
      & Integer'Image (Values (I)))
  end loop;
end Show_Callback;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Callback_Ada
MD5: f49c54f0d14193d305c0e962a392ab67

Runtime output

Value [ 1] = 11
Value [ 2] = 12
Value [ 3] = 13
Value [ 4] = 14
Value [ 5] = 15

Similar to the implementation in C, the Process_Values procedure receives the access to a callback routine, which is then called for each value of the Values array.

Note that the declaration of Process_One_Callback makes use of the not null access declaration. By using this approach, we ensure that any parameter of this type has a valid value, so we can always call the callback routine.
72.4 Design by components using dynamic libraries

In the previous sections, we have shown how to use packages to create separate components of a system. As we know, when designing a complex system, it is advisable to separate concerns into distinct units, so we can use Ada packages to represent each unit of a system. In this section, we go one step further and create separate dynamic libraries for each component, which we'll then link to the main application.

Let's suppose we have a main system (Main_System) and a component A (Component_A) that we want to use in the main system. For example:

[Ada]

Listing 80: component_a.ads

```ada
-- File: component_a.ads
package Component_A is
  type Float_Array is array (Positive range <>) of Float;
  function Average (Data : Float_Array) return Float;
end Component_A;
```

Listing 81: component_a.adb

```ada
-- File: component_a.adb
package body Component_A is
  function Average (Data : Float_Array) return Float is
    Total : Float := 0.0;
    begin
      for Value of Data loop
        Total := Total + Value;
      end loop;
      return Total / Float (Data'Length);
    end Average;
end Component_A;
```

Listing 82: main_system.adb

```ada
-- File: main_system.adb
with Ada.Text_IO; use Ada.Text_IO;
with Component_A; use Component_A;
procedure Main_System is
  Values : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
  Average_Value : Float;
  begin
    Average_Value := Average (Values);
    Put_Line ("Average = " & Float'Image (Average_Value));
  end Main_System;
```

Code block metadata
Note that, in the source-code example above, we're indicating the name of each file. We'll now see how to organize those files in a structure that is suitable for the GNAT build system (GPRbuild).

In order to discuss how to create dynamic libraries, we need to dig into some details about the build system. With GNAT, we can use project files for GPRbuild to easily design dynamic libraries. Let's say we use the following directory structure for the code above:

```
|- component_a
  |- component_a.gpr
  |- src
    | component_a.adb
    | component_a.ads
|- main_system
  |- main_system.gpr
  |- src
    | main_system.adb
```

Here, we have two directories: component_a and main_system. Each directory contains a project file (with the .gpr file extension) and a source-code directory (src).

In the source-code example above, we've seen the content of files component_a.ads, component_a.adb and main_system.adb. Now, let's discuss how to write the project file for Component_A (component_a.gpr), which will build the dynamic library for this component:

```ada
library project Component_A is
  for Source_Dirs use "src";
  for Object_Dir use "obj"
  for Create_Missing_Dirs use "True"
  for Library_Name use "component_a"
  for Library_Kind use "dynamic"
  for Library_Dir use "lib"
end Component_A;
```

The project is defined as a library project instead of project. This tells GPRbuild to build a library instead of an executable binary. We then specify the library name using the Library_Name attribute, which is required, so it must appear in a library project. The next two library-related attributes are optional, but important for our use-case. We use:

- **Library_Kind** to specify that we want to create a dynamic library — by default, this attribute is set to static;
- **Library_Dir** to specify the directory where the library is stored.

In the project file of our main system (main_system.gpr), we just need to reference the project of Component_A using a with clause and indicating the correct path to that project file:

```ada
with "../component_a/component_a.gpr";

project Main_System is
  for Source_Dirs use "src"
  for Object_Dir use "obj"
```

(continues on next page)
for Create_Missing_Dirs use "True";
for Main use ("main_system.adb");
end Main_System;

**GPRbuild** takes care of selecting the correct settings to link the dynamic library created for Component_A with the main application (Main_System) and build an executable.

We can use the same strategy to create a Component_B and dynamically link to it in the Main_System. We just need to create the separate structure for this component — with the appropriate Ada packages and project file — and include it in the project file of the main system using a *with* clause:

```ada
with "../component_a/component_a.gpr";
with "../component_b/component_b.gpr";
...
```

Again, **GPRbuild** takes care of selecting the correct settings to link both dynamic libraries together with the main application.

You can find more details and special setting for library projects in the [GPRbuild documentation](https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/gnat_project_manager.html#library-projects)334.

### In the GNAT toolchain

The GNAT toolchain includes a more advanced example focusing on how to load dynamic libraries at runtime. You can find it in the share/examples/gnat/plugins directory of the GNAT toolchain installation. As described in the README file from that directory, this example "comprises a main program which probes regularly for the existence of shared libraries in a known location. If such libraries are present, it uses them to implement features initially not present in the main program."

---

334 [https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/gnat_project_manager.html#library-projects](https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/gnat_project_manager.html#library-projects)
73.1 Overall expectations

All in all, there should not be significant performance differences between code written in Ada and code written in C, provided that they are semantically equivalent. Taking the current GNAT implementation and its GCC C counterpart for example, most of the code generation and optimization phases are shared between C and Ada — so there’s not one compiler more efficient than the other. Furthermore, the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. They should be equivalent on average.

When comparing the performance of C and Ada code, differences might be observed. This usually comes from the fact that, while the two piece appear semantically equivalent, they happen to be actually quite different; C code semantics do not implicitly apply the same run-time checks that Ada does. This section will present common ways for improving Ada code performance.

73.2 Switches and optimizations

Clever use of compilation switches might optimize the performance of an application significantly. In this section, we'll briefly look into some of the switches available in the GNAT toolchain.

73.2.1 Optimizations levels

Optimization levels can be found in many compilers for multiple languages. On the lowest level, the GNAT compiler doesn’t optimize the code at all, while at the higher levels, the compiler analyses the code and optimizes it by removing unnecessary operations and making the most use of the target processor’s capabilities.

By being part of GCC, GNAT offers the same -O_ switches as GCC:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-00</td>
<td>No optimization: the generated code is completely unoptimized. This is the default optimization level.</td>
</tr>
<tr>
<td>-01</td>
<td>Moderate optimization.</td>
</tr>
<tr>
<td>-02</td>
<td>Full optimization.</td>
</tr>
<tr>
<td>-03</td>
<td>Same optimization level as for -02. In addition, further optimization strategies, such as aggressive automatic inlining and vectorization.</td>
</tr>
</tbody>
</table>
Learning Ada

Note that the higher the level, the longer the compilation time. For fast compilation during development phase, unless you’re working on benchmarking algorithms, using -00 is probably a good idea.

In addition to the levels presented above, GNAT also has the -Os switch, which allows for optimizing code and data usage.

### 73.2.2 Inlining

As we’ve seen in the previous section, automatic inlining depends on the optimization level. The highest optimization level (-O3), for example, performs aggressive automatic inlining. This could mean that this level inlines too much rather than not enough. As a result, the cache may become an issue and the overall performance may be worse than the one we would achieve by compiling the same code with optimization level 2 (-02). Therefore, the general recommendation is to not just select -03 for the optimized version of an application, but instead compare it the optimized version built with -02.

In some cases, it's better to reduce the optimization level and perform manual inlining instead of automatic inlining. We do that by using the Inline aspect. Let’s reuse an example from a previous chapter and inline the Average function:

[Ada]

```
Listing 1: float_arrays.ads
package Float_Arrays is
  type Float_Array is array (Positive range <>) of Float;
  function Average (Data : Float_Array) return Float
    with Inline;
end Float_Arrays;
```

```
Listing 2: float_arrays.adb
package body Float_Arrays is
  function Average (Data : Float_Array) return Float is
    Total : Float := 0.0;
    begin
      for Value of Data loop
        Total := Total + Value;
      end loop;
      return Total / Float (Data'Length);
    end Average;
end Float_Arrays;
```

```
Listing 3: compute_average.adb
with Ada.Text_IO; use Ada.Text_IO;
with Float_Arrays; use Float_Arrays;
procedure Compute_Average is
  Values : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
  Average_Value : Float;
  begin
    Average_Value := Average (Values);
  end Compute_Average;
```
Put_Line ("Average = " & Float'Image (Average_Value));
end Compute_Average;

**Code block metadata**

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Inlining
MD5: faf9d0d8cd5aef7d7a48bcd950b1256fa

**Runtime output**

Average = 1.15000E+01

When compiling this example, GNAT will inline Average in the Compute_Average procedure. In order to effectively use this aspect, however, we need to set the optimization level to at least -O1 and use the -gnatn switch, which instructs the compiler to take the Inline aspect into account.

Note, however, that the Inline aspect is just a *recommendation* to the compiler. Sometimes, the compiler might not be able to follow this recommendation, so it won't inline the subprogram. In this case, we get a compilation warning from GNAT.

These are some examples of situations where the compiler might not be able to inline a subprogram:

- when the code is too large,
- when it's too complicated — for example, when it involves exception handling —, or
- when it contains tasks, etc.

In addition to the Inline aspect, we also have the Inline_Always aspect. In contrast to the former aspect, however, the Inline_Always aspect isn't primarily related to performance. Instead, it should be used when the functionality would be incorrect if inlining was not performed by the compiler. Examples of this are procedures that insert Assembly instructions that only make sense when the procedure is inlined, such as memory barriers.

Similar to the Inline aspect, there might be situations where a subprogram has the Inline_Always aspect, but the compiler is unable to inline it. In this case, we get a compilation error from GNAT.

### 73.3 Checks and assertions

#### 73.3.1 Checks

Ada provides many runtime checks to ensure that the implementation is working as expected. For example, when accessing an array, we would like to make sure that we're not accessing a memory position that is not allocated for that array. This is achieved by an index check.

Another example of runtime check is the verification of valid ranges. For example, when adding two integer numbers, we would like to ensure that the result is still in the valid range — that the value is neither too large nor too small. This is achieved by a range check. Likewise, arithmetic operations shouldn't overflow or underflow. This is achieved by an overflow check.

Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots. For example, checking the index of an array in a sorting algorithm may significantly decrease its performance. In
Learning Ada

those cases, suppressing the check may be an option. We can achieve this suppression by using `pragma Suppress` (Index_Check). For example:

```ada
procedure Sort (A : in out Integer_Array) is
   pragma Suppress (Index_Check);
begin
   -- (implementation removed...)
   null;
end Sort;
```

In case of overflow checks, we can use `pragma Suppress` (Overflow_Check) to suppress them:

```ada
function Some_Computation (A, B : Int32) return Int32 is
   pragma Suppress (Overflow_Check);
begin
   -- (implementation removed...)
   null;
end Sort;
```

We can also deactivate overflow checks for integer types using the `-gnato` switch when compiling a source-code file with GNAT. In this case, overflow checks in the whole file are deactivated.

It is also possible to suppress all checks at once using `pragma Suppress` (All_Checks). In addition, GNAT offers a compilation switch called `-gnatp`, which has the same effect on the whole file.

Note, however, that this kind of suppression is just a recommendation to the compiler. There's no guarantee that the compiler will actually suppress any of the checks because the compiler may not be able to do so — typically because the hardware happens to do it. For example, if the machine traps on any access via address zero, requesting the removal of null access value checks in the generated code won't prevent the checks from happening.

It is important to differentiate between required and redundant checks. Let's consider the following example in C:

```c
Listing 4: main.c

#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 8, b = 0, res;

    res = a / b;

    // printing the result
    printf("res = %d\n", res);

    return 0;
}
```

Because C doesn't have language-defined checks, as soon as the application tries to divide a value by zero in `res = a / b`, it'll break — on Linux, for example, you may get the...
following error message by the operating system: Floating point exception (core dumped). Therefore, we need to manually introduce a check for zero before this operation. For example:

[C]

Listing 5: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 8, b = 0, res;
    if (b != 0) {
        res = a / b;
        // printing the result
        printf("res = %d\n", res);
    } else {
        // printing error message
        printf("Error: cannot calculate value (division by zero)\n");
    }
    return 0;
}
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Division_By_Zero_Check_C
MD5: 67ea0140d8248674b4aac06825c7c0be

Runtime output

Error: cannot calculate value (division by zero)

This is the corresponding code in Ada:

[Ada]

Listing 6: show_division_by_zero.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Division_By_Zero is
    A : Integer := 8;
    B : Integer := 0;
    Res : Integer;
begin
    Res := A / B;
    Put_Line ("Res = " & Integer'Image (Res));
end Show_Division_By_Zero;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Division_By_Zero_Ada
MD5: 2af6690eb977203ef7ce2178d15255af

Build output
Learning Ada

Similar to the first version of the C code, we're not explicitly checking for a potential division by zero here. In Ada, however, this check is automatically inserted by the language itself. When running the application above, an exception is raised when the application tries to divide the value in A by zero. We could introduce exception handling in our example, so that we get the same message as we did in the second version of the C code:

[Ada]

Listing 7: show_division_by_zero.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Division_By_Zero is
   A : Integer := 8;
   B : Integer := 0;
   Res : Integer;
begin
   Res := A / B;
   Put_Line ("Res = " & Integer'Image (Res));
exception
   when Constraint_Error =>
      Put_Line ("Error: cannot calculate value (division by zero)");
   when others => null;
end Show_Division_By_Zero;
```

This example demonstrates that the division check for Res := A / B is required and shouldn't be suppressed. In contrast, a check is redundant — and therefore not required — when we know that the condition that leads to a failure can never happen. In many cases, the compiler itself detects redundant checks and eliminates them (for higher optimization levels). Therefore, when improving the performance of your application, you should:

1. keep all checks active for most parts of the application;
2. identify the hot-spots of your application;
3. identify which checks haven't been eliminated by the optimizer on these hot-spots;
4. identify which of those checks are redundant;
5. only suppress those checks that are redundant, and keep the required ones.

### 73.3.2 Assertions

We’ve already discussed assertions in *this section of the SPARK chapter* (page 1462). Assertions are user-defined checks that you can add to your code using the `pragma Assert`. For example:

```ada
function Some_Computation (A, B : Int32) return Int32 is
    Res : Int32;
begin
    -- (implementation removed...)
    pragma Assert (Res >= 0);
    return Res;
end Sort;
```

Assertions that are specified with `pragma Assert` are not enabled by default. You can enable them by setting the assertion policy to `check` — using `pragma Assertion_Policy` (Check) — or by using the `-gnata` switch when compiling with GNAT.

Similar to the checks discussed previously, assertions can generate significant overhead when used at hot-spots. Restricting those assertions to development (e.g. debug version) and turning them off on the release version may be an option. In this case, formal proof — as discussed in the *SPARK chapter* (page 1455) — can help you. By formally proving that assertions will never fail at run-time, you can safely deactivate them.

### 73.4 Dynamic vs. static structures

Ada generally speaking provides more ways than C or C++ to write simple dynamic structures, that is to say structures that have constraints computed after variables. For example, it’s quite typical to have initial values in record types:

```ada
type R is record
    F : Some_Field := Call_To_Some_Function;
end record;
```

However, the consequences of the above is that any declaration of a instance of this type without an explicit value for F will issue a call to `Call_To_Some_Function`. More subtle issue may arise with elaboration. For example, it's possible to write:

```ada
package Some_Functions is
    function Some_Function_Call return Integer is (2);
    function Some_Other_Function_Call return Integer is (10);
end Some_Functions;
```
Learning Ada

Listing 9: values.ads

```ada
with Some_Functions; use Some_Functions;

package Values is
  A_Start : Integer := Some_Function_Call;
  A_End : Integer := Some_Other_Function_Call;
end Values;
```

Listing 10: arr_def.ads

```ada
with Values; use Values;

package Arr_Def is
  type Arr is array (Integer range A_Start .. A_End) of Integer;
end Arr_Def;
```

MD5: 0c97cecb64d27e935724c8b5f941fb4f

It may indeed be appealing to be able to change the values of A_Start and A_End at startup so as to align a series of arrays dynamically. The consequence, however, is that these values will not be known statically, so any code that needs to access to boundaries of the array will need to read data from memory. While it's perfectly fine most of the time, there may be situations where performances are so critical that static values for array boundaries must be enforced.

Here's a last case which may also be surprising:

[Ada]

Listing 11: arr_def.ads

```ada
package Arr_Def is
  type Arr is array (Integer range <>) of Integer;
  type R (D1, D2 : Integer) is record
    F1 : Arr (1 .. D1);
    F2 : Arr (1 .. D2);
  end record;
end Arr_Def;
```

Project: Courses.Ada_For_EMBEDD_C_Dev.Performance.Record_With_Arrays
MD5: e7b2656433279d36db87506276b68398

In the code above, R contains two arrays, F1 and F2, respectively constrained by the discriminant D1 and D2. The consequence is, however, that to access F2, the run-time needs to know how large F1 is, which is dynamically constrained when creating an instance. Therefore, accessing to F2 requires a computation involving D1 which is slower than, let's say, two pointers in an C array that would point to two different arrays.

Generally speaking, when values are used in data structures, it's useful to always consider where they're coming from, and if their value is static (computed by the compiler) or dynamic (only known at run-time). There's nothing fundamentally wrong with dynamically constrained types, unless they appear in performance-critical pieces of the application.
73.5 Pointers vs. data copies

In the section about pointers (page 1483), we mentioned that the Ada compiler will automatically pass parameters by reference when needed. Let's look into what "when needed" means. The fundamental point to understand is that the parameter types determine how the parameters are passed in and/or out. The parameter modes do not control how parameters are passed.

Specifically, the language standards specifies that scalar types are always passed by value, and that some other types are always passed by reference. It would not make sense to make a copy of a task when passing it as a parameter, for example. So parameters that can be passed reasonably by value will be, and those that must be passed by reference will be. That's the safest approach.

But the language also specifies that when the parameter is an array type or a record type, and the record/array components are all by-value types, then the compiler decides: it can pass the parameter using either mechanism. The critical case is when such a parameter is large, e.g., a large matrix. We don't want the compiler to pass it by value because that would entail a large copy, and indeed the compiler will not do so. But if the array or record parameter is small, say the same size as an address, then it doesn't matter how it is passed and by copy is just as fast as by reference. That's why the language gives the choice to the compiler. Although the language does not mandate that large parameters be passed by reference, any reasonable compiler will do the right thing.

The modes do have an effect, but not in determining how the parameters are passed. Their effect, for parameters passed by value, is to determine how many times the value is copied. For mode in and mode out there is just one copy. For mode in out there will be two copies, one in each direction.

Therefore, unlike C, you don't have to use access types in Ada to get better performance when passing arrays or records to subprograms. The compiler will almost certainly do the right thing for you.

Let's look at this example:

[Listing 12: main.c]

```c
#include <stdio.h>

struct Data {
  int prev, curr;
};

void update(struct Data *d, int v) {
  d->prev = d->curr;
  d->curr = v;
}

void display(const struct Data *d) {
  printf("Prev : %d\n", d->prev);
  printf("Curr : %d\n", d->curr);
}

int main(int argc, const char * argv[]) {
  struct Data D1 = { 0, 1 };
}
```
(continues on next page)
In this C code example, we're using pointers to pass D1 as a reference to update and display. In contrast, the equivalent code in Ada simply uses the parameter modes to specify the data flow directions. The mechanisms used to pass the values do not appear in the source code.

[Ada]

Listing 13: update_record.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Update_Record is

  type Data is record
    Prev : Integer;
    Curr : Integer;
  end record;

  procedure Update (D : in out Data; V : Integer) is
  begin
    D.Prev := D.Curr;
    D.Curr := V;
    end Update;

  procedure Display (D : Data) is
  begin
    Put_Line ("Prev: " & Integer'Image (D.Prev));
    Put_Line ("Curr: " & Integer'Image (D.Curr));
    end Display;

  begin
    D1 : Data := (0, 1);
    Update (D1, 3);
    Display (D1);
    end Update_Record;
```
In the calls to Update and Display, D1 is always be passed by reference. Because no extra copy takes place, we get a performance that is equivalent to the C version. If we had used arrays in the example above, D1 would have been passed by reference as well:

[Ada]

Listing 14: update_array.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Update_Array is

  type Data_State is (Prev, Curr);
  type Data is array (Data_State) of Integer;

  procedure Update (D : in out Data;
                   V : Integer) is
  begin
    D (Prev) := D (Curr);
    D (Curr) := V;
  end Update;

  procedure Display (D : Data) is
  begin
    Put_Line ("Prev: " & Integer'Image (D (Prev)));
    Put_Line ("Curr: " & Integer'Image (D (Curr)));
  end Display;

begin
  D1 : Data := (0, 1);

  begin
    Update (D1, 3);
    Display (D1);
  end Update_Array;
```

Again, no extra copy is performed in the calls to Update and Display, which gives us optimal performance when dealing with arrays and avoids the need to use access types to optimize the code.
73.5.1 Function returns

Previously, we've discussed the cost of passing complex records as arguments to subprograms. We've seen that we don't have to use explicit access type parameters to get better performance in Ada. In this section, we'll briefly discuss the cost of function returns.

In general, we can use either procedures or functions to initialize a data structure. Let's look at this example in C:

[C]

Listing 15: main.c

```
#include <stdio.h>

struct Data {
    int prev, curr;
};

void init_data(struct Data *d)
{
    d->prev = 0;
    d->curr = 1;
}

struct Data get_init_data()
{
    struct Data d = { 0, 1 };
    return d;
}

int main(int argc, const char * argv[])
{
    struct Data D1;
    D1 = get_init_data();
    init_data(&D1);
    return 0;
}
```

This code example contains two subprograms that initialize the Data structure:

- `init_data()`, which receives the data structure as a reference (using a pointer) and initializes it, and
- `get_init_data()`, which returns the initialized structure.

In C, we generally avoid implementing functions such as `get_init_data()` because of the extra copy that is needed for the function return.

This is the corresponding implementation in Ada:

[Ada]
procedure Init_Record is
  type Data is record
    Prev : Integer;
    Curr : Integer;
  end record;

procedure Init (D : out Data) is
begin
  D := (Prev => 0, Curr => 1);
end Init;

function Init return Data is
  D : constant Data := (Prev => 0, Curr => 1);
begin
  return D;
end Init;

D1 : Data;
pragma Unreferenced (D1);
begin
  D1 := Init;
  Init (D1);
end Init_Record;

Code block metadata
Project: Courses.Ada_For_Embedded_C_DEV.Performance.Init_Rec_Proc_And_Func_Ada
MD5: 0f930eea432a82d78840b72c0714b283

Build output
init_record.adb:25:10: warning: pragma Unreferenced given for "D1" [enabled by default]

In this example, we have two versions of Init: one using a procedural form, and the other one using a functional form. Note that, because of Ada's support for subprogram overloading, we can use the same name for both subprograms.

The issue is that assignment of a function result entails a copy, just as if we assigned one variable to another. For example, when assigning a function result to a constant, the function result is copied into the memory for the constant. That's what is happening in the above examples for the initialized variables.

Therefore, in terms of performance, the same recommendations apply: for large types we should avoid writing functions like the Init function above. Instead, we should use the procedural form of Init. The reason is that the compiler necessarily generates a copy for the Init function, while the Init procedure uses a reference for the output parameter, so that the actual record initialization is performed in place in the caller's argument.

An exception to this is when we use functions returning values of limited types, which by definition do not allow assignment. Here, to avoid allowing something that would otherwise look suspiciously like an assignment, the compiler generates the function body so that it builds the result directly into the object being assigned. No copy takes place.

We could, for example, rewrite the example above using limited types:

[Ada]
In this example, \( D1 : Data := Init; \) has the same cost as the call to the procedural form — \( \text{Init} \ (D1); \) — that we've seen in the previous example. This is because the assignment is done in place.

Note that limited types require the use of the extended return statements (\textit{return} \ldots \textit{do} \ldots \textit{end return}) in function implementations. Also note that, because the Data type is limited, we can only use the Init function in the declaration of \( D1; \) a statement in the code such as \( D1 := \text{Init}; \) is therefore forbidden.
ARGUMENTATION AND BUSINESS PERSPECTIVES

The technical benefits of a migration from C to Ada are usually relatively straightforward to demonstrate. Hopefully, this course provides a good basis for it. However, when faced with an actual business decision to make, additional considerations need to be taken into account, such as return on investment, perennity of the solution, tool support, etc. This section will cover a number of usual questions and provide elements of answers.

74.1 What's the expected ROI of a C to Ada transition?

Switching from one technology to another is a cost, may that be in terms of training, transition of the existing environment or acquisition of new tools. This investment needs to be matched with an expected return on investment, or ROI, to be consistent. Of course, it's incredibly difficult to provide a firm answer to how much money can be saved by transitioning, as this is highly dependent on specific project objectives and constraints. We're going to provide qualitative and quantitative arguments here, from the perspective of a project that has to reach a relatively high level of integrity, that is to say a system where the occurrence of a software failure is a relatively costly event.

From a qualitative standpoint, there are various times in the software development life cycle where defects can be found:

1. on the developer's desk
2. during component testing
3. during integration testing
4. after deployment
5. during maintenance

Numbers from studies vary greatly on the relative costs of defects found at each of these phases, but there's a clear ordering between them. For example, a defect found while developing is orders of magnitude less expensive to fix than a defect found e.g. at integration time, which may involve costly debugging sessions and slow down the entire system acceptance. The whole purpose of Ada and SPARK is to push defect detection to the developer's desk as much as possible; at least for all of these defects that can be identified at that level. While the strict act of writing software may be taking more effort because of all of the additional safeguards, this should have a significant and positive impact down the line and help to control costs overall. The exact value this may translate into is highly business dependent.

From a quantitative standpoint, two studies have been done almost 25 years apart and provide similar insights:

- Rational Software in 1995 found that the cost of developing software in Ada was overall half as much as the cost of developing software in C.
Learning Ada

- VDC ran a study in 2018, finding that the cost savings of developing with Ada over C ranged from 6% to 38% in savings.

From a qualitative standpoint, in particular with regards to Ada and C from a formal proof perspective, an interesting presentation was made in 2017 by two researchers. They tried to apply formal proof on the same piece of code, developed in Ada/SPARK on one end and C/Frama-C on the other. Their results indicate that the Ada/SPARK technology is indeed more conducive to formal proof methodologies.

Although all of these studies have their own biases, they provide a good idea of what to expect in terms of savings once the initial investment in switching to Ada is made. This is assuming everything else is equal, in particular that the level of integrity is the same. In many situations, the migration to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will rise (it's more expensive to develop better software) and Ada is viewed as a means to mitigate this rise in development costs.

That being said, the point of this argument is not to say that it's not possible to write very safe and secure software with languages different than Ada. With the right expertise, the right processes and the right tools, it's done every day. The point is that Ada overall reduces the level of processes, expertise and tools necessary and will allow to reach the same target at a lower cost.

### 74.2 Who is using Ada today?

Ada was initially born as a DoD project, and thus got its initial customer base in aerospace and defence (A&D). At the time these lines are written and from the perspective of AdaCore, A&D is still the largest consumer of Ada today and covers about 70% of the market. This creates a consistent and long lasting set of established users as these project last often for decades, using the same codebase migrating from platform to platform.

More recently however, there has been an emerging interest for Ada in new communities of users such as automotive, medical device, industrial automation and overall cyber-security. This can probably be explained by a rise of safety, reliability and cyber-security requirements. The market is moving relatively rapidly today and we're anticipating an increase of the Ada footprint in these domains, while still remaining a technology of choice for the development of mission critical software.

### 74.3 What is the future of the Ada technology?

The first piece of the answer lies in the user base of the Ada language, as seen in the previous question. Projects using Ada in the aerospace and defence domain maintain source code over decades, providing healthy funding foundation for Ada-based technologies.

AdaCore being the author of this course, it's difficult for us to be fair in our description of other Ada compilation technologies. We will leave to the readers the responsibility of forging their own opinion. If they present a credible alternative to the GNAT compiler, then this whole section can be considered as void.

Assuming GNAT is the only option available, and acknowledging that this is an argument that we're hearing from a number of Ada adopters, let's discuss the "sole source" issue.

First of all, it's worth noting that industries are using a lot of software that is provided by only one source, so while non-ideal, these situations are also quite common.

In the case of the GNAT compiler however, while AdaCore is the main maintainer, this maintenance is done as part of an open-source community. This means that nothing prevents
a third party to start selling a competing set of products based on the same compiler, pro-
vided that it too adopts the open-source approach. Our job is to be more cost-effective
than the alternative, and indeed for the vast part this has prevented a competing offering
to emerge. However, should AdaCore disappear or switch focus, Ada users would not be
prevented from carrying on using its software (there is no lock) and a third party could take
over maintenance. This is not a theoretical case, this has been done in the past either by
companies looking at supporting their own version of GNAT, vendors occupying a specific
niche that was left uncovered, or hobbyists developing their own builds.

With that in mind, it's clear that the "sole source" provider issue is a circumstantial —
nothing is preventing other vendors from emerging if the conditions are met.

74.4 Is the Ada toolset complete?

A language by itself is of little use for the development of safety-critical software. Instead,
a complete toolset is needed to accompany the development process, in particular tools
for edition, testing, static analysis, etc.

AdaCore provides a number of these tools either in through its core or add-on package.
These include (as of 2019):

- An IDE (GNAT Studio)
- An Eclipse plug-in (GNATbench)
- A debugger (GDB)
- A testing tool (GNATtest)
- A structural code coverage tool (GNATcoverage)
- A metric computation tool (GNATmetric)
- A coding standard checker (GNATcheck)
- Static analysis tools (CodePeer, SPARK Pro)
- A Simulink code generator (QGen)
- An Ada parser to develop custom tools (libadalang)

Ada is, however, an internationally standardized language, and many companies are pro-
viding third party solutions to complete the toolset. Overall, the language can be and is
used with tools on par with their equivalent C counterparts.

74.5 Where can I find Ada or SPARK developers?

A common question from teams on the verge of selecting Ada and SPARK is how to manage
the developer team growth and turnover. While Ada and SPARK are taught by a growing
number of universities worldwide, it may still be challenging to hire new staff with prior Ada
experience.

Fortunately, Ada’s base semantics are very close to those of C/C++, so that a good embed-
ded software developer should be able to learn it relatively easily. This course is definitely
a resource available to get started. Online training material is also available, together with
on-site in person training.

In general, getting an engineer operational in Ada and SPARK shouldn't take more than a
few weeks worth of time.
74.6 How to introduce Ada and SPARK in an existing code base?

The most common scenario when introducing Ada and SPARK to a project or a team is to do it within a pre-existing C codebase, which can already spread over hundreds of thousands if not millions lines of code. Re-writing this software to Ada or SPARK is of course not practical and counterproductive.

Most teams select either a small piece of existing code which deserves particular attention, or new modules to develop, and concentrate on this. Developing this module or part of the application will also help in developing the coding patterns to be used for the particular project and company. This typically concentrates an effort of a few people on a few thousands lines of code. The resulting code can be linked to the rest of the C application. From there, the newly established practices and their benefit can slowly spread through the rest of the environment.

Establishing this initial core in Ada and SPARK is critical, and while learning the language isn’t a particularly difficult task, applying it to its full capacity may require some expertise. One possibility to accelerate this initial process is to use AdaCore mentorship services.
Although Ada's syntax might seem peculiar to C developers at first glance, it was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner — as it is often the case in C.

Especially in the embedded domain, C developers are used to working at a very low level, which includes mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations because it was designed to replace Assembly language for faster, more efficient programming.

Ada can be used to describe high level semantics and architectures. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded Assembly code or bit-level data management. However, although Ada supports bitwise operations such as masks and shifts, they should be relatively rarely needed. When translating C code to Ada, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other. For example, we can interpret an arbitrary data type as a bit-field and perform low-level operations on it.

Because Ada is a strongly typed language, it doesn't define any implicit type conversions like C. If we try to compile Ada code that contains type mismatches, we'll get a compilation error. Because the compiler prevents mixing variables of different types without explicit type conversion, we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect. In this sense, Ada's type system encourages programmers to think about data at a high level of abstraction. Ada supports overlays and unchecked conversions as a way of converting between unrelated data type, which are typically used for interfacing with low-level elements such as registers.

In Ada, arrays aren't interchangeable with operations on pointers like in C. Also, array types are considered first-class citizens and have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and we can specify both the starting and ending bounds. In addition, Ada offers high-level operations for copying, slicing, and assigning values to arrays.

Although Ada supports pointers, most situations that would require a pointer in C do not in Ada. In the vast majority of the cases, indirect memory management can be hidden from the developer and thus prevent many potential errors. In C, pointers are typically used to pass references to subprograms, for example. In contrast, Ada parameter modes indicate the flow of information to the reader, leaving the means of passing that information to the compiler.

When translating pointers from C code to Ada, we need to assess whether they are needed in the first place. Ada pointers (access types) should only be used with complex structures that cannot be allocated at run-time. There are many situations that would require a pointer in C, but do not in Ada. For example, arrays — even when dynamically allocated —, results of functions, passing of large structures as parameters, access to registers, etc.
Because of the absence of namespaces, global names in C tend to be very long. Also, because of the absence of overloading, they can even encode type names in their name. In Ada, a package is a namespace. Also, we can use the private part of a package to declare private types and private subprograms. In fact, private types are useful for preventing the users of those types from depending on the implementation details. Another use-case is the prevention of package users from accessing the package state/data arbitrarily.

Ada has a dedicated set of features for interfacing with other languages, so we can easily interface with our existing C code before translating it to Ada. Also, GNAT includes automatic binding generators. Therefore, instead of re-writing the entire C code upfront, which isn't practical or cost-effective, we can selectively translate modules from C to Ada.

When it comes to implementing concurrency and real time, Ada offers several options. Ada provides high level constructs such as tasks and protected objects to express concurrency and synchronization, which can be used when running on top of an operating system such as Linux. On more constrained systems, such as bare metal or some real-time operating systems, a subset of the Ada tasking capabilities — known as the Ravenscar and Jorvik profiles — is available. Though restricted, this subset also has nice properties, in particular the absence of deadlock, the absence of priority inversion, schedulability and very small footprint. On bare metal systems, this also essentially means that Ada comes with its own real-time kernel. The advantage of using the full Ada tasking model or the restricted profiles is to enhance portability.

Ada includes many features typically used for embedded programming:

- Built-in support for handling interrupts, so we can process interrupts by attaching a handler — as a protected procedure — to it.
- Built-in support for handling both volatile and atomic data.
- Support for register overlays, which we can use to create a structure that facilitates manipulating bits from registers.
- Support for creating data streams for serialization of arbitrary information and transmission over a communication channel, such as a serial port.
- Built-in support for fixed-point arithmetic, which is an option when our target device doesn't have a floating-point unit or the result of calculations needs to be bit-exact.

Also, Ada compilers such as GNAT have built-in support for directly mixing Ada and Assembly code.

Ada also supports contracts, which can be associated with types and variables to refine values and define valid and invalid values. The most common kind of contract is a range constraint — using the range reserved word. Ada also supports contract-based programming in the form of preconditions and postconditions. One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations.

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. In C, variability is usually achieved through macros and function pointers, the former being tied to static variability and the latter to dynamic variability. Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. Examples of static variability in Ada are: genericity, simple derivation, configuration pragma files, and configuration packages. Examples of dynamic variability in Ada are: records with discriminants, variant records — which may include the use of unions —, object orientation, pointers to subprograms, and design by components using dynamic libraries.

There shouldn't be significant performance differences between code written in Ada and code written in C — provided that they are semantically equivalent. One reason is that the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. Therefore, they should be equivalent on average. However, when a piece of code in Ada is significantly slower than its counterpart in C, this usually comes from the fact that, while the two pieces of code appear
to be semantically equivalent, they happen to be actually quite different. Fortunately, there are strategies that we can use to improve the performance and make it equivalent to the C version. These are some examples:

- Clever use of compilation switches, which might optimize the performance of an application significantly.
- Suppression of checks at specific parts of the implementation.
  - Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots.
- Restriction of assertions to development code.
  - For example, we may use assertions in the debug version of the code and turn them off in the release version.
  - Also, we may use formal proof to decide which assertions we turn off in the release version. By formally proving that assertions will never fail at run-time, we can safely deactivate them.

Formal proof — a form of static analysis — can give strong guarantees about checks, for all possible conditions and all possible inputs. It verifies conditions prior to execution, even prior to compilation, so we can remove bugs earlier in the development phase. This is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after deployment. Preventing bug introduction into the deployed system is the least expensive approach of all.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the gnatprove verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof.

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering exceptions that interrupt the normal flow of execution. However, we may be able to prove that the language-defined checks won't raise exceptions at run-time. This is known as proving Absence of Run-Time Errors. Successful proof of these checks is highly significant in itself. One of the major resulting benefits is that we can deploy the final executable with checks disabled.

In many situations, the migration of C code to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will raise. However, Ada is a more expressive, powerful language, designed to reduce errors earlier in the life-cycle, thus reducing costs. Therefore, Ada makes it possible to write very safe and secure software at a lower cost than languages such as C.
APPENDIX A: HANDS-ON OBJECT-ORIENTED PROGRAMMING

The goal of this appendix is to present a hands-on view on how to translate a system from C to Ada and improve it with object-oriented programming.

76.1 System Overview

Let's start with an overview of a simple system that we'll implement and use below. The main system is called AB and it combines two systems A and B. System AB is not supposed to do anything useful. However, it can serve as a good model for the hands-on we're about to start.

This is a list of requirements for the individual systems A and B, and the combined system AB:

- **System A:**
  - The system can be activated and deactivated.
    - During activation, the system's values are reset.
  - Its current value (in floating-point) can be retrieved.
    - This value is the average of the two internal floating-point values.
  - Its current state (activated or deactivated) can be retrieved.

- **System B:**
  - The system can be activated and deactivated.
    - During activation, the system's value is reset.
  - Its current value (in floating-point) can be retrieved.
  - Its current state (activated or deactivated) can be retrieved.

- **System AB**
  - The system contains an instance of system A and an instance of system B.
  - The system can be activated and deactivated.
    - System AB activates both systems A and B during its own activation.
    - System AB deactivates both systems A and B during its own deactivation.
  - Its current value (in floating-point) can be retrieved.
    - This value is the average of the current values of systems A and B.
  - Its current state (activated or deactivated) can be retrieved.
    - AB is only considered activated when both systems A and B are activated.
Learning Ada

- The system's health can be checked.

  * This check consists in calculating the absolute difference \( D \) between the current values of systems \( A \) and \( B \) and checking whether \( D \) is below a threshold of 0.1.

The source-code in the following section contains an implementation of these requirements.

### 76.2 Non Object-Oriented Approach

In this section, we look into implementations (in both C and Ada) of system AB that don’t make use of object-oriented programming.

#### 76.2.1 Starting point in C

Let's start with an implementation in C for the system described above:

[C]

Listing 1: system_a.h

```c
typedef struct {
    float val[2];
    int active;
} A;

void A_activate (A *a);

int A_is_active (A *a);

float A_value (A *a);

void A_deactivate (A *a);
```

Listing 2: system_a.c

```c
#include "system_a.h"

void A_activate (A *a)
{
    int i;
    for (i = 0; i < 2; i++)
    {
        a->val[i] = 0.0;
    }
    a->active = 1;
}

int A_is_active (A *a)
{
    return a->active == 1;
}

float A_value (A *a)
{
    return (a->val[0] + a->val[1]) / 2.0;
}
```

(continues on next page)
void A_deactivate (A *a)
{
  a->active = 0;
}

Listing 3: system_b.h

typedef struct {
  float val;
  int  active;
} B;

void B_activate (B *b);
int B_is_active (B *b);
float B_value (B *b);
void B_deactivate (B *b);

Listing 4: system_b.c

#include "system_b.h"

void B_activate (B *b)
{
  b->val = 0.0;
  b->active = 1;
}

int B_is_active (B *b)
{
  return b->active == 1;
}

float B_value (B *b)
{
  return b->val;
}

void B_deactivate (B *b)
{
  b->active = 0;
}

Listing 5: system_ab.h

#include "system_a.h"
#include "system_b.h"

typedef struct {
  A a;
  B b;
} AB;

void AB_activate (AB *ab);
int AB_is_active (AB *ab);

(continues on next page)
float AB_value (AB *ab);

int AB_check (AB *ab);

void AB_deactivate (AB *ab);

---

**Listing 6: system_ab.c**

```c
#include <math.h>
#include "system_ab.h"

void AB_activate (AB *ab)
{
    A_activate (&ab->a);
    B_activate (&ab->b);
}

int AB_is_active (AB *ab)
{
    return A_is_active(&ab->a) && B_is_active(&ab->b);
}

float AB_value (AB *ab)
{
    return (A_value (&ab->a) + B_value (&ab->b)) / 2;
}

int AB_check (AB *ab)
{
    const float threshold = 0.1;
    return fabs (A_value (&ab->a) - B_value (&ab->b)) < threshold;
}

void AB_deactivate (AB *ab)
{
    A_deactivate (&ab->a);
    B_deactivate (&ab->b);
}
```

---

**Listing 7: main.c**

```c
#include <stdio.h>
#include "system_ab.h"

void display_active (AB *ab)
{
    if (AB_is_active (ab))
        printf ("System AB is active.\n");
    else
        printf ("System AB is not active.\n");
}

void display_check (AB *ab)
{
    if (AB_check (ab))
        printf ("System AB check: PASSED.\n");
    else
        printf ("System AB check: FAILED.\n");
}
```

(continues on next page)
Here, each system is implemented in a separate set of header and source-code files. For example, the API of system AB is in system_ab.h and its implementation in system_ab.c.

In the main application, we instantiate system AB and activate it. Then, we proceed to display the activation state and the result of the system's health check. Finally, we deactivate the system and display the activation state again.

### 76.2.2 Initial translation to Ada

The direct implementation in Ada is:

```
[Ada]
Listing 8: system_a.ads

package System_A is

  type Val_Array is array (Positive range <>) of Float;

type A is record
  Val : Val_Array (1 .. 2);
  Active : Boolean;
end record;

procedure A_Activate (E : in out A);
function A_Is_Active (E : A) return Boolean;
function A_Value (E : A) return Float;
```

(continues on next page)
procedure A_Deactivate (E : in out A);
end System_A;

Listing 9: system_a.adb

package body System_A is
  procedure A_Activate (E : in out A) is
  begin
    E.Val := (others => 0.0);
    E.Active := True;
  end A_Activate;

  function A_Is_Active (E : A) return Boolean is
  begin
    return E.Active;
  end A_Is_Active;

  function A_Value (E : A) return Float is
  begin
    return (E.Val (1) + E.Val (2)) / 2.0;
  end A_Value;

  procedure A_Deactivate (E : in out A) is
  begin
    E.Active := False;
  end A_Deactivate;
end System_A;

Listing 10: system_b.ads

package System_B is
  type B is record
    Val  : Float;
    Active : Boolean;
  end record;

  procedure B_Activate (E : in out B);
  function B_Is_Active (E : B) return Boolean;
  function B_Value (E : B) return Float;
  procedure B_Deactivate (E : in out B);
end System_B;

Listing 11: system_b.adb

package body System_B is
  procedure B_Activate (E : in out B) is
  begin
    E.Val  := 0.0;
    E.Active := True;
  end B_Activate;

(continues on next page)
function B_Is_Active (E : B) return Boolean is
begin
  return E.Active;
end B_Is_Active;

function B_Value (E : B) return Float is
begin
  return E.Val;
end B_Value;

procedure B_Deactivate (E : in out B) is
begin
  E.Active := False;
end B_Deactivate;

end System_B;

Listing 12: system_ab.ads

with System_A; use System_A;
with System_B; use System_B;

package System_AB is

  type AB is record
    SA : A;
    SB : B;
  end record;

  procedure AB_Activate (E : in out AB);
  function AB_Is_Active (E : AB) return Boolean;
  function AB_Value (E : AB) return Float;
  function AB_Check (E : AB) return Boolean;
  procedure AB_Deactivate (E : in out AB);

end System_AB;

Listing 13: system_ab.adb

package body System_AB is

  procedure AB_Activate (E : in out AB) is
begin
  A_Activate (E.SA);
  B_Activate (E.SB);
end AB_Activate;

  function AB_Is_Active (E : AB) return Boolean is
begin
  return A_Is_Active (E.SA) and B_Is_Active (E.SB);
end AB_Is_Active;

  function AB_Value (E : AB) return Float is
begin
  return (A_Value (E.SA) + B_Value (E.SB)) / 2.0;
end AB_Value;

(continues on next page)
function AB_Check (E : AB) return Boolean is
    Threshold : constant := 0.1;
begin
    return abs (A_Value (E.SA) - B_Value (E.SB)) < Threshold;
end AB_Check;

procedure AB_Deactivate (E : in out AB) is
begin
    A_Deactivate (E.SA);
    B_Deactivate (E.SB);
end AB_Deactivate;

end System_AB;

with Ada.Text_IO; use Ada.Text_IO;
with System_AB; use System_AB;

procedure Main is

    procedure Display_Active (E : AB) is
    begin
        if AB_Is_Active (E) then
            Put_Line ("System AB is active");
        else
            Put_Line ("System AB is not active");
        end if;
    end Display_Active;

    procedure Display_Check (E : AB) is
    begin
        if AB_Check (E) then
            Put_Line ("System AB check: PASSED");
        else
            Put_Line ("System AB check: FAILED");
        end if;
    end Display_Check;

    S : AB;
    begin
        Put_Line ("Activating system AB...");
        AB_Activate (S);
        Display_Active (S);
        Display_Check (S);
        Put_Line ("Deactivating system AB...");
        AB_Deactivate (S);
        Display_Active (S);
    end Main;

Listing 14: main.adb

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.HandsOnOOP.System_AB_Ada
MD5: f2e3dfb3874e5edc5ea90c01961cf64

Runtime output
As you can see, this is a direct translation that doesn't change much of the structure of the original C code. Here, the goal was to simply translate the system from one language to another and make sure that the behavior remains the same.

### 76.2.3 Improved Ada implementation

By analyzing this direct implementation, we may notice the following points:

- Packages System_A, System_B and System_AB are used to describe aspects of the same system. Instead of having three distinct packages, we could group them as child packages of a common parent package — let's call it Simple, since this system is supposed to be simple. This approach has the advantage of allowing us to later use the parent package to implement functionality that is common for all parts of the system.

- Since we have subprograms that operate on types A, B and AB, we should avoid exposing the record components by moving the type declarations to the private part of the corresponding packages.

- Since Ada supports subprogram overloading — as discussed in this section from chapter 2 (page 1409) —, we don't need to have different names for subprograms with similar functionality. For example, instead of having A_Is_Active and B_Is_Active, we can simply name these functions Is_Active for both types A and B.

- Some of the functions — such as A_Is_Active and A_Value — are very simple, so we could simplify them with expression functions.

This is an update to the implementation that addresses all the points above:

```ada
[Ada]

Listing 15: simple.ads

package Simple
  with Pure
is
end Simple;

Listing 16: simple-system_a.ads

package Simple.System_A is
  type A is private;
  procedure Activate (E : in out A);
  function Is_Active (E : A) return Boolean;
  function Value (E : A) return Float;
  procedure Finalize (E : in out A);
private
  type Val_Array is array (Positive range <>) of Float;
end Simple.System_A;
```

(continues on next page)
type A is record
    Val : Val_Array (1..2);
    Active : Boolean;
end record;
end Simple.System_A;

Listing 17: simple-system_a.adb

package body Simple.System_A is

    procedure Activate (E : in out A) is
        begin
            E.Val := (others => 0.0);
            E.Active := True;
            end Activate;

    function Is_Active (E : A) return Boolean is
        (E.Active);

    function Value (E : A) return Float is
        begin
            return (E.Val (1) + E.Val (2)) / 2.0;
            end Value;

    procedure Finalize (E : in out A) is
        begin
            E.Active := False;
            end Finalize;

end Simple.System_A;

Listing 18: simple-system_b.ads

package Simple.System_B is

type B is private;

    procedure Activate (E : in out B);

    function Is_Active (E : B) return Boolean;

    function Value (E : B) return Float;

    procedure Finalize (E : in out B);

private

    type B is record
        Val : Float;
        Active : Boolean;
    end record;

end Simple.System_B;

Listing 19: simple-system_b.adb

package body Simple.System_B is

(continues on next page)
procedure Activate (E : in out B) is
begin
  E.Val := 0.0;
  E.Active := True;
end Activate;

function Is_Active (E : B) return Boolean is
begin
  return E.Active;
end Is_Active;

function Value (E : B) return Float is
(E.Val);

procedure Finalize (E : in out B) is
begin
  E.Active := False;
end Finalize;
end Simple.System_B;

with Simple.System_A; use Simple.System_A;
with Simple.System_B; use Simple.System_B;

package Simple.System_AB is

  type AB is private;
  procedure Activate (E : in out AB);
  function Is_Active (E : AB) return Boolean;
  function Value (E : AB) return Float;
  function Check (E : AB) return Boolean;
  procedure Finalize (E : in out AB);

private

  type AB is record
    SA : A;
    SB : B;
  end record;
end Simple.System_AB;

package body Simple.System_AB is

  procedure Activate (E : in out AB) is
begin
  Activate (E.SA);
  Activate (E.SB);
end Activate;

function Is_Active (E : AB) return Boolean is
(continues on next page)
(Is_Active (E.SA) and Is_Active (E.SB));

function Value (E : AB) return Float is
  ((Value (E.SA) + Value (E.SB)) / 2.0);

function Check (E : AB) return Boolean is
  Threshold : constant := 0.1;
begin
  return abs (Value (E.SA) - Value (E.SB)) < Threshold;
end Check;

procedure Finalize (E : in out AB) is
begin
  Finalize (E.SA);
  Finalize (E.SB);
end Finalize;
end Simple.System_AB;

Listing 22: main.adb

with Ada.Text_IO; use Ada.Text_IO;
with Simple.System_AB; use Simple.System_AB;

procedure Main is

  procedure Display_Active (E : AB) is
  begin
    if Is_Active (E) then
      Put_Line ("System AB is active");
    else
      Put_Line ("System AB is not active");
    end if;
  end Display_Active;

  procedure Display_Check (E : AB) is
  begin
    if Check (E) then
      Put_Line ("System AB check: PASSED");
    else
      Put_Line ("System AB check: FAILED");
    end if;
  end Display_Check;

begin
  S := AB;
  Put_Line ("Activating system AB...”);
  Activate (S);
  Display_Active (S);
  Display_Check (S);
  Put_Line ("Deactivating system AB...”);
  Finalize (S);
  Display_Active (S);
end Main;

Code block metadata
76.3 First Object-Oriented Approach

Until now, we haven’t used any of the object-oriented programming features of the Ada language. So we can start by analyzing the API of systems A and B and deciding how to best abstract some of its elements using object-oriented programming.

76.3.1 Interfaces

The first thing we may notice is that we actually have two distinct sets of APIs there:

- one API for activating and deactivating the system.
- one API for retrieving the value of the system.

We can use this distinction to declare two interface types:

- Activation_IF for the Activate and Deactivate procedures and the Is_Active function;
- Value_Retrieval_IF for the Value function.

This is how the declaration could look like:

```ada
type Activation_IF is interface;
  procedure Activate (E : in out Activation_IF) is abstract;
  function Is_Active (E : Activation_IF) return Boolean is abstract;
  procedure Deactivate (E : in out Activation_IF) is abstract;

type Value_Retrieval_IF is interface;
  function Value (E : Value_Retrieval_IF) return Float is abstract;
```

Note that, because we are declaring interface types, all operations on those types must be abstract or, in the case of procedures, they can also be declared null. For example, we could change the declaration of the procedures above to this:

```ada
procedure Activate (E : in out Activation_IF) is null;
procedure Deactivate (E : in out Activation_IF) is null;
```

When an operation is declared abstract, we must override it for the type that derives from the interface. When a procedure is declared null, it acts as a do-nothing default. In this case, overriding the operation is optional for the type that derives from this interface.
### 76.3.2 Base type

Since the original system needs both interfaces we've just described, we have to declare another type that combines those interfaces. We can do this by declaring the interface type `Sys_Base`, which serves as the base type for systems A and B. This is the declaration:

```ada
type Sys_Base is interface and Activation_IF and Value_Retrieval_IF;
```

Since the system activation functionality is common for both systems A and B, we could implement it as part of `Sys_Base`. That would require changing the declaration from a simple interface to an abstract record:

```ada
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with null record;
```

Now, we can add the Boolean component to the record (as a private component) and over-ride the subprograms of the `Activation_IF` interface. This is the adapted declaration:

```ada
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with private;

overriding procedure Activate (E : in out Sys_Base);
overriding function Is_Active (E : Sys_Base) return Boolean;
overriding procedure Deactivate (E : in out Sys_Base);

private

type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with record
  Active : Boolean;
end record;
```

### 76.3.3 Derived types

In the declaration of the `Sys_Base` type we've just seen, we're not overriding the `Value` function — from the `Value_Retrieval_IF` interface — for the `Sys_Base` type, so it remains an abstract function for `Sys_Base`. Therefore, the `Sys_Base` type itself remains abstract and needs be explicitly declared as such.

We use this strategy to ensure that all types derived from `Sys_Base` need to implement their own version of the `Value` function. For example:

```ada
type A is new Sys_Base with private;

overriding function Value (E : A) return Float;
```

Here, the A type is derived from the `Sys_Base` and it includes its own version of the `Value` function by overriding it. Therefore, A is not an abstract type anymore and can be used to declare objects:

```ada
procedure Main is
  Obj : A;
  V  : Float;
begin
  Obj.Activate;
  V := Obj.Value;
end Main;
```

---

**Important**
Note that the use of the `overriding` keyword in the subprogram declaration is not strictly necessary. In fact, we could leave this keyword out, and the code would still compile. However, if provided, the compiler will check whether the information is correct.

Using the `overriding` keyword can help to avoid bad surprises — when you *may think* that you’re overriding a subprogram, but you’re actually not. Similarly, you can also write `not overriding` to be explicit about subprograms that are new primitives of a derived type. For example:

```ada
not overriding function Check (E : AB) return Boolean;
```

We also need to declare the values that are used internally in systems A and B. For system A, this is the declaration:

```ada
type A is new Sys_Base with private;
  overriding function Value (E : A) return Float;
private
  type Val_Array is array (Positive range <>) of Float;
  type A is new Sys_Base with record
    Val : Val_Array (1..2);
  end record;
```

### 76.3.4 Subprograms from parent

In the previous implementation, we’ve seen that the `A_Activate` and `B_Activate` procedures perform the following steps:

- initialize internal values;
- indicate that the system is active (by setting the `Active` flag to `True`).

In the implementation of the `Activate` procedure for the `Sys_Base` type, however, we’re only dealing with the second step. Therefore, we need to override the `Activate` procedure and make sure that we initialize internal values as well. First, we need to declare this procedure for type A:

```ada
type A is new Sys_Base with private;
overriding procedure Activate (E : in out A);
```

In the implementation of `Activate`, we should call the `Activate` procedure from the parent (`Sys_Base`) to ensure that whatever was performed for the parent will be performed in the derived type as well. For example:

```ada
overriding procedure Activate (E : in out A) is begin
  E.Val := (others => 0.0); -- Calling Activate for Sys_Base type:
  Sys_Base (E).Activate; -- this call initializes the Active flag.
end;
```

Here, by writing `Sys_Base (E)`, we’re performing a view conversion. Basically, we’re telling the compiler to view `E` not as an object of type `A`, but of type `Sys_Base`. When we do this, any operation performed on this object will be done as if it was an object of `Sys_Base` type, which includes calling the `Activate` procedure of the `Sys_Base` type.
Learning Ada

Important
If we write \( T \) (Obj).Proc, we're telling the compiler to call the Proc procedure of type \( T \) and apply it on Obj.

If we write \( T' \text{Class} \) (Obj).Proc, however, we're telling the compiler to dispatch the call. For example, if Obj is of derived type \( T_2 \) and there's an overridden Proc procedure for type \( T_2 \), then this procedure will be called instead of the Proc procedure for type \( T \).

76.3.5 Type AB

While the implementation of systems A and B is almost straightforward, it gets more interesting in the case of system AB. Here, we have a similar API, but we don't need the activation mechanism implemented in the abstract type Sys_Base. Therefore, deriving from Sys_Base is not the best option. Instead, when declaring the AB type, we can simply use the same interfaces as we did for Sys_Base, but keep it independent from Sys_Base. For example:

```ada
type AB is new Activation_IF and Value_Retrieval_IF with private;
private
    type AB is new Activation_IF and Value_Retrieval_IF with record
    SA : A;
    SB : B;
end record;
```

Naturally, we still need to override all the subprograms that are part of the Activation_IF and Value_Retrieval_IF interfaces. Also, we need to implement the additional Check function that was originally only available on system AB. Therefore, we declare these subprograms:

```ada
overriding procedure Activate (E : in out AB);
overriding function Is_Active (E : AB) return Boolean;
overriding procedure Deactivate (E : in out AB);

overriding function Value (E : AB) return Float;
not overriding function Check (E : AB) return Boolean;
```

76.3.6 Updated source-code

Finally, this is the complete source-code example:

[Ada]

```
Listing 23: simple.ads

package Simple is
    type Activation_IF is interface;
    procedure Activate (E : in out Activation_IF) is abstract;
    function Is_Active (E : Activation_IF) return Boolean is abstract;
    procedure Deactivate (E : in out Activation_IF) is abstract;
```

(continues on next page)
type Value_Retrieval_IF is interface;

function Value (E : Value_Retrieval_IF) return Float is abstract;

type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
with private;

overriding procedure Activate (E : in out Sys_Base);
overriding function Is_Active (E : Sys_Base) return Boolean;
overriding procedure Deactivate (E : in out Sys_Base);

private

end Sys_Base;

end Simple;

Listing 24: simple.adb

package body Simple is

overriding procedure Activate (E : in out Sys_Base) is
begin
  E.Active := True;
end Activate;

overriding function Is_Active (E : Sys_Base) return Boolean is
  (E.Active);
overriding procedure Deactivate (E : in out Sys_Base) is
begin
  E.Active := False;
end Deactivate;
end Simple;

Listing 25: simple-system_a.ads

package Simple.System_A is

type A is new Sys_Base with private;

overriding procedure Activate (E : in out A);

overriding function Value (E : A) return Float;

private

type Val_Array is array (Positive range <>) of Float;

type A is new Sys_Base with record
  Val : Val_Array (1 .. 2);
end record;
end Simple.System_A;
Listing 26: simple-system_a.adb

```
package body Simple.System_A is

   procedure Activate (E : in out A) is
      begin
         E.Val := (others => 0.0);
         Sys_Base (E).Activate;
      end Activate;

   function Value (E : A) return Float is
      pragma Assert (E.Val'Length = 2);
      begin
         return (E.Val (1) + E.Val (2)) / 2.0;
      end Value;

end Simple.System_A;
```

Listing 27: simple-system_b.ads

```
package Simple.System_B is

   type B is new Sys_Base with private;

   overriding procedure Activate (E : in out B);
   overriding function Value (E : B) return Float;

private

   type B is new Sys_Base with record
      Val : Float;
   end record;

end Simple.System_B;
```

Listing 28: simple-system_b.adb

```
package body Simple.System_B is

   procedure Activate (E : in out B) is
      begin
         E.Val := 0.0;
         Sys_Base (E).Activate;
      end Activate;

   function Value (E : B) return Float is
      (E.Val);

end Simple.System_B;
```

Listing 29: simple-system_ab.ads

```
with Simple.System_A; use Simple.System_A;
with Simple.System_B; use Simple.System_B;

package Simple.System_AB is

   type AB is new Activation_IF and Value_Retrieval_IF with private;

(continues on next page)
```
overriding procedure Activate (E : in out AB);
overriding function Is_Active (E : AB) return Boolean;
overriding procedure Deactivate (E : in out AB);
overriding function Value (E : AB) return Float;
not overriding function Check (E : AB) return Boolean;

private

type AB is new Activation_IF and Value_Retrieval_IF with record
  SA : A;
  SB : B;
end record;
end Simple.System_AB;

Listing 30: simple-system_ab.adb

package body Simple.System_AB is

  procedure Activate (E : in out AB) is
  begin
    E.SA.Activate;
    E.SB.Activate;
  end Activate;

  function Is_Active (E : AB) return Boolean is
    (E.SA.Is_Active and E.SB.Is_Active);
  end Is_Active;

  procedure Deactivate (E : in out AB) is
  begin
    E.SA.Deactivate;
    E.SB.Deactivate;
  end Deactivate;

  function Value (E : AB) return Float is
    ((E.SA.Value + E.SB.Value) / 2.0);
  end Value;

  function Check (E : AB) return Boolean is
    Threshold : constant := 0.1;
  begin
    return abs (E.SA.Value - E.SB.Value) < Threshold;
  end Check;

end Simple.System_AB;

Listing 31: main.adb

with Ada.Text_IO; use Ada.Text_IO;
with Simple.System_AB; use Simple.System_AB;

procedure Main is

  procedure Display_Active (E : AB) is
  begin
    if Is_Active (E) then
      Put_Line ("System AB is active");
    else
      (continues on next page)
Learning Ada

(continued from previous page)

```ada
12 Put_Line ("System AB is not active");
end if;
end Display_Active;

procedure Display_Check (E : AB) is
begin
if Check (E) then
Put_Line ("System AB check: PASSED");
else
Put_Line ("System AB check: FAILED");
end if;
end Display_Check;

S : AB;
begin
Put_Line ("Activating system AB...");
Activate (S);
Display_Active (S);
Display_Check (S);
Put_Line ("Deactivating system AB...");
Deactivate (S);
Display_Active (S);
end Main;
```

Code block metadata

Project: Courses.Ada_For_Emb  
MD5: 02adee1f81b025007244bd6d13e8b5a3

Runtime output

Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
System AB is not active

76.4 Further Improvements

When analyzing the complete source-code, we see that there are at least two areas that we could still improve.

76.4.1 Dispatching calls

The first issue concerns the implementation of the Activate procedure for types derived from Sys_Base. For those derived types, we're expecting that the Activate procedure of the parent must be called in the implementation of the overriding Activate procedure. For example:

```ada
package body Simple.System_A is
procedure Activate (E : in out A) is
begin
(continues on next page)
```
If a developer forgets to call that specific Activate procedure, however, the system won’t work as expected. A better strategy could be the following:

- Declare a new Activation_Reset procedure for Sys_Base type.
- Make a dispatching call to the Activation_Reset procedure in the body of the Activate procedure (of the Sys_Base type).
- Let the derived types implement their own version of the Activation_Reset procedure.

This is a simplified view of the implementation using the points described above:

```ada
package Simple is
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with private;
not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;
end Simple;
package body Simple is
procedure Activate (E : in out Sys_Base) is
begin
  -- NOTE: calling "E.Activation_Reset" does NOT dispatch!
  -- We need to use the 'Class attribute here -- not using this
  -- attribute is an error that will be caught by the compiler.
  Sys_Base'Class (E).Activation_Reset;
  E.Active := True;
end Activate;
end Simple;
package Simple.System_A is
type A is new Sys_Base with private;
private
type Val_Array is array (Positive range <>) of Float;
type A is new Sys_Base with record
  Val : Val_Array (1 .. 2);
end record;
overriding procedure Activation_Reset (E : in out A);
end Simple.System_A;
package body Simple.System_A is
procedure Activation_Reset (E : in out A) is
begin
  E.Val := (others => 0.0);
end Activation_Reset;
```

(continues on next page)
An important detail is that, in the implementation of Activate, we use Sys_Base'Class to ensure that the call to Activation_Reset will dispatch. If we had just written E.Activation_Reset instead, then we would be calling the Activation_Reset procedure of Sys_Base itself, which is not what we actually want here. The compiler will catch the error if you don't do the conversion to the class-wide type, because it would otherwise be a statically-bound call to an abstract procedure, which is illegal at compile-time.

### 76.4.2 Dynamic allocation

The next area that we could improve is in the declaration of the system AB. In the previous implementation, we were explicitly describing the two components of that system, namely a component of type A and a component of type B:

```ada
type AB is new Activation_IF and Value_Retrieval_IF with record
  SA : A;
  SB : B;
end record;
```

Of course, this declaration matches the system requirements that we presented in the beginning. However, we could use strategies that make it easier to incorporate requirement changes later on. For example, we could hide this information about systems A and B by simply declaring an array of components of type access Sys_Base'Class and allocate them dynamically in the body of the package. Naturally, this approach might not be suitable for certain platforms. However, the advantage would be that, if we wanted to replace the component of type B by a new component of type C, for example, we wouldn't need to change the interface. This is how the updated declaration could look like:

```ada
type Sys_Base_Class_Access is access Sys_Base'Class;
type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;

type AB is limited new Activation_IF and Value_Retrieval_IF with record
  S_Array : Sys_Base_Array (1 .. 2);
end record;
```

**Important**

Note that we're now using the `limited` keyword in the declaration of type AB. That is necessary because we want to prevent objects of type AB being copied by assignment, which would lead to two objects having the same (dynamically allocated) subsystems A and B internally. This change requires that both Activation_IF and Value_Retrieval_IF are declared limited as well.

The body of Activate could then allocate those components:

```ada
procedure Activate (E : in out AB) is
begin
  E.S_Array := (new A, new B);
  for S of E.S_Array loop
    S.Activate;
  end loop;
end Activate;
```

And the body of Deactivate could deallocate them:
procedure Deactivate (E : in out AB) is
  procedure Free is
    new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
  begin
    for S of E.S_Array loop
      S.Deactivate;
      Free (S);
    end loop;
  end Deactivate;

76.4.3 Limited controlled types

Another approach that we could use to implement the dynamic allocation of systems A and B is to declare AB as a limited controlled type—based on the Limited_Controlled type of the Ada.Finalization package.

The Limited_Controlled type includes the following operations:

- Initialize, which is called when objects of a type derived from the Limited_Controlled type are being created—by declaring an object of the derived type, for example—, and
- Finalize, which is called when objects are being destroyed—for example, when an object gets out of scope at the end of a subprogram where it was created.

In this case, we must override those procedures, so we can use them for dynamic memory allocation. This is a simplified view of the update implementation:

package Simple.System_AB is
  type AB is limited new Ada.Finalization.Limited_Controlled and
    Activation_IF and Value_Retrieval_IF with private;

  overriding procedure Initialize (E : in out AB);
  overriding procedure Finalize (E : in out AB);
end Simple.System_AB;

package body Simple.System_AB is
  overriding procedure Initialize (E : in out AB) is
    begin
      E.S_Array := (new A, new B);
      end Initialize;

  overriding procedure Finalize (E : in out AB) is
    procedure Free is
      new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
    begin
      for S of E.S_Array loop
        Free (S);
      end loop;
      end Finalize;
end Simple.System_AB;
Finally, this is the complete updated source-code example:

```
[Ada]

Listing 32: simple.ads

package Simple is

    type Activation_IF is limited interface;
    procedure Activate (E : in out Activation_IF) is abstract;
    function Is_Active (E : Activation_IF) return Boolean is abstract;
    procedure Deactivate (E : in out Activation_IF) is abstract;

    type Value_Retrieval_IF is limited interface;
    function Value (E : Value_Retrieval_IF) return Float is abstract;

    type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
        private
        overriding procedure Activate (E : in out Sys_Base);
        overriding function Is_Active (E : Sys_Base) return Boolean;
        overriding procedure Deactivate (E : in out Sys_Base);
        not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;
    private
        type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
            record
                Active : Boolean;
            end record;

end Simple;

Listing 33: simple.adb

package body Simple is

    procedure Activate (E : in out Sys_Base) is
        begin
            -- NOTE: calling "E.Activation_Reset" does NOT dispatch!
            -- We need to use the 'Class attribute:
            -- Sys_Base'Class (E).Activation_Reset;
            E.Active := True;
        end Activate;

    function Is_Active (E : Sys_Base) return Boolean is
        (E.Active);

    procedure Deactivate (E : in out Sys_Base) is
        begin
            E.Active := False;
        end Deactivate;

end Simple;
```
package Simple.System_A is
  type A is new Sys_Base with private;
  overriding function Value (E : A) return Float;
private
  type Val_Array is array (Positive range <>) of Float;
  type A is new Sys_Base with record
    Val : Val_Array (1 .. 2);
  end record;
  overriding procedure Activation_Reset (E : in out A);
end Simple.System_A;

package body Simple.System_A is
  procedure Activation_Reset (E : in out A) is
    begin
      E.Val := (others => 0.0);
    end Activation_Reset;
  function Value (E : A) return Float
    pragma Assert (E.Val'Length = 2);
    begin
      return (E.Val (1) + E.Val (2)) / 2.0;
    end Value;
end Simple.System_A;

package Simple.System_B is
  type B is new Sys_Base with private;
  overriding function Value (E : B) return Float;
private
  type B is new Sys_Base with record
    Val : Float;
  end record;
  overriding procedure Activation_Reset (E : in out B);
end Simple.System_B;

package body Simple.System_B is
  procedure Activation_Reset (E : in out B) is
(continues on next page)
begin
E.Val := 0.0;
end Activation_Reset;

function Value (E : B) return Float is
  (E.Val);
end Simple.System_B;

Listing 38: simple-system_ab.ads

with Ada.Finalization;

package Simple.System_AB is
  type AB is limited new Ada.Finalization.Limited_Controlled and
    Activation_IF and Value_Retrieval_IF with private;
  overriding procedure Activate (E : in out AB);
  overriding function Is_Active (E : AB) return Boolean;
  overriding procedure Deactivate (E : in out AB);
  not overriding function Check (E : AB) return Boolean;

private
  type Sys_Base_Class_Access is access Sys_Base'Class;
  type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;
  type AB is limited new Ada.Finalization.Limited_Controlled and
    Activation_IF and Value_Retrieval_IF with record
      S_Array : Sys_Base_Array (1 .. 2);
    end record;
  overriding procedure Initialize (E : in out AB);
  overriding procedure Finalize (E : in out AB);
end Simple.System_AB;

Listing 39: simple-system_ab.adb

with Ada.Unchecked_Deallocation;

with Simple.System_A; use Simple.System_A;
with Simple.System_B; use Simple.System_B;

package body Simple.System_AB is
  overriding procedure Initialize (E : in out AB) is
    begin
      E.S_Array := (new A, new B);
    end Initialize;
  overriding procedure Finalize (E : in out AB) is
    new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
    begin
      for S in E.S_Array loop
        Free
      end loop;
    end Finalize;
end Simple.System_AB;
procedure Activate (E : in out AB) is
begin
   for S of E.S_Array loop
      S.Activate;
   end loop;
end Activate;

function Is_Active (E : AB) return Boolean is
   (for all S of E.S_Array => S.Is_Active);
end Is_Active;

procedure Deactivate (E : in out AB) is
begin
   for S of E.S_Array loop
      S.Deactivate;
   end loop;
end Deactivate;

function Value (E : AB) return Float is
   ((E.S_Array (1).Value + E.S_Array (2).Value) / 2.0);
end Value;

function Check (E : AB) return Boolean is
   Threshold : constant := 0.1;
   begin
      return abs (E.S_Array (1).Value - E.S_Array (2).Value) < Threshold;
   end Check;
end Check;

end Simple.System_AB;

Listing 40: main.adb

with Ada.Text_IO; use Ada.Text_IO;

with Simple.System_AB; use Simple.System_AB;

procedure Main is

   procedure Display_Active (E : AB) is
   begin
      if Is_Active (E) then
         Put_Line ("System AB is active");
      else
         Put_Line ("System AB is not active");
      end if;
   end Display_Active;

   procedure Display_Check (E : AB) is
   begin
      if Check (E) then
         Put_Line ("System AB check: PASSED");
      else
         Put_Line ("System AB check: FAILED");
      end if;
   end Display_Check;

   S : AB;
   begin
      Put_Line ("Activating system AB...");
end Main;
Learning Ada

28 Activate (S);
29 Display_Active (S);
30 Display_Check (S);
31 Put_Line ("Deactivating system AB..." interim code
32 Deactivate (S);
33 Display_Active (S);
34 end Main;

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.HandsOnOOP.System_AB_Ada_OOP_2
MD5: f8d0d4a07aa045cb30bddd88db2215a

Runtime output

Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
System AB is not active

Naturally, this is by no means the best possible implementation of system AB. By applying other software design strategies that we haven't covered here, we could most probably think of different ways to use object-oriented programming to improve this implementation. Also, in comparison to the original implementation (page 1581), we recognize that the amount of source-code has grown. On the other hand, we now have a system that is factored nicely, and also more extensible.
Part VIII

SPARK Ada for the MISRA C Developer
This book presents the SPARK technology — the SPARK subset of Ada and its supporting static analysis tools — through an example-driven comparison with the rules in the widely known MISRA C subset of the C language.

This document was prepared by Yannick Moy, with contributions and review from Ben Bros-gol.

Note: The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note: Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website336. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9d7a71533d982d05e6ab

you will find it in this directory:

projects/Courses/Intro_To_Ada/Imperative_Language/Greet/cba89a34b87c9d7a71533d982d05e6ab/

In order to use this code example, just follow these steps:
1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).
MISRAC appeared in 1998 as a coding standard for C; it focused on avoiding error-prone programming features of the C programming language rather than on enforcing a particular programming style. A study of coding standards for C by Les Hatton³³⁷ found that, compared to ten typical coding standards for C, MISRA C was the only one to focus exclusively on error avoidance rather than style enforcement, and by a very large margin.

The popularity of the C programming language, as well as its many traps and pitfalls, have led to the huge success of MISRA C in domains where C is used for high-integrity software. This success has driven tool vendors to propose many competing implementations of MISRA C checkers. Tools compete in particular on the coverage of MISRA C guidelines that they help enforce, as it is impossible to enforce the 16 directives and 143 rules (collectively referred to as guidelines) of MISRA C.

The 16 directives are broad guidelines, and it is not possible to define compliance in a unique and automated way. For example, "all code should be traceable to documented requirements" (Directive 3.1). Thus no tool is expected to enforce directives, as the MISRA C:2012 states in introduction to the guidelines: "different tools may place widely different interpretations on what constitutes a non-compliance."

The 143 rules on the contrary are completely and precisely defined, and "static analysis tools should be capable of checking compliance with rules". But the same sentence continues with "subject to the limitations described in Section 6.5", which addresses "decidability of rules". It turns out that 27 rules out of 143 are not decidable, so no tool can always detect all violations of these rules without at the same time reporting "false alarms" on code that does not constitute a violation.

An example of an undecidable rule is rule 1.3: "There shall be no occurrence of undefined or critical unspecified behaviour." Appendix H of MISRA C:2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable. For the most part, MISRA C checkers ignore undecidable rules such as rule 1.3 and instead focus on the 116 rules for which detection of violations can be automated. It is telling in that respect that the MISRA C:2012 document and its accompanying set of examples (which can be downloaded from the MISRA website³³⁹) does not provide any example for rule 1.3.

However, violations of undecidable rules such as rule 1.3 are known to have dramatic impact on software quality. Violations of rule 1.3 in particular are commonly amplified by compilers using the permission in the C standard to optimize aggressively without looking at the consequences for programs with undefined or critical unspecified behavior. It would be valid to ignore these rules if violations did not occur in practice, but on the contrary even experienced programmers write C code with undefined or critical unspecified behavior. An example comes from the MISRA C Committee itself in its "Appendix I: Example deviation record" of the MISRA C:2012 document, repeated in "Appendix A: Example deviation record" of the MISRA C: Compliance 2016 document³⁴⁰, where the following code is proposed as

³³⁷ https://www.leshatton.org/Documents/MISRAC.pdf
³³⁸ https://en.wikipedia.org/wiki/MISRA_C
³³⁹ https://www.misra.org.uk
³⁴⁰ https://www.misra.org.uk/LinkClick.aspx?fileticket=w_Syhpkf7x%A%3d&tabid=57
a deviation of rule 10.6 “The value of a composite expression shall not be assigned to an object with wider essential type”:

```c
uint32_t prod = qty * time_step;
```

Here, the multiplication of two unsigned 16-bit values and assignment of the result to an unsigned 32-bit variable constitutes a violation of the aforementioned rule, which gets justified for efficiency reasons. What the authors seem to have missed is that the multiplication is then performed with the signed integer type `int` instead of the target unsigned type `uint32_t`. Thus the multiplication of two unsigned 16-bit values may lead to an overflow of the 32-bit intermediate signed result, which is an occurrence of an undefined behavior. In such a case, a compiler is free to assume that the value of `prod` cannot exceed $2^{31} - 1$ (the maximal value of a signed 32-bit integer) as otherwise an undefined behavior would have been triggered. For example, the undefined behavior with values 65535 for `qty` and `time_step` is reported when running the code compiled by either the GCC or LLVM compiler with option `-fsanitize=undefined`.

The MISRA C checkers that detect violations of undecidable rules are either unsound tools that can detect only some of the violations, or sound tools that guarantee to detect all such violations at the cost of possibly many false reports of violations. This is a direct consequence of undecidability. However, static analysis technology is available that can achieve soundness without inundating users with false alarms. One example is the SPARK toolset developed by AdaCore, Altran and Inria, which is based on four principles:

- The base language Ada provides a solid foundation for static analysis through a well-defined language standard, strong typing and rich specification features.
- The SPARK subset of Ada restricts the base language in essential ways to support static analysis, by controlling sources of ambiguity such as side-effects and aliasing.
- The static analysis tools work mostly at the granularity of an individual function, making the analysis more precise and minimizing the possibility of false alarms.
- The static analysis tools are interactive, allowing users to guide the analysis if necessary or desired.

In this document, we show how SPARK can be used to achieve high code quality with guarantees that go beyond what would be feasible with MISRA C.

An on-line and interactive version of this document is available at AdaCore's learn.adacore.com site.\(^\text{341}\)

\(^{341}\) [https://learn.adacore.com/courses/SPARK_for_the_MISRA_C_Developer](https://learn.adacore.com/courses/SPARK_for_the_MISRA_C_Developer)
ENFORCING BASIC PROGRAM CONSISTENCY

Many consistency properties that are taken for granted in other languages are not enforced in C. The basic property that all uses of a variable or function are consistent with its type is not enforced by the language and is also very difficult to enforce by a tool. Three features of C contribute to that situation:

- the textual-based inclusion of files means that every included declaration is subject to a possibly different reinterpretation depending on context.
- the lack of consistency requirements across translation units means that type inconsistencies can only be detected at link time, something linkers are ill-equipped to do.
- the default of making a declaration externally visible means that declarations that should be local will be visible to the rest of the program, increasing the chances for inconsistencies.

MISRA C contains guidelines on all three fronts to enforce basic program consistency.

78.1 Taming Text-Based Inclusion

The text-based inclusion of files is one of the dated idiosyncracies of the C programming language that was inherited by C++ and that is known to cause quality problems, especially during maintenance. Although multiple inclusion of a file in the same translation unit can be used to emulate template programming, it is generally undesirable. Indeed, MISRA C defines Directive 4.10 precisely to forbid it for header files: "Precautions shall be taken in order to prevent the contents of a header file being included more than once".

The subsequent section on "Preprocessing Directives" contains 14 rules restricting the use of text-based inclusion through preprocessing. Among other things these rules forbid the use of the #undef directive (which works around conflicts in macro definitions introduced by text-based inclusion) and enforces the well-known practice of enclosing macro arguments in parentheses (to avoid syntactic reinterpretations in the context of the macro use).

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context instead of textual inclusion of content, using with clauses:

```
1 with Ada.Text_IO;
2
3 procedure Hello_World is
4 begin
5   Ada.Text_IO.Put_Line ("hello, world!");
6 end Hello_World;
```

Listing 1: hello_world.adb
Learning Ada

Runtime output

hello, world!

Note that with clauses are only allowed at the beginning of files; the compiler issues an error if they are used elsewhere:

Listing 2: hello_world.adb

procedure Hello_World is
with Ada.Text_IO; -- Illegal
begin
  Ada.Text_IO.Put_Line ("hello, world!");
end Hello_World;

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: afa19e8e2c114a5832b49e9efcbe675e

Importing a unit (i.e., specifying it in a with clause) multiple times is harmless, as it is equivalent to importing it once, but a compiler warning lets us know about the redundancy:

Listing 3: hello_world.adb

with Ada.Text_IO;
with Ada.Text_IO; -- Legal but useless
procedure Hello_World is
begin
  Ada.Text_IO.Put_Line ("hello, world!");
end Hello_World;

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 270928968d7beb4809af9e62df530722

Runtime output

hello, world!

The order in which units are imported is irrelevant. All orders are valid and have the same semantics.

No conflict arises from importing multiple units, even if the same name is defined in several, since each unit serves as namespace for the entities which it defines. So we can define our own version of Put_Line in some Helper unit and import it together with the standard version defined in Ada.Text_IO:

Listing 4: helper.ads

package Helper is
  procedure Put_Line (S : String);
end Helper;
Learning Ada

Listing 5: helper.adb

```ada
with Ada.Text_IO;

package body Helper is
    procedure Put_Line (S : String) is
    begin
        Ada.Text_IO.Put_Line ("Start helper version");
        Ada.Text_IO.Put_Line (S);
        Ada.Text_IO.Put_Line ("End helper version");
    end Put_Line;
end Helper;
```

Listing 6: hello_world.adb

```ada
with Ada.Text_IO;
with Helper;

procedure Hello_World is
    begin
        Ada.Text_IO.Put_Line ("hello, world!");
        Helper.Put_Line ("hello, world!");
    end Hello_World;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 5fa012cc996e24e3b1f604e35bbba44f

Runtime output

hello, world!
Start helper version
hello, world!
End helper version

The only way a conflict can arise is if we want to be able to reference Put_Line directly, without using the qualified name Ada.Text_IO.Put_Line or Helper.Put_Line. The use clause makes public declarations from a unit available directly:

Listing 7: helper.ads

```ada
package Helper is
    procedure Put_Line (S : String);
end Helper;
```

Listing 8: helper.adb

```ada
with Ada.Text_IO;

package body Helper is
    procedure Put_Line (S : String) is
    begin
        Ada.Text_IO.Put_Line ("Start helper version");
        Ada.Text_IO.Put_Line (S);
        Ada.Text_IO.Put_Line ("End helper version");
    end Put_Line;
end Helper;
```

78.1. Taming Text-Based Inclusion
Listing 9: hello_world.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Helper; use Helper;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("hello, world!");
   Helper.Put_Line ("hello, world!");
   Put_Line ("hello, world!"); -- ERROR
end Hello_World;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 405e138d78e0dc869e8a340681d87e61

Build output

```
hello_world.adb:8:04: error: ambiguous expression (cannot resolve "Put_Line")
hello_world.adb:8:04: error: possible interpretation at helper.ads:2
hello_world.adb:8:04: error: possible interpretation at a-textio.ads:507
gprbuild: *** compilation phase failed
```

Here, both units Ada.Text_IO and Helper define a procedure Put_Line taking a String as argument, so the compiler cannot disambiguate the direct call to Put_Line and issues an error.

Note that it helpfully points to candidate declarations, so that the user can decide which qualified name to use as in the previous two calls.

Issues arising in C as a result of text-based inclusion of files are thus completely prevented in SPARK (and Ada) thanks to semantic import of units. Note that the C++ committee identified this weakness some time ago and has approved\(^{342}\) the addition of modules to C++20, which provide a mechanism for semantic import of units.

### 78.2 Hardening Link-Time Checking

An issue related to text-based inclusion of files is that there is no single source for declaring the type of a variable or function. If a file origin.c defines a variable var and functions fun and print:

Listing 10: origin.c

```c
#include <stdio.h>
int var = 0;
int fun() {
   return 1;
}
void print() {
   printf("var = %d\n", var);
}
```

Code block metadata

and the corresponding header file `origin.h` declares `var`, `fun`, and `print` as having external linkage:

```
Listing 11: origin.h

extern int var;
extern int fun();
extern void print();
```

then client code can include `origin.h` with declarations for `var` and `fun`:

```
Listing 12: main.c

#include "origin.h"

int main() {
    var = fun();
    print();
    return 0;
}
```

or, equivalently, repeat these declarations directly:

```
Listing 13: main.c

extern int var;
extern int fun();
extern void print();

int main() {
    var = fun();
    print();
    return 0;
}
```

### 78.2. Hardening Link-Time Checking
Then, if an inconsistency is introduced in the type of var of fun between these alternative declarations and their actual type, the compiler cannot detect it. Only the linker, which has access to the set of object files for a program, can detect such inconsistencies. However, a linker's main task is to link, not to detect inconsistencies, and so inconsistencies in the type of variables and functions in most cases cannot be detected. For example, most linkers cannot detect if the type of var or the return type of fun is changed to float in the declarations above. With the declaration of var changed to float, the above program compiles and runs without errors, producing the erroneous output var = 1065353216 instead of var = 1. With the return type of fun changed to float instead, the program still compiles and runs without errors, producing this time the erroneous output var = 0.

The inconsistency just discussed is prevented by MISRA C Rule 8.3 "All declarations of an object or function shall use the same names and type qualifiers". This is a decidable rule, but it must be enforced at system level, looking at all translation units of the complete program. MISRA C Rule 8.6 also requires a unique definition for a given identifier across translation units, and Rule 8.5 requires that an external declaration shared between translation units comes from the same file. There is even a specific section on "Identifiers" containing 9 rules requiring uniqueness of various categories of identifiers.

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context using with clauses to provide a unique declaration for each entity.

### 78.3 Going Towards Encapsulation

Many problems in C stem from the lack of encapsulation. There is no notion of namespace that would allow a file to make its declarations available without risking a conflict with other files. Thus MISRA C has a number of guidelines that discourage the use of external declarations:

- Directive 4.8 encourages hiding the definition of structures and unions in implementation files (.c files) when possible: "If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden."

- Rule 8.7 forbids the use of external declarations when not needed: "Functions and objects should not be defined with external linkage if they are referenced in only one translation unit."

- Rule 8.8 forces the explicit use of keyword static when appropriate: "The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage."

The basic unit of modularization in SPARK, as in Ada, is the package. A package always has a spec (in an .ads file), which defines the interface to other units. It generally also has a body (in an .adb file), which completes the spec with an implementation. Only declarations from the package spec are visible from other units when they import (with) the package. In fact, only declarations from what is called the "visible part" of the spec (before the keyword private) are visible from units that with the package.

Listing 14: helper.ads
```ada
package Helper is
    procedure Public_Put_Line (S : String);
private
    procedure Private_Put_Line (S : String);
end Helper;
```
Learning Ada

Listing 15: helper.adb

```ada
with Ada.Text_IO;

package body Helper is

procedure Public_Put_Line (S : String) is
begin
  Ada.Text_IO.Put_Line (S);
end Public_Put_Line;

procedure Private_Put_Line (S : String) is
begin
  Ada.Text_IO.Put_Line (S);
end Private_Put_Line;

procedure Body_Put_Line (S : String) is
begin
  Ada.Text_IO.Put_Line (S);
end Body_Put_Line;

end Helper;
```

Listing 16: hello_world.adb

```ada
with Helper; use Helper;

procedure Hello_World is
begin
  Public_Put_Line ("hello, world!");
  Private_Put_Line ("hello, world!"); -- ERROR
  Body_Put_Line ("hello, world!"); -- ERROR
end Hello_World;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 148fd8101cc7241390675534f5e359c

Build output

hello_world.adb:6:04: error: "Private_Put_Line" is not visible
hello_world.adb:6:04: error: non-visible (private) declaration at helper.ads:4
hello_world.adb:7:04: error: "Body_Put_Line" is undefined
gprbuild: *** compilation phase failed

Note the different errors on the calls to the private and body versions of Put_Line. In the first case the compiler can locate the candidate procedure but it is illegal to call it from Hello_World, in the second case the compiler does not even know about any Body_Put_Line when compiling Hello_World since it only looks at the spec and not the body.

SPARK (and Ada) also allow defining a type in the private part of a package spec while simply declaring the type name in the public ("visible") part of the spec. This way, client code — i.e., code that with's the package — can use the type, typically through a public API, but have no access to how the type is implemented:

Listing 17: vault.ads

```ada
package Vault is
  type Data is private;
  function Get (X : Data) return Integer;
  procedure Set (X : out Data; Value : Integer);
```

(continues on next page)

78.3. Going Towards Encapsulation
private

    type Data is record
    Val : Integer;
    end record;
end Vault;

Listing 18: vault.adb

package body Vault is

    function Get (X : Data) return Integer is (X.Val);
    procedure Set (X : out Data; Value : Integer) is
    begin
        X.Val := Value;
    end Set;
end Vault;

Listing 19: information_system.ads

    with Vault;

    package Information_System is
        Archive : Vault.Data;
    end Information_System;

Listing 20: hacker.adb

    with Information_System;
    with Vault;

    procedure Hacker is
        V : Integer := Vault.Get (Information_System.Archive);
    begin
        Vault.Set (Information_System.Archive, V + 1);
        Information_System.Archive.Val := 0;  -- ERROR
    end Hacker;

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Conistency.Hacker
MD5: 065ed34dc727e2eb0bdc50a667cb1f78

Build output

hacker.adb:8:22: error: invalid prefix in selected component "Information_System._Archive"
gprbuild: *** compilation phase failed

Note that it is possible to declare a variable of type Vault.Data in package Information_System and to get/set it through its API in procedure Hacker, but not to directly access its Val field.
C's syntax is concise but also very permissive, which makes it easy to write programs whose effect is not what was intended. MISRA C contains guidelines to:

- clearly distinguish code from comments
- specially handle function parameters and result
- ensure that control structures are not abused

### 79.1 Distinguishing Code and Comments

The problem arises from block comments in C, starting with /* and ending with */. These comments do not nest with other block comments or with line comments. For example, consider a block comment surrounding three lines that each increase variable \( a \) by one:

```c
/*
++a;
++a;
++a; */
```

Now consider what happens if the first line is commented out using a block comment and the third line is commented out using a line comment (also known as a C++ style comment, allowed in C since C99):

```c
/* */
/* ++a; */
++a;
// ++a; */
```

The result of commenting out code that was already commented out is that the second line of code becomes live! Of course, the above example is simplified, but similar situations do arise in practice, which is the reason for MISRA C Directive 4.1 "Sections of code should not be 'commented out'". This is reinforced with Rules 3.1 and 3.2 from the section on "Comments" that forbid in particular the use of /* inside a comment like we did above.

These situations cannot arise in SPARK (or in Ada), as only line comments are permitted, using --:

```c
-- A := A + 1;
-- A := A + 1;
-- A := A + 1;
```

So commenting again the first and third lines does not change the effect:

```c
-- -- A := A + 1;
-- A := A + 1;
-- -- A := A + 1;
```
79.2 Specially Handling Function Parameters and Result

79.2.1 Handling the Result of Function Calls

It is possible in C to ignore the result of a function call, either implicitly or else explicitly by converting the result to `void`

```c
f();
(void)f();
```

This is particularly dangerous when the function returns an error status, as the caller is then ignoring the possibility of errors in the callee. Thus the MISRA C Directive 4.7: "If a function returns error information, then that error information shall be tested". In the general case of a function returning a result which is not an error status, MISRA C Rule 17.7 states that "The value returned by a function having non-void return type shall be used", where an explicit conversion to `void` counts as a use.

In SPARK, as in Ada, the result of a function call must be used, for example by assigning it to a variable or by passing it as a parameter, in contrast with procedures (which are equivalent to void-returning functions in C). SPARK analysis also checks that the result of the function is actually used to influence an output of the calling subprogram. For example, the first two calls to `F` in the following are detected as unused, even though the result of the function call is assigned to a variable, which is itself used in the second case:

```ada
package Fun is
  function F return Integer is (1);
end Fun;

procedure Use_F (Z : out Integer);

procedure Use_F (Z : out Integer) is
  X, Y : Integer;
begin
  X := F;
  Y := F;
  X := Y;
  Z := F;
end Use_F;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Func_Return
MD5: 4fc78b4136677d6338984ab8ccfa5cd1

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
use_f.adb:6:06: warning: unused assignment

(continues on next page)
Only the result of the third call is used to influence the value of an output of Use_F, here the output parameter Z of the procedure.

### 79.2.2 Handling Function Parameters

In C, function parameters are treated as local variables of the function. They can be modified, but these modifications won't be visible outside the function. This is an opportunity for mistakes. For example, the following code, which appears to swap the values of its parameters, has in reality no effect:

```c
void swap (int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

MISRAC Rule 17.8 prevents such mistakes by stating that "A function parameter should not be modified".

No such rule is needed in SPARK, since function parameters are only inputs so cannot be modified, and procedure parameters have a mode defining whether they can be modified or not. Only parameters of mode out or ada:in out can be modified — and these are prohibited from functions in SPARK — and their modification is visible at the calling site. For example, assigning to a parameter of mode in (the default parameter mode if omitted) results in compilation errors:

```ada
1 procedure Swap (X, Y : Integer);

2 procedure Swap (X, Y : Integer) is
3         Tmp : Integer := X;
4     begin
5         X := Y;  -- ERROR
6         Y := Tmp;  -- ERROR
7     end Swap;
```

Here is the output of AdaCore's GNAT compiler:

```ada
1. procedure Swap (X, Y : Integer) is
2.     Tmp : Integer := X;
3.     begin
```

(continues on next page)
The correct version of Swap in SPARK takes parameters of mode in out:

Listing 6: swap.ads

```ada
procedure Swap (X, Y : in out Integer);
```

Listing 7: swap.adb

```ada
procedure Swap (X, Y : in out Integer) is
  Tmp : constant Integer := X;
begin
  X := Y;
  Y := Tmp;
end Swap;
```

### Code block metadata

- **Project:** Courses.SPARC.For_The_MISRA_C_Syntactic_Guarantees.Swap
- **MD5:** c983a229fc5a69db5dbb85f499a91b325

### Prover output

- Phase 1 of 2: generation of Global contracts ...
- Phase 2 of 2: analysis of data and information flow ...

### 79.3 Ensuring Control Structures Are Not Abused

The previous issue (ignoring the result of a function call) is an example of a control structure being abused, due to the permissive syntax of C. There are many such examples, and MISRA C contains a number of guidelines to prevent such abuse.

### 79.3.1 Preventing the Semicolon Mistake

Because a semicolon can act as a statement, and because an if-statement and a loop accept a simple statement (possibly only a semicolon) as body, inserting a single semicolon can completely change the behavior of the code:

```c
int func() {
  if (0)
    return 1;
  while (1)
    return 0;
}
```
As written, the code above returns with status 0. If a semicolon is added after the first line
(if (0);), then the code returns with status 1. If a semicolon is added instead after the
third line (while (1);), then the code does not return. To prevent such surprises, MISRA C
Rule 15.6 states that "The body of an iteration-statement or a selection-statement shall be
a compound statement" so that the code above must be written:

```c
int func() {
    if (0) {
        return 1;
    }
    while (1) {
        return 0;
    }
    return 0;
}
```

Note that adding a semicolon after the test of the if or while statement has the same
effect as before! But doing so would violate MISRA C Rule 15.6.

In SPARK, the semicolon is not a statement by itself, but rather a marker that terminates
a statement. The null statement is an explicit null;, and all blocks of statements have explicit begin and end markers, which prevents mistakes that are possible in C. The SPARK
(also Ada) version of the above C code is as follows:

```ada
function Func return Integer is
begin
    if False then
        return 1;
    end if;
    while True loop
        return 0;
    end loop;
    return 0;
end Func;
```

### Code block metadata

**Project:** Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Semicolon  
**MD5:** 34fc5967c41d337aada17429ee5f44e9

### Prover output

Phase 1 of 2: generation of Global contracts ...  
Phase 2 of 2: analysis of data and information flow ...  
func.adb:3:04: warning: statement has no effect  
func.adb:4:07: warning: this statement is never reached
79.3.2 Avoiding Complex Switch Statements

Switch statements are well-known for being easily misused. Control can jump to any case section in the body of the switch, which in C can be before any statement contained in the body of the switch. At the end of the sequence of statements associated with a case, execution continues with the code that follows unless a break is encountered. This is a recipe for mistakes, and MISRA C enforces a simpler well-formed syntax for switch statements defined in Rule 16.1: "All switch statements shall be well-formed".

The other rules in the section on "Switch statements" go on detailing individual consequences of Rule 16.1. For example Rule 16.3 forbids the fall-through from one case to the next: "An unconditional break statement shall terminate every switch-clause". As another example, Rule 16.4 mandates the presence of a default case to handle cases not taken into account explicitly: "Every switch statement shall have a default label".

The analog of the C switch statements in SPARK (and in Ada) is the case statement. This statement has a simpler and more robust structure than the C switch, with control automatically exiting after one of the case alternatives is executed, and the compiler checking that the alternatives are disjoint (like in C) and complete (unlike in C). So the following code is rejected by the compiler:

Listing 10: sign_domain.ads

```ada
package Sign_Domain is

  type Sign is (Negative, Zero, Positive);

  function Opposite (A : Sign) return Sign is
    (case A is
      when Negative => Positive,
      when Positive => Negative);

  function Multiply (A, B : Sign) return Sign is
    (case A is
      when Negative => Opposite (B),
      when Zero | Positive => Zero,
      when Positive => B); -- ERROR

  procedure Get_Sign (X : Integer; S : out Sign);

end Sign_Domain;
```

Listing 11: sign_domain.adb

```ada
package body Sign_Domain is

  procedure Get_Sign (X : Integer; S : out Sign) is
    begin
      case X is
        when 0 => S := Zero;
        when others => S := Negative; -- ERROR
        when 1 .. Integer'Last => S := Positive;
      end case;
    end Get_Sign;

end Sign_Domain;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Case_Statement
MD5: d345a4d23b5b2402f8bd103e5e550a3b
Learning Ada

Build output

sign_domain.adb:7:15: error: the choice "others" must appear alone and last
sign_domain.ads:6:07: error: missing case value: "Zero"
sign_domain.ads:14:15: error: duplication of choice value: "Positive" at line 13
gprbuild: *** compilation phase failed

The error in function Opposite is that the when choices do not cover all values of the target expression. Here, A is of the enumeration type Sign, so all three values of the enumeration must be covered.

The error in function Multiply is that Positive is covered twice, in the second and the third alternatives. This is not allowed.

The error in procedure Get_Sign is that the others choice (the equivalent of C default case) must come last. Note that an others choice would be useless in Opposite and Multiply, as all Sign values are covered.

Here is a correct version of the same code:

Listing 12: sign_domain.ads

```ada
package Sign_Domain is
  type Sign is (Negative, Zero, Positive);
  function Opposite (A : Sign) return Sign is
    (case A is
      when Negative => Positive,
      when Zero => Zero,
      when Positive => Negative);
  function Multiply (A, B : Sign) return Sign is
    (case A is
      when Negative => Opposite (B),
      when Zero => Zero,
      when Positive => B);
  procedure Get_Sign (X : Integer; S : out Sign);
end Sign_Domain;
```

Listing 13: sign_domain.adb

```ada
package body Sign_Domain is
  procedure Get_Sign (X : Integer; S : out Sign) is
    begin
      case X is
        when 0 => S := Zero;
        when 1 .. Integer'Last => S := Positive;
        when others => S := Negative;
      end case;
    end Get_Sign;
end Sign_Domain;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Case_Statement
MD5: 1c99fc53d2d2c0ddbea5e5b0a6c5746

79.3. Ensuring Control Structures Are Not Abused 1625
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
sign_domain.ads:17:37: info: initialization of "S" proved

79.3.3 Avoiding Complex Loops

Similarly to C switches, for-loops in C can become unreadable. MISRA C thus enforces a simpler well-formed syntax for for-loops, defined in Rule 14.2: "A for loop shall be well-formed". The main effect of this simplification is that for-loops in C look like for-loops in SPARK (and in Ada), with a loop counter that is incremented or decremented at each iteration. Section 8.14 defines precisely what a loop counter is:

1. It has a scalar type;
2. Its value varies monotonically on each loop iteration; and
3. It is used in a decision to exit the loop.

In particular, Rule 14.2 forbids any modification of the loop counter inside the loop body. Here's the example used in MISRA C:2012 to illustrate this rule:

bool_t flag = false;
for ( int16_t i = 0; ( i < 5 ) && !flag; i++ )
{
    if ( C )
    {
        flag = true; /* Compliant - allows early termination of loop */
    }
    i = i + 3; /* Non-compliant - altering the loop counter */
}

The equivalent SPARK (and Ada) code does not compile, because of the attempt to modify the value of the loop counter:

Listing 14: well_formed_loop.adb

procedure Well_Formed_Loop ( C : Boolean) is
    Flag : Boolean := False;
begin
    for I in 0 .. 4 loop
        exit when not Flag;
        if C then
            Flag := True;
        end if;
        I := I + 3; -- ERROR
    end loop;
    end Well_Formed_Loop;

Code block metadata

Project: Courses.SPARX.For_The_MISRA_C_DEV.Syntactic_Guarantees.Well_Formed_Loop
MD5: 842564c961aa018e03f81439995ec

Build output

well_formed_loop.adb:11:07: error: assignment to loop parameter not allowed
gprbuild: *** compilation phase failed
Removing the problematic line leads to a valid program. Note that the additional condition being tested in the C for-loop has been moved to a separate exit statement at the start of the loop body.

SPARK (and Ada) loops can increase (or, with explicit syntax, decrease) the loop counter by 1 at each iteration.

```ada
for I in reverse 0 .. 4 loop
    ...
    -- Successive values of I are 4, 3, 2, 1, 0
end loop;
```

SPARK loops can iterate over any discrete type; i.e., integers as above or enumerations:

```ada
type Sign is (Negative, Zero, Positive);
for S in Sign loop
    ...
end loop;
```

### 79.3.4 Avoiding the Dangling Else Issue

C does not provide a closing symbol for an if-statement. This makes it possible to write the following code, which appears to try to return the absolute value of its argument, while it actually does the opposite:

Listing 15: main.c

```c
#include <stdio.h>

int absval (int x) {
    int result = x;
    if (x == 0)
        result = 0;
    else
        result = -x;
    return result;
}

int main() {
    printf("absval(5) = %d\n", absval(5));
    printf("absval(0) = %d\n", absval(0));
    printf("absval(-10) = %d\n", absval(-10));
}
```

The warning issued by GCC or LLVM with option -Wdangling-else (implied by -Wall) gives a clue about the problem: although the else branch is written as though it completes the outer if-statement, in fact it completes the inner if-statement.
MISRA C Rule 15.6 avoids the problem: "The body of an iteration-statement or a selection-statement shall be a compound statement". That's the same rule as the one shown earlier for Preventing the Semicolon Mistake (page 1622). So the code for absval must be written:

Listing 16: main.c

```c
#include <stdio.h>

int absval (int x) {
    int result = x;
    if (x >= 0) {
        if (x == 0) {
            result = 0;
        }
    } else {
        result = -x;
    }
    return result;
}

int main() {
    printf("absval(5) = %d\n", absval(5));
    printf("absval(0) = %d\n", absval(0));
    printf("absval(-10) = %d\n", absval(-10));
}
```

In SPARK (as in Ada), each if-statement has a matching end marker end if; so the dangling-else problem cannot arise. The above C code is written as follows:

Listing 17: absval.ads

```ada
function Absval (X : Integer) return Integer;
```

Listing 18: absval.adb

```ada
function Absval (X : Integer) return Integer is
    Result : Integer := X;
    begin
        if X >= 0 then
            if X = 0 then
                Result := 0;
            end if;
        else
            Result := -X;
        end if;
    return Result;
end Absval;
```

which has the expected behavior.
Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
absval.adb:9:17: medium: overflow check might fail, cannot prove upper bound for -X [reason for check: result of negation must fit in a 32-bits machine integer].
[possible fix: add precondition (-X in Integer) to subprogram at absval.ads:1]
gnatprove: unproved check messages considered as errors

Interestingly, SPARK analysis detects here that the negation operation on line 9 might overflow. That's an example of runtime error detection which will be covered in the chapter on Detecting Undefined Behavior (page 1665).
Annex C of MISRA C:2012 summarizes the problem succinctly:

"ISO C may be considered to exhibit poor type safety as it permits a wide range of implicit type conversions to take place. These type conversions can compromise safety as their implementation-defined aspects can cause developer confusion."

The most severe consequences come from inappropriate conversions involving pointer types, as they can cause memory safety violations. Two sections of MISRA C are dedicated to these issues: "Pointertype conversions" (9 rules) and "Pointers and arrays" (8 rules).

Inappropriate conversions between scalar types are only slightly less severe, as they may introduce arbitrary violations of the intended functionality. MISRA C has gone to great lengths to improve the situation, by defining a stricter type system on top of the C language. This is described in Appendix D of MISRA C:2012 and in the dedicated section on "The essential type model" (8 rules).

### 80.1 Enforcing Strong Typing for Pointers

Pointers in C provide a low-level view of the addressable memory as a set of integer addresses. To write at address 42, just go through a pointer:

```c
int main() {
    int *p = 42;
    *p = 0;
    return 0;
}
```

Running this program is likely to hit a segmentation fault on an operating system, or to cause havoc in an embedded system, both because address 42 will not be correctly aligned on a 32-bit or 64-bit machine and because this address is unlikely to correspond to valid addressable data for the application. The compiler might issue a helpful warning on the above code (with option -Wint-conversion implied by -Wall in GCC or LLVM), but note that the warning disappears when explicitly converting value 42 to the target pointer type, although the problem is still present.

Beyond their ability to denote memory addresses, pointers are also used in C to pass references as inputs or outputs to function calls, to construct complex data structures with indirection or sharing, and to denote arrays of elements. Pointers are thus at once pervasive, powerful and fragile.
80.1.1 Pointers Are Not Addresses

In an attempt to rule out issues that come from direct addressing of memory with pointers, MISRA C states in Rule 11.4 that "A conversion should not be performed between a pointer to object and an integer type". As this rule is classified as only Advisory, MISRA C completes it with two Required rules:

- Rule 11.6: "A cast shall not be performed between pointer to void and an arithmetic type"
- Rule 11.7: "A cast shall not be performed between pointer to object and a non-integer arithmetic type"

In Ada, pointers are not addresses, and addresses are not integers. An opaque standard type `System.Address` is used for addresses, and conversions to/from integers are provided in a standard package `System.Storage_Elements`. The previous C code can be written as follows in Ada:

```ada
with System;  
with System.Storage_Elements;  

procedure Pointer is  
    A: constant System.Address := System.Storage_Elements.To_Address (42);  
    M: aliased Integer with Address => A;  
    P: constant access Integer := M'Access;  
    begin  
    P.all := 0;  
end Pointer; 
```

The integer value 42 is converted to a memory address `A` by calling `System.Storage_Elements.To_Address`, which is then used as the address of integer variable `M`. The pointer variable `P` is set to point to `M` (which is allowed because `M` is declared as `aliased`). Ada requires more verbiage than C:

- The integer value 42 must be explicitly converted to type `Address`
- To get a pointer to a declared variable such as `M`, the declaration must be marked as `aliased`

The added syntax helps first in making clear what is happening and, second, in ensuring that a potentially dangerous feature (assigning to a value at a specific machine address) is not used inadvertently.

The above example is legal in SPARK, but the SPARK analysis tool issues warnings as it cannot control how the program or its environment may update the memory cell at address 42.
### 80.1.2 Pointers Are Not References

Passing parameters by reference is critical for efficient programs, but the absence of references distinct from pointers in C incurs a serious risk. Any parameter of a pointer type can be copied freely to a variable whose lifetime is longer than the object pointed to, a problem known as "dangling pointers". MISRA C forbids such uses in Rule 18.6: "The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist". Unfortunately, enforcing this rule is difficult, as it is undecidable.

In SPARK, parameters can be passed by reference, but no pointer to the parameter can be stored past the return point of the function, which completely solves this issue. In fact, the decision to pass a parameter by copy or by reference rests in many cases with the compiler, but such compiler dependency has no effect on the functional behavior of a SPARK program. In the example below, the compiler may decide to pass parameter P of procedure Rotate_X either by copy or by reference, but regardless of the choice the postcondition of Rotate_X will hold: the final value of P will be modified by rotation around the X axis.

**Listing 3: geometry.ads**

```ada
package Geometry is
    type Point_3D is record
        X, Y, Z : Float;
    end record;

    procedure Rotate_X (P : in out Point_3D) with
        Post => P = P'.Old'Update (Y => P.Z'Old, Z => -P.Y'Old);
end Geometry;
```

**Listing 4: geometry.adb**

```ada
package body Geometry is
    procedure Rotate_X (P : in out Point_3D) is
        Tmp : constant Float := P.Y;
    begin
        P.Y := P.Z;
        P.Z := -Tmp;
    end Rotate_X;
end Geometry;
```

### Code block metadata

- **Project**: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Geometry
- **MD5**: d3801cf1413887fdd5fff8b6b86b7742

### Prover output

- **Phase 1 of 2**: generation of Global contracts ...
- **Phase 2 of 2**: flow analysis and proof ...
- **geometry.ads**:8:14: info: postcondition proved

SPARK’s analysis tool can mathematically prove that the postcondition is true.
80.1.3 Pointers Are Not Arrays

The greatest source of vulnerabilities regarding pointers is their use as substitutes for arrays. Although the C language has a syntax for declaring and accessing arrays, this is just a thin syntactic layer on top of pointers. Thus:

- Array access is just pointer arithmetic;
- If a function is to manipulate an array then the array's length must be separately passed as a parameter; and
- The program is susceptible to the various vulnerabilities originating from the confusion of pointers and arrays, such as buffer overflow.

Consider a function that counts the number of times a value is present in an array. In C, this could be written:

```c
#include <stdio.h>

int count(int *p, int len, int v) {
    int count = 0;
    while (len--) {
        if (*p++ == v) {
            count++;
        }
    }
    return count;
}

int main() {
    int p[5] = {0, 3, 9, 3, 3};
    int c = count(p, 5, 3);
    printf("value 3 is seen %d times in p\n", c);
    return 0;
}
```

Function `count` has no control over the range of addresses accessed from pointer `p`. The critical property that the `len` parameter is a valid length for an array of integers pointed to by parameter `p` rests completely with the caller of `count`, and `count` has no way to check that this is true.

To mitigate the risks associated with pointers being used for arrays, MISRA C contains eight rules in a section on "Pointers and arrays". These rules forbid pointer arithmetic (Rule 18.4) or, if this Advisory rule is not followed, require pointer arithmetic to stay within bounds (Rule 18.1). But, even if we rewrite the loop in count to respect all decidable MISRA C rules, the program's correctness still depends on the caller of count passing a correct value of `len`:
int count = 0;
for (int i = 0; i < len; i++) {
    if (p[i] == v) {
        count++;
    }
}
return count;

int main() {
    int p[5] = {0, 3, 9, 3, 3};
    int c = count(p, 5, 3);
    printf("value 3 is seen %d times in p\n", c);
    return 0;
}

Code block metadata
Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Arrays_MISRA_C
MD5: d04179de3f1e309541b3d88e53eb5e3a

Runtime output
value 3 is seen 3 times in p

The resulting code is more readable, but still vulnerable to incorrect values of parameter len passed by the caller of count, which violates undecidable MISRA C Rules 18.1 (pointer arithmetic should stay within bounds) and 1.3 (no undefined behavior). Contrast this with the same function in SPARK (and Ada):

Listing 7: types.ads
package Types is
type Int_Array is array (Positive range <>) of Integer;
end Types;

Listing 8: count.ads
with Types; use Types;

function Count (P : Int_Array; V : Integer) return Natural;

Listing 9: count.adb
function Count (P : Int_Array; V : Integer) return Natural is
    Count : Natural := 0;
begin
    for I in P'Range loop
        if P(I) = V then
            Count := Count + 1;
        end if;
    end loop;
    return Count;
end Count;

Listing 10: test_count.adb
with Ada.Text_IO; use Ada.Text_IO;
with Types; use Types;
with Count;

80.1. Enforcing Strong Typing for Pointers
procedure Test_Count is
  P : constant Int_Array := (0, 3, 9, 3, 3);
  C : constant Integer := Count (P, 3);
begin
  Put_Line ("value 3 is seen" & C'Img & " times in p");
end Test_Count;

The array parameter P is not simply a homogeneous sequence of Integer values. The compiler must represent P so that its lower and upper bounds (P'First and P'Last) and thus also its length (P'Length) can be retrieved. Function Count can simply loop over the range of valid array indexes P'First .. P'Last (or P'Range for short). As a result, function Count can be verified in isolation to be free of vulnerabilities such as buffer overflow, as it does not depend on the values of parameters passed by its callers. In fact, we can go further in SPARK and show that the value returned by Count is no greater than the length of parameter P by stating this property in the postcondition of Count and asking the SPARK analysis tool to prove it:

Listing 11: types.ads

package Types is
type Int_Array is array (Positive range <>) of Integer;
end Types;

Listing 12: count.ads

with Types; use Types;

function Count (P : Int_Array; V : Integer) return Natural with
Post => Count'Result <= P'Length;

Listing 13: count.adb

function Count (P : Int_Array; V : Integer) return Natural is
  Count : Natural := 0;
begin
  for I in P'Range loop
    pragma Loop_Invariant (Count <= I - P'First);
    if P (I) = V then
      Count := Count + 1;
    end if;
  end loop;
  return Count;
end Count;

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Arrays_Ada
MD5: 82e9d18d4b8ad8aa87ca8520bd7b830c

Runtime output

value 3 is seen 3 times in p
Learning Ada

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
count.adb:6:30: info: loop invariant preservation proved
count.adb:6:30: info: loop invariant initialization proved
count.adb:6:41: info: overflow check proved
count.adb:8:25: info: overflow check proved
count.ads:4:11: info: postcondition proved
count.ads:4:28: info: range check proved

The only help that SPARK analysis required from the programmer, in order to prove the postcondition, is a loop invariant (a special kind of assertion) that reflects the value of Count at each iteration.

80.1.4 Pointers Should Be Typed

The C language defines a special pointer type `void*` that corresponds to an untyped pointer. It is legal to convert any pointer type to and from `void*`, which makes it a convenient way to simulate C++ style templates. Consider the following code which indirectly applies `assign_int` to integer i and `assign_float` to floating-point f by calling `assign` on both:

Listing 14: main.c

```c
#include <stdio.h>

void assign_int (int *p) {
  *p = 42;
}

void assign_float (float *p) {
  *p = 42.0;
}

typed void (*assign_fun)(void *p);

void assign(assign_fun fun, void *p) {
  fun(p);
}

int main() {
  int i;
  float f;
  assign((assign_fun)&assign_int, &i);
  assign((assign_fun)&assign_float, &f);
  printf("i = %d; f = %f\n", i, f);
}
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Typed_Pointers_C
MD5: fc00ba9eb97640037488569347591cc2

Runtime output

i = 42; f = 42.000000

The references to the variables i and f are implicitly converted to the `void*` type as a way to apply `assign` to any second parameter p whose type matches the argument type of its first argument fun. The use of an untyped argument means that the responsibility for the correct typing rests completely with the programmer. Swap i and f in the calls to
Learning Ada

assign and you still get a compilable program without warnings, that runs and produces completely bogus output:

\[
i = 1109917696; f = 0.000000
\]

instead of the expected:

\[
i = 42; f = 42.000000
\]

Generics in SPARK (and Ada) can implement the desired functionality in a fully typed way, with any errors caught at compile time, where procedure Assign applies its parameter procedure Initialize to its parameter V:

```
Listing 15: assign.ads

generic
  type T is private;
  with procedure Initialize (V : out T);
procedure Assign (V : out T);
```

```
Listing 16: assign.adb

procedure Assign (V : out T) is
begin
  Initialize (V);
end Assign;
```

```
Listing 17: apply_assign.adb

with Ada.Text_IO; use Ada.Text_IO;
with Assign;

procedure Apply_Assign is
  procedure Assign_Int (V : out Integer) is
  begin
    V := 42;
  end Assign_Int;

  procedure Assign_Float (V : out Float) is
  begin
    V := 42.0;
  end Assign_Float;

  procedure Assign_I is new Assign (Integer, Assign_Int);
  procedure Assign_F is new Assign (Float, Assign_Float);

  I : Integer;
  F : Float;
begin
  Assign_I (I);
  Assign_F (F);
  Put_Line ("I =" & I'Img & "; F =" & F'Img);
end Apply_Assign;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Typed_Pointers_Ada
MD5: af23d6f8a742676139aac38a385c7bf7

Runtime output
I = 42; F = 4.20000E+01

The generic procedure Assign must be instantiated with a specific type for T and a specific procedure (taking a single out parameter of this type) for Initialize. The procedure resulting from the instantiation applies to a variable of this type. So switching I and F above would result in an error detected by the compiler. Likewise, an instantiation such as the following would also be a compile-time error:

```
procedure Assign_I is new Assign (Integer, Assign_Float);
```

### 80.2 Enforcing Strong Typing for Scalars

In C, all scalar types can be converted both implicitly and explicitly to any other scalar type. The semantics is defined by rules of promotion and conversion, which can confuse even experts. One example was noted earlier, in the Preface (page 1609). Another example appears in an article introducing a safe library for manipulating scalars by Microsoft expert David LeBlanc. In its conclusion, the author acknowledges the inherent difficulty in understanding scalar type conversions in C, by showing an early buggy version of the code to produce the minimum signed integer:

```
return (T)(1 << (BitCount() - 1));
```

The issue here is that the literal 1 on the left-hand side of the shift is an int, so on a 64-bit machine with 32-bit int and 64-bit type T, the above is shifting 32-bit value 1 by 63 bits. This is a case of undefined behavior, producing an unexpected output with the Microsoft compiler. The correction is to convert the first literal 1 to T before the shift:

```
return (T)((T)1 << (BitCount() - 1));
```

Although he'd asked some expert programmers to review the code, no one found this problem.

To avoid these issues as much as possible, MISRA C defines its own type system on top of C types, in the section on "The essential type model" (eight rules). These can be seen as additional typing rules, since all rules in this section are decidable, and can be enforced at the level of a single translation unit. These rules forbid in particular the confusing cases mentioned above. They can be divided into three sets of rules:

- restricting operations on types
- restricting explicit conversions
- restricting implicit conversions

### 80.2.1 Restricting Operations on Types

Apart from the application of some operations to floating-point arguments (the bitwise, mod and array access operations) which are invalid and reported by the compiler, all operations apply to all scalar types in C. MISRA C Rule 10.1 constrains the types on which each operation is possible as follows.
Arithmetic Operations on Arithmetic Types

Adding two Boolean values, or an Apple and an Orange, might sound like a bad idea, but it is easily done in C:

Listing 18: main.c

```c
#include <stdbool.h>
#include <stdio.h>

int main() {
    bool b1 = true;
    bool b2 = false;
    bool b3 = b1 + b2;

typedef enum {Apple, Orange} fruit;
    fruit f1 = Apple;
    fruit f2 = Orange;
    fruit f3 = f1 + f2;

    printf("b3 = %d; f3 = %d\n", b3, f3);

    return 0;
}
```

No error from the compiler here. In fact, there is no undefined behavior in the above code. Variables b3 and f3 both end up with value 1. Of course it makes no sense to add Boolean or enumerated values, and thus MISRA C Rule 18.1 forbids the use of all arithmetic operations on Boolean and enumerated values, while also forbidding most arithmetic operations on characters. That leaves the use of arithmetic operations for signed or unsigned integers as well as floating-point types and the use of modulo operation % for signed or unsigned integers.

Here's an attempt to simulate the above C code in SPARK (and Ada):

Listing 19: bad_arith.ads

```ada
package Bad_Arith is

    B1 : constant Boolean := True;
    B2 : constant Boolean := False;
    B3 : constant Boolean := B1 + B2;

type Fruit is (Apple, Orange);
    F1 : constant Fruit := Apple;
    F2 : constant Fruit := Orange;
    F3 : constant Fruit := F1 + F2;

end Bad_Arith;
```

Code block metadata
It is possible, however, to get the predecessor of a Boolean or enumerated value with \texttt{Value}'\texttt{Pred} and its successor with \texttt{Value}'\texttt{Succ}, as well as to iterate over all values of the type:

\begin{verbatim}
Listing 20: ok_arith.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Ok_Arith
is
  B1 : constant Boolean := False;
  B2 : constant Boolean := Boolean'Pred (B1);
  B3 : constant Boolean := Boolean'Pred (B2);

  type Fruit is (Apple, Orange);
  F1 : constant Fruit := Apple;
  F2 : constant Fruit := Fruit'Succ (F1);
  F3 : constant Fruit := Fruit'Succ (F2);

begin
  pragma Assert (B1 = B3);
  pragma Assert (F1 = F3);

  for B in Boolean loop
    Put_Line (B'Img);
  end loop;

  for F in Fruit loop
    Put_Line (F'Img);
  end loop;

end Ok_Arith;
\end{verbatim}

80.2. Enforcing Strong Typing for Scalars
Learning Ada

Boolean Operations on Boolean

"Two bee or not two bee? Let's C":

Listing 21: main.c

```c
#include <stdbool.h>
#include <stdio.h>

int main() {
    typedef enum {Ape, Bee, Cat} Animal;
    bool answer = (2 * Bee) || ! (2 * Bee);
    printf("two bee or not two bee? %d\n", answer);
    return 0;
}
```

Runtime output
two bee or not two bee? 1

The answer to the question posed by Shakespeare's Hamlet is 1, since it reduces to A or not A and this is true in classical logic.

As previously noted, MISRA C forbids the use of the multiplication operator with an operand of an enumerated type. Rule 18.1 also forbids the use of Boolean operations "and", "or", and "not" (&&, ||, !, respectively, in C) on anything other than Boolean operands. It would thus prohibit the Shakespearian code above.

Below is an attempt to express the same code in SPARK (and Ada), where the Boolean operators are and, or, and not. The and and or operators evaluate both operands, and the language also supplies short-circuit forms that evaluate the left operand and only evaluate the right operand when its value may affect the result.

Listing 22: bad_hamlet.ads

```ada
package Bad_Hamlet is
    type Animal is (Ape, Bee, Cat);
    Answer : Boolean := 2 * Bee or not 2 * Bee; -- Illegal
end Bad_Hamlet;
```

Build output
bad_hamlet.ads:3:28: error: expected type universal integer
bad_hamlet.ads:3:28: error: found type "Animal" defined at line 2
bad_hamlet.ads:3:43: error: expected a modular type
bad_hamlet.ads:3:43: error: found type "Animal" defined at line 2
gprbuild: *** compilation phase failed

As expected, the compiler rejects this code. There is no available * operation that works on an enumeration type, and likewise no available or or not operation.
Bitwise Operations on Unsigned Integers

Here's a genetic engineering example that combines a Bee with a Dog to produce a Cat, by manipulating the atomic structure (the bits in its representation):

Listing 23: main.c

```c
#include <stdbool.h>
#include <assert.h>

int main()
{
    typedef enum {Ape, Bee, Cat, Dog} Animal;
    Animal mutant = Bee ^ Dog;
    assert (mutant == Cat);
    return 0;
}
```

This algorithm works by accessing the underlying bitwise representation of Bee and Dog (0x01 and 0x03, respectively) and, by applying the exclusive-or operator ^, transforming it into the underlying bitwise representation of a Cat (0x02). While powerful, manipulating the bits in the representation of values is best reserved for unsigned integers as illustrated in the book *Hacker's Delight*[^344]. MISRA C Rule 18.1 thus forbids the use of all bitwise operations on anything but unsigned integers.

Below is an attempt to do the same in SPARK (and Ada). The bitwise operators are and, or, xor, and not, and the related bitwise functions are Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left and Rotate_Right:

Listing 24: bad_genetics.ads

```ada
package Bad_Genetics is
    type Animal is (Ape, Bee, Cat, Dog);
    Mutant : Animal := Bee xor Dog; -- ERROR
    pragma Assert (Mutant = Cat);
end Bad_Genetics;
```

The declaration of Mutant is illegal, since the xor operator is only available for Boolean and unsigned integer (modular) values; it is not available for Animal. The same restriction applies to the other bitwise operators listed above. If we really wanted to achieve the effect of the above code in legal SPARK (or Ada), then the following approach will work (the type Unsigned_8 is an 8-bit modular type declared in the predefined package Interfaces).


80.2. Enforcing Strong Typing for Scalars
Listing 25: unethical_genetics.ads

```
with Interfaces; use Interfaces;
package Unethical_Genetics is
  type Animal is (Ape, Bee, Cat, Dog);
  A : constant array (Animal) of Unsigned_8 :=
      (Animal'Pos (Ape), Animal'Pos (Bee),
       Animal'Pos (Cat), Animal'Pos (Dog));
  Mutant : Animal := Animal'Val (A (Bee) xor A (Dog));
pragma Assert (Mutant = Cat);
end Unethical_Genetics;
```

80.2.2 Restricting Explicit Conversions

A simple way to bypass the restrictions of Rule 10.1 is to explicitly convert the arguments of an operation to a type that the rule allows. While it can often be useful to cast a value from one type to another, many casts that are allowed in C are either downright errors or poor replacements for clearer syntax.

One example is to cast from a scalar type to Boolean. A better way to express \((\text{bool})x\) is to compare \(x\) to the zero value of its type: \(x \neq 0\) for integers, \(x \neq 0.0\) for floats, \(x \neq '0'\) for characters, \(x \neq \text{Enum}\) where \text{Enum} is the first enumerated value of the type. Thus, MISRA C Rule 10.5 advises avoiding casting non-Boolean values to Boolean.

Rule 10.5 also advises avoiding other casts that are, at best, obscure:

- from a Boolean to any other scalar type
- from a floating-point value to an enumeration or a character
- from any scalar type to an enumeration

The rules are not symmetric, so although a float should not be cast to an enum, casting an enum to a float is allowed. Similarly, although it is advised to not cast a character to an enum, casting an enum to a character is allowed.

The rules in SPARK are simpler. There are no conversions between numeric types (integers, fixed-point and floating-point) and non-numeric types (such as Boolean, Character, and other enumeration types). Conversions between different non-numeric types are limited to those that make semantic sense, for example between a derived type and its parent type. Any numeric type can be converted to any other numeric type, with precise rules for rounding/truncating values when needed and run-time checking that the converted value is in the range associated with the target type.
80.2.3 Restricting Implicit Conversions

Rules 10.1 and 10.5 restrict operations on types and explicit conversions. That's not enough to avoid problematic C programs; a program violating one of these rules can be expressed using only implicit type conversions. For example, the Shakespearian code in section Boolean Operations on Boolean (page 1642) can be reformulated to satisfy both Rules 10.1 and 10.5:

Listing 26: main.c

```c
#include <stdbool.h>
#include <stdio.h>

int main() {
    typedef enum {Ape, Bee, Cat} Animal;
    int b = Bee;
    bool t = 2 * b;
    bool answer = t || !t;
    printf("two bee or not two bee? %d\n", answer);
    return 0;
}
```

Here, we're implicitly converting the enumerated value Bee to an int, and then implicitly converting the integer value \(2 \times b\) to a Boolean. This does not violate 10.1 or 10.5, but it is prohibited by MISRA C Rule 10.3: "The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category".

Rule 10.1 also does not prevent arguments of an operation from being inconsistent, for example comparing a floating-point value and an enumerated value. But MISRA C Rule 10.4 handles this situation: "Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category".

In addition, three rules in the "Composite operators and expressions" section avoid common mistakes related to the combination of explicit/implicit conversions and operations.

The rules in SPARK (and Ada) are far simpler: there are no implicit conversions! This applies both between types of a different essential type category as MISRA C puts it, as well as between types that are structurally the same but declared as different types.

Listing 27: bad_conversions.adb

```ada
procedure Bad_Conversions is
    pragma Warnings (Off);
    F : Float := 0.0;
    I : Integer := 0;
    type Animal is (Ape, Bee, Cat);
    type My_Animal is new Animal; -- derived type
    A : Animal := Cat;
    M : My_Animal := Bee;
    B : Boolean := True;
    C : Character := 'a';
begin
    F := I; -- ERROR
```

(continues on next page)
Learning Ada

(continued from previous page)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>I := A; -- ERROR</td>
</tr>
<tr>
<td>14</td>
<td>A := B; -- ERROR</td>
</tr>
<tr>
<td>15</td>
<td>M := A; -- ERROR</td>
</tr>
<tr>
<td>16</td>
<td>B := C; -- ERROR</td>
</tr>
<tr>
<td>17</td>
<td>C := F; -- ERROR</td>
</tr>
</tbody>
</table>

18 end Bad_Conversions;

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Implicit_Conversion_Bad_Ada
MD5: f10b50048595df0b4ed77c06a7f08412

Build output

bad_conversions.adb:12:09: error: expected type "Standard.Float"
bad_conversions.adb:12:09: error: found type "Standard.Integer"
bad_conversions.adb:13:09: error: found type "Animal" defined at line 5
bad_conversions.adb:14:09: error: expected type "Animal" defined at line 5
bad_conversions.adb:14:09: error: found type "Standard.Boolean"
bad_conversions.adb:15:09: error: expected type "My_Animal" defined at line 6
bad_conversions.adb:15:09: error: found type "Animal" defined at line 5
bad_conversions.adb:16:09: error: found type "Standard.Character"
bad_conversions.adb:17:09: error: expected type "Standard.Character"
bad_conversions.adb:17:09: error: found type "Standard.Float"
gprbuild: *** compilation phase failed

The compiler reports a mismatch on every statement in the above procedure (the declarations are all legal).

Adding explicit conversions makes the assignments to F and M valid, since SPARK (and Ada) allow conversions between numeric types and between a derived type and its parent type, but all other conversions are illegal:

Listing 28: bad_conversions.adb

procedure Bad_Conversions is
pragma Warnings (Off);
F : Float := 0.0;
I : Integer := 0;
type Animal is (Ape, Bee, Cat);
type My_Animal is new Animal; -- derived type
A : Animal := Cat;
M : My_Animal := Bee;
B : Boolean := True;
C : Character := 'a';
begn
F := Float (I); -- OK
I := Integer (A); -- ERROR
A := Animal (B); -- ERROR
M := My_Animal (A); -- OK
B := Boolean (C); -- ERROR
C := Character (F); -- ERROR
end Bad_Conversions;

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Implicit_Conversion_Bad_Ada
MD5: 4d3f6a8629d51f27b6628dae5fc7b680

1646 Chapter 80. Enforcing Strong Typing
Although an enumeration value cannot be converted to an integer (or vice versa) either implicitly or explicitly, SPARK (and Ada) provide functions to obtain the effect of a type conversion. For any enumeration type \( T \), the function \( T'\text{Pos}(e) \) takes an enumeration value from type \( T \) and returns its relative position as an integer, starting at 0. For example, \( \text{Animal}'\text{Pos}(\text{Bee}) \) is 1, and \( \text{Boolean}'\text{Pos}(\text{False}) \) is 0. In the other direction, \( T'\text{Val}(n) \), where \( n \) is an integer, returns the enumeration value in type \( T \) at relative position \( n \). If \( n \) is negative or greater than \( T'\text{Pos}(T'\text{Last}) \) then a run-time exception is raised.

Hence, the following is valid SPARK (and Ada) code; \texttt{Character} is defined as an enumeration type:

```
Listing 29: ok_conversions.adb

procedure Ok_Conversions is
   pragma Warnings (Off);
   F : Float := 0.0;
   I : Integer := 0;
   type Animal is (Ape, Bee, Cat);
   type My_Animal is new Animal;
begin
   F := Float (I);
   I := Animal'Pos (A);
   I := My_Animal'Pos (M);
   I := Boolean'Pos (B);
   I := Character'Pos (C);
   I := Integer (F);
   A := Animal'Val (2);
end Ok_Conversions;
```
As with most programming languages, C does not require that variables be initialized at their declaration, which makes it possible to unintentionally read uninitialized data. This is a case of undefined behavior, which can sometimes be used to attack the program.

### 81.1 Detecting Reads of Uninitialized Data

MISRA C attempts to prevent reads of uninitialized data in a specific section on "Initialization", containing five rules. The most important is Rule 9.1: "The value of an object with automatic storage duration shall not be read before it has been set". The first example in the rule is interesting, as it shows a non-trivial (and common) case of conditional initialization, where a function \( f \) initializes an output parameter \( p \) only in some cases, and the caller \( g \) of \( f \) ends up reading the value of the variable \( u \) passed in argument to \( f \) in cases where it has not been initialized:

```c
#include <stdint.h>

void f ( int b, uint16_t *p );
```

```c
#include "f.h"

void f ( int b, uint16_t *p )
{
    if ( b )
    {
        *p = 3U;
    }
}
```

```c
#include <stdint.h>
#include "f.h"

static void g ( void )
{
    uint16_t u;
    f ( 0, &u );
    if ( u == 3U )
```

(continues on next page)
Detecting the violation of Rule 9.1 can be arbitrarily complex, as the program points corresponding to a variable’s initialization and read can be separated by many calls and conditions. This is one of the undecidable rules, for which most MISRA C checkers won’t detect all violations.

In SPARK, the guarantee that all reads are to initialized data is enforced by the SPARK analysis tool, GNATprove, through what is referred to as flow analysis. Every subprogram is analyzed separately to check that it cannot read uninitialized data. To make this modular analysis possible, SPARK programs need to respect the following constraints:

- all inputs of a subprogram should be initialized on subprogram entry
- all outputs of a subprogram should be initialized on subprogram return

Hence, given the following code translated from C, GNATprove reports that function F might not always initialize output parameter P:

Listing 4: init.ads

```ada
with Interfaces; use Interfaces;

package Init is
  procedure F (B : Boolean; P : out Unsigned_16);
  procedure G;
end Init;
```

Listing 5: init.adb

```ada
package body Init is
  procedure F (B : Boolean; P : out Unsigned_16) is
    begin
      if B then
        P := 3;
      end if;
    end F;

  procedure G is
    U : Unsigned_16;
    begin
      F (False, U);
      if U = 3 then
        null;
      end if;
    end G;
end Init;
```

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
init.ads:4:30: medium: "P" might not be initialized in "F" [reason for check: OUT parameter should be initialized on return] [possible fix: initialize "P" on all paths or make "P" an IN OUT parameter]
gnatprove: unproved check messages considered as errors

We can correct the program by initializing P to value 0 when condition B is not satisfied:

Listing 6: init.ads

```ada
with Interfaces; use Interfaces;

package Init is
  procedure F (B : Boolean; P : out Unsigned_16);
  procedure G;
end Init;
```

Listing 7: init.adb

```ada
package body Init is

  procedure F (B : Boolean; P : out Unsigned_16) is
    begin
      if B then
        P := 3;
      else
        P := 0;
      end if;
    end F;

  procedure G is
    U : Unsigned_16;
    begin
      F (False, U);
      if U = 3 then
        null;
      end if;
    end G;

end Init;
```

81.1. Detecting Reads of Uninitialized Data
GNATprove now does not report any possible reads of uninitialized data. On the contrary, it confirms that all reads are made from initialized data.

In contrast with C, SPARK does not guarantee that global data (called library-level data in SPARK and Ada) is zero-initialized at program startup. Instead, GNATprove checks that all global data is explicitly initialized (at declaration or elsewhere) before it is read. Hence it goes beyond the MISRA C Rule 9.1, which considers global data as always initialized even if the default value of all-zeros might not be valid data for the application. Here’s a variation of the above code where variable U is now global:

```
with Interfaces; use Interfaces;

package Init is
  U : Unsigned_16;
  procedure F (B : Boolean);
  procedure G;
end Init;
```

```
package body Init is
  procedure F (B : Boolean) is
    begin
      if B then
        U := 3;
      end if;
      F;
  end F;
  procedure G is
    begin
      F (False);
      if U = 3 then
        null;
      end if;
      G;
    end G;
end Init;
```

```
with Init;
procedure Call_Init is
begin
  Init.G;
end Call_Init;
```

**Code block metadata**

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Read_Uninitialized_Data_Ada
MD5: a85cde45a658727975367b041a1a5dc3

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
call_init.adb:5:08: medium: "U" might not be initialized after elaboration of main
GNATprove reports here that variable \( U \) might not be initialized at program startup, which is indeed the case here. It reports this issue on the main program \texttt{Call_Init} because its analysis showed that \( F \) needs to take \( U \) as an initialized input (since \( F \) is not initializing \( U \) on all paths, \( U \) keeps its value on the other path, which needs to be an initialized value), which means that \( G \) which calls \( F \) also needs to take \( U \) as an initialized input, which in turn means that \texttt{Call_Init} which calls \( G \) also needs to take \( U \) as an initialized input. At this point, we've reached the main program, so the initialization phase (referred to as \textit{elaboration} in SPARK and Ada) should have taken care of initializing \( U \). This is not the case here, hence the message from GNATprove.

It is possible in SPARK to specify that \( G \) should initialize variable \( U \); this is done with a \textit{data dependency} contract introduced with aspect \texttt{Global} following the declaration of procedure \( G \):

\begin{verbatim}
Listing 11: init.ads

with Interfaces; use Interfaces;
package Init is
  U : Unsigned_16;
  procedure F (B : Boolean);
  procedure G with Global => (Output => U);
end Init;

Listing 12: init.adb

package body Init is
  procedure F (B : Boolean) is
    if B then
      U := 3;
    end if;
  end F;
  procedure G is
    begin
      F (False);
      if U = 3 then
        null;
      end if;
    end G;
end Init;

Listing 13: call_init.adb

with Init;
procedure Call_Init is
  begin
    Init.G;
  end Call_Init;
\end{verbatim}
Learning Ada

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Read_Uninitialized_Data_Ada
MD5: 100122ca3c8c60c134822a85d564a60a

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
init.adb:12:07: high: "U" is not initialized
init.adb:12:07: high: "U" is not an input in the Global contract of subprogram "G"
init.adb:12:07: high: either make "U" an input in the Global contract or initialize it before use
init.adb:14:07: warning: statement has no effect
gnatprove: unproved check messages considered as errors

GNATprove reports the error on the call to F in G, as it knows at this point that F needs U to be initialized but the calling context in G cannot provide that guarantee. If we provide the same data dependency contract for F, then GNATprove reports the error on F itself, similarly to what we saw for an output parameter U.

81.2 Detecting Partial or Redundant Initialization of Arrays and Structures

The other rules in the section on "Initialization" deal with common errors in initializing aggregates and designated initializers in C99 to initialize a structure or array at declaration. These rules attempt to patch holes created by the lax syntax and rules in C standard. For example, here are five valid initializations of an array of 10 elements in C:

Listing 14: main.c

```
int main() {
    int a[10] = {0};
    int b[10] = {0, 0};
    int c[10] = {0, [8] = 0};
    int d[10] = {0, [8] = 0, 0};
    int e[10] = {0, [8] = 0, 0, [8] = 1};
    return 0;
}
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Redundant_Init
MD5: 1212a5565fc3a382e7f967d1cf0b48f9

Only a is fully initialized to all-zeros in the above code snippet. MISRA C Rule 9.3 thus forbids all other declarations by stating that "Arrays shall not be partially initialized". In addition, MISRA C Rule 9.4 forbids the declaration of e by stating that "An element of an object shall not be initialised more than once" (in e's declaration, the element at index 8 is initialized twice).

The same holds for initialization of structures. Here is an equivalent set of declarations with the same potential issues:

Listing 15: main.c

```
int main() {
    typedef struct { int x; int y; int z; } rec;
}'''

(continues on next page)
Learning Ada

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Redundant_Init
MD5: e562ef70e8ca170d2bd09281cf2a075

Here only a, d and e are fully initialized. MISRA C Rule 9.3 thus forbids the declarations of b and c. In addition, MISRA C Rule 9.4 forbids the declaration of e.

In SPARK and Ada, the aggregate used to initialize an array or a record must fully cover the components of the array or record. Violations lead to compilation errors, both for records:

Listing 16: init_record.ads

```ada
package Init_Record is
  type Rec is record
    X, Y, Z : Integer;
  end record;
  R : Rec := (X => 1);  -- ERROR, Y and Z not specified
end Init_Record;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Record_1
MD5: 6b28bffe6270c5ea5055123c5b89c508

Build output

init_record.ads:5:15: error: no value supplied for component "Y"
ininit_record.ads:5:15: error: no value supplied for component "Z"
gprbuild: *** compilation phase failed

and for arrays:

Listing 17: init_array.ads

```ada
package Init_Array is
  type Arr is array (1 .. 10) of Integer;
  A : Arr := (1 => 1);  -- ERROR, elements 2..10 not specified
end Init_Array;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Array_1
MD5: 81aa6363ba770ded10be8d3d8776914

Build output

init_array.ads:3:15: warning: too few elements for type "Arr" defined at line 2
init_array.ads:3:15: warning: expected 10 elements; found 1 element
init_array.ads:3:15: warning: Constraint_Error will be raised at run time

81.2. Detecting Partial or Redundant Initialization of Arrays and Structures
Similarly, redundant initialization leads to compilation errors for records:

**Listing 18: init_record.ads**

```ada
package Init_Record is
  type Rec is record
    X, Y, Z : Integer;
  end record;
  R : Rec := (X => 1, Y => 1, Z => 1, X => 2); -- ERROR, X duplicated
end Init_Record;
```

**Code block metadata**

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Record_2
MD5: 07d3f790009be97cef2daaf08b2f7af6d

**Build output**

init_record.ads:5:40: error: more than one value supplied for "X"
gprbuild: *** compilation phase failed

and for arrays:

**Listing 19: init_array.ads**

```ada
package Init_Array is
  type Arr is array (1 .. 10) of Integer;
  A : Arr := (1 .. 8 => 1, 9 .. 10 => 2, 7 => 3); -- ERROR, A(7) duplicated
end Init_Array;
```

**Code block metadata**

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Array_2
MD5: 12f5fa4615abccde43f63f72340fd4a0

**Build output**

init_array.ads:3:43: error: index value in array aggregate duplicates the one given at line 3
init_array.ads:3:43: error: 7
Build error: *** compilation phase failed

Finally, while it is legal in Ada to leave uninitialized parts in a record or array aggregate by using the box notation (meaning that the default initialization of the type is used, which may be no initialization at all), SPARK analysis rejects such use when it leads to components not being initialized, both for records:

**Listing 20: init_record.ads**

```ada
package Init_Record is
  type Rec is record
    X, Y, Z : Integer;
  end record;
  R : Rec := (X => 1, others => <>); -- ERROR, Y and Z not specified
end Init_Record;
```

**Code block metadata**

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Record_3
MD5: a7736f2b563c39fb4ab10007e927ad97

**Prover output**
Phase 1 of 2: generation of Global contracts ...

Phase 2 of 2: analysis of data and information flow ...

init_record.ads:5:04: error: "R" is not allowed in SPARK (due to box notation, without default initialization)

init_record.ads:5:04: error: violation of pragma SPARK_Mode at /vagrant/frontend/dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Initialization/Init_Record_3/a7736f2b563c39fb4ab10007e927ad97/main_spark.adc:12

init_record.ads:5:15: error: box notation without default initialization is not allowed in SPARK (SPARK RM 4.3(1))

init_record.ads:5:15: error: violation of pragma SPARK_Mode at /vagrant/frontend/dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Initialization/Init_Record_3/a7736f2b563c39fb4ab10007e927ad97/main_spark.adc:12

gnatprove: error during analysis of data and information flow

and for arrays:

Listing 21: init_array.ads

```ada
package Init_Array is
  type Arr is array (1 .. 10) of Integer;
  A : Arr := (1 .. 8 => 1, 9 .. 10 => <>); -- ERROR, A(9..10) not specified
end Init_Array;
```

81.2. Detecting Partial or Redundant Initialization of Arrays and Structures

1657
As with most programming languages, C allows side effects in expressions. This leads to subtle issues about conflicting side effects, when subexpressions of the same expression read/write the same variable.

### 82.1 Preventing Undefined Behavior

Conflicting side effects are a kind of undefined behavior; the C Standard (section 6.5) defines the concept as follows:

"Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored"  

This legalistic wording is somewhat opaque, but the notion of sequence points is summarized in Annex C of the C90 and C99 standards. MISRA C repeats these conditions in the Amplification of Rule 13.2, including the read of a volatile variable as a side effect similar to writing a variable.

This rule is undecidable, so MISRA C completes it with two rules that provide simpler restrictions preventing some side effects in expressions, thus reducing the potential for undefined behavior:

- **Rule 13.3:** "A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator".
- **Rule 13.4:** "The result of an assignment operator should not be used".

In practice, conflicting side effects usually manifest themselves as portability issues, since the result of the evaluation of an expression depends on the order in which a compiler decides to evaluate its subexpressions. So changing the compiler version or the target platform might lead to a different behavior of the application.

To reduce the dependency on evaluation order, MISRA C Rule 13.1 states: "Initializer lists shall not contain persistent side effects". This case is theoretically different from the previously mentioned conflicting side effects, because initializers that comprise an initializer list are separated by sequence points, so there is no risk of undefined behavior if two initializers have conflicting side effects. But given that initializers are executed in an unspecified order, the result of a conflict is potentially as damaging for the application.
82.2 Reducing Programmer Confusion

Even in cases with no undefined or unspecified behavior, expressions with multiple side effects can be confusing to programmers reading or maintaining the code. This problem arises in particular with C’s increment and decrement operators that can be applied prior to or after the expression evaluation, and with the assignment operator = in C since it can easily be mistaken for equality. Thus MISRA C forbids the use of the increment / decrement (Rule 13.3) and assignment (Rule 13.4) operators in expressions that have other potential side effects.

In other cases, the presence of expressions with side effects might be confusing, if the programmer wrongly thinks that the side effects are guaranteed to occur. Consider the function decrease_until_one_is_null below, which decreases both arguments until one is null:

```c
#include <stdio.h>

void decrease_until_one_is_null (int *x, int *y) {
    if (x == 0 || y == 0) {
        return;
    }
    while (!(*x == 0 && *y == 0)) {
        // nothing
    }
}

int main() {
    int x = 42, y = 42;
    decrease_until_one_is_null (&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

The program produces the following output:

```
x = 0, y = 1
```

I.e., starting from the same value 42 for both x and y, only x has reached the value zero after decrease_until_one_is_null returns. The reason is that the side effect on y is performed only conditionally. To avoid such surprises, MISRA C Rule 13.5 states: "The right hand operand of a logical && or || operator shall not contain persistent side effects"; this rule forbids the code above.

MISRA C Rule 13.6 similarly states: "The operand of the sizeof operator shall not contain any expression which has potential side effects". Indeed, the operand of `sizeof` is evaluated only in rare situations, and only according to C99 rules, which makes any side effect in such an operand a likely mistake.
82.3 Side Effects and SPARK

In SPARK, expressions cannot have side effects; only statements can. In particular, there are no increment/decrement operators, and no assignment operator. There is instead an assignment statement, whose syntax using := clearly distinguishes it from equality (using =). And in any event an expression is not allowed as a statement and this a construct such as \( X = Y \); would be illegal. Here is how a variable \( X \) can be assigned, incremented and decremented:

\[
X := 1; \\
X := X + 1; \\
X := X - 1;
\]

There are two possible side effects when evaluating an expression:

- a read of a volatile variable
- a side effect occurring inside a function that the expression calls

Reads of volatile variables in SPARK are restricted to appear immediately at statement level, so the following is not allowed:

```
package Volatile_Read is
  X : Integer with Volatile;
  procedure P (Y : out Integer);
end Volatile_Read;
```

```
package body Volatile_Read is
  procedure P (Y : out Integer) is
  begin
    Y := X - X; -- ERROR
  end P;
end Volatile_Read;
```

Instead, every read of a volatile variable must occur immediately before being assigned to another variable, as follows:

```
package Volatile_Read is
  X : Integer with Volatile;
  procedure P (Y : out Integer);
end Volatile_Read;
```

---

**Project:** Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Volatile_Read_1  
**MD5:** 7ec58b4d1432d03d60b5ea6019cc031e  

**Prover output**

Phase 1 of 2: generation of Global contracts ...

```
volatile_read.adb:4:12: error: volatile object cannot appear in this context
  -(SPARK RM 7.1.3(10))
volatile_read.adb:4:16: error: volatile object cannot appear in this context
  -(SPARK RM 7.1.3(10))
gnatprove: error during generation of Global contracts
```

---
Listing 5: volatile_read.adb

```ada
package body Volatile_Read is
  procedure P (Y : out Integer) is
    X1 : constant Integer := X;
    X2 : constant Integer := X;
  begin
    Y := X1 - X2;
  end P;
end Volatile_Read;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Volatile_Read_2
MD5: 1224af597a12a8ca77b96976c76b422f

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
volatile_read.ads:3:17: info: initialization of "Y" proved

Note here that the order of capture of the volatile value of X might be significant. For example, X might denote a quantity which only increases, like clock time, so that the above expression X1 - X2 would always be negative or zero.

Even more significantly, functions in SPARK cannot have side effects; only procedures can. The only effect of a SPARK function is the computation of a result from its inputs, which may be passed as parameters or as global variables. In particular, SPARK functions cannot have out or in out parameters:

Listing 6: bad_function.ads

```ada
function Bad_Function (X, Y : Integer; Sum, Max : out Integer) return Boolean;
-- ERROR, since "out" parameters are not allowed
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Function_With_Out_Param
MD5: 204dd22d6f1e15208ae34ebc3828974

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
bad_function.ads:1:10: error: function with "out" parameter is not allowed in SPARK
bad_function.ads:1:10: error: violation of pragma SPARK_Mode at /vagrant/frontend/..dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Side_Effect/Function_With_Out_Param/204dd22d6f1e15208ae34ebc3828974/main_spark.adc:12
gnatprove: error during analysis of data and information flow

More generally, SPARK does not allow functions that have a side effect in addition to returning their result, as is typical of many idioms in other languages, for example when setting a new value and returning the previous one:

Listing 7: bad_functions.ads

```ada
package Bad_Functions is
  function Set (V : Integer) return Integer;
  function Get return Integer;
end Bad_Functions;
```

1662 Chapter 82. Controlling Side Effects
**Listing 8: bad_functions.adb**

```ada
package body Bad_Functions is

  Value : Integer := 0;

  function Set (V : Integer) return Integer is
    Previous : constant Integer := Value;
  begin
    Value := V;  -- ERROR
    return Previous;
  end Set;

  function Get return Integer is (Value);

end Bad_Functions;
```

---

**Code block metadata**

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Side_Effect_Ada
MD5: 3337b6025c4996e7fa8c7e27b4df42c1

**Prover output**

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
bad_functions.ads:2:13: error: function with output global "Value" is not allowed in SPARK

GNATprove detects that function Set has a side effect on global variable Value and issues an error. The correct idiom in SPARK for such a case is to use a procedure with an `out` parameter to return the desired result:

**Listing 9: ok_subprograms.ads**

```ada
package Ok_Subprograms is

  procedure Set (V : Integer; Prev : out Integer);

  function Get return Integer;

end Ok_Subprograms;
```

**Listing 10: ok_subprograms.adb**

```ada
package body Ok_Subprograms is

  Value : Integer := 0;

  procedure Set (V : Integer; Prev : out Integer) is
  begin
    Prev := Value;
    Value := V;
  end Set;

  function Get return Integer is (Value);

end Ok_Subprograms;
```

---

**Code block metadata**

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.No_Side_Effect_Ada
MD5: 04e2235b8b6a01706434d35f6636674c
Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
ok_subprograms.ads:2:32: info: initialization of "Prev" proved

With the above restrictions in SPARK, none of the conflicts of side effects that can occur in C can occur in SPARK, and this is guaranteed by flow analysis.
CHAPTER
EIGHTYTHREE

DETECTING UNDEFINED BEHAVIOR

Undefined behavior (and critical unspecified behavior, which we'll treat as undefined behavior) are the plague of C programs. Many rules in MISRA C are designed to avoid undefined behavior, as evidenced by the twenty occurrences of "undefined" in the MISRA C:2012 document.

MISRA C Rule 1.3 is the overarching rule, stating very simply:

"There shall be no occurrence of undefined or critical unspecified behaviour."

The deceptive simplicity of this rule rests on the definition of undefined or critical unspecified behaviour. Appendix H of MISRA:C 2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable.

It is therefore not surprising that a majority of MISRA C checkers do not make a serious attempt to verify compliance with MISRA C Rule 1.3.

83.1 Preventing Undefined Behavior in SPARK

Since SPARK is a subset of the Ada programming language, SPARK programs may exhibit two types of undefined behaviors that can occur in Ada:

- **bounded error**: when the program enters a state not defined by the language semantics, but the consequences are bounded in various ways. For example, reading uninitialized data can lead to a bounded error, when the value read does not correspond to a valid value for the type of the object. In this specific case, the Ada Reference Manual states that either a predefined exception is raised or execution continues using the invalid representation.

- **erroneous execution**: when the program enters a state not defined by the language semantics, but the consequences are not bounded by the Ada Reference Manual. This is the closest to an undefined behavior in C. For example, concurrently writing through different tasks to the same unprotected variable is a case of erroneous execution.

Many cases of undefined behavior in C would in fact raise exceptions in SPARK. For example, accessing an array beyond its bounds raises the exception Constraint_Error while reaching the end of a function without returning a value raises the exception Program_Error.

The SPARK Reference Manual defines the SPARK subset through a combination of legality rules (checked by the compiler, or the compiler-like phase preceding analysis) and verification rules (checked by the formal analysis tool GNATprove). Bounded errors and erroneous execution are prevented by a combination of legality rules and the flow analysis part of GNATprove, which in particular detects potential reads of uninitialized data, as described in Detecting Reads of Uninitialized Data (page 1649). The following discussion focuses on how SPARK can verify that no exceptions can be raised.
83.2 Proof of Absence of Run-Time Errors in SPARK

The most common run-time errors are related to misuse of arithmetic (division by zero, overflows, exceeding the range of allowed values), arrays (accessing beyond an array bounds, assigning between arrays of different lengths), and structures (accessing components that are not defined for a given variant).

Arithmetic run-time errors can occur with signed integers, unsigned integers, fixed-point and floating-point (although with IEEE 754 floating-point arithmetic, errors are manifest as special run-time values such as NaN and infinities rather than as exceptions that are raised). These errors can occur when applying arithmetic operations or when converting between numeric types (if the value of the expression being converted is outside the range of the type to which it is being converted).

Operations on enumeration values can also lead to run-time errors; e.g., T'Pred(T'First) or T'Succ(T'Last) for an enumeration type T, or T'Val(N) where N is an integer value that is outside the range 0 .. T'Pos(T'Last).

The Update procedure below contains what appears to be a simple assignment statement, which sets the value of array element A(I+J) to P/Q.

Listing 1: show_runtime_errors.ads

```ada
package Show_Runtime_Errors is

   type Nat_Array is array (Integer range <>) of Natural;
   -- The values in subtype Natural are 0, 1, ... Integer'Last

   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);

end Show_Runtime_Errors;
```

Listing 2: show_runtime_errors.adb

```ada
package body Show_Runtime_Errors is

   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
      begin
         A (I + J) := P / Q;
      end Update;

end Show_Runtime_Errors;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Undefinend_Behavior.Runtime_Errors
MD5: 8ad4488974ab9e49ac17bf094ae33eac

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_runtime_errors.adb:5:12: medium: overflow check might fail, cannot prove lower bound for I + J [reason for check: result of addition must fit in a 32-bits machine integer] [possible fix: add precondition (if J >= 0 then I <= Integer'Last - J else I >= Integer'First - J) to subprogram at show_runtime_errors.ads:6]
show_runtime_errors.adb:5:12: medium: array index check might fail [reason for check: result of addition must be a valid index into the array] [possible fix: add precondition (if J >= 0 then I <= A'Last - J else I >= A'First - J) to subprogram at show_runtime_errors.ads:6]
(continues on next page)
However, for an arbitrary invocation of this procedure, say Update(A, I, J, P, Q), an exception can be raised in a variety of circumstances:

- The computation \(I + J\) may overflow, for example if \(I\) is `Integer' Last` and \(J\) is positive.

  \[
  A \ (\text{Integer'} \text{Last} + 1) := P / Q;
  \]

- The value of \(I + J\) may be outside the range of the array \(A\).

  \[
  A \ (A' \text{Last} + 1) := P / Q;
  \]

- The division \(P / Q\) may overflow in the special case where \(P\) is `Integer' First` and \(Q\) is `-1`, because of the asymmetric range of signed integer types.

  \[
  A \ (I + J) := \text{Integer'} \text{First} / (-1);
  \]

- Since the array can only contain non-negative numbers (the element subtype is `Natural`), it is also an error to store a negative value in it.

  \[
  A \ (I + J) := 1 / (-1);
  \]

- Finally, if \(Q\) is 0 then a divide by zero error will occur.

  \[
  A \ (I + J) := P / 0;
  \]

For each of these potential run-time errors, the compiler will generate checks in the executable code, raising an exception if any of the checks fail:

\[
\begin{align*}
A \ (\text{Integer'} \text{Last} + 1) := P / Q; & \quad -- \text{raised CONSTRAINT\_ERROR : overflow check failed} \\
A \ (A' \text{Last} + 1) := P / Q; & \quad -- \text{raised CONSTRAINT\_ERROR : index check failed} \\
A \ (I + J) := \text{Integer'} \text{First} / (-1); & \quad -- \text{raised CONSTRAINT\_ERROR : overflow check failed} \\
A \ (I + J) := 1 / (-1); & \quad -- \text{raised CONSTRAINT\_ERROR : range check failed} \\
A \ (I + J) := P / 0; & \quad -- \text{raised CONSTRAINT\_ERROR : divide by zero}
\end{align*}
\]

These run-time checks incur an overhead in program size and execution time. Therefore it may be appropriate to remove them if we are confident that they are not needed.

The traditional way to obtain the needed confidence is through testing, but it is well known that this can never be complete, at least for non-trivial programs. Much better is to guarantee the absence of run-time errors through sound static analysis, and that’s where SPARK and GNATprove can help.

83.2. Proof of Absence of Run-Time Errors in SPARK
More precisely, GNATprove logically interprets the meaning of every instruction in the program, taking into account both control flow and data/information dependencies. It uses this analysis to generate a logical formula called a verification condition for each possible check.

A (\texttt{Integer}'Last + 1) := P / Q;
\textit{-- medium: overflow check might fail}

A (A'Last + 1) := P / Q;
\textit{-- medium: array index check might fail}

A (I + J) := \texttt{Integer}'First / (-1);
\textit{-- medium: overflow check might fail}

A (I + J) := 1 / (-1);
\textit{-- medium: range check might fail}

A (I + J) := P / 0;
\textit{-- medium: divide by zero might fail}

The verification conditions are then given to an automatic prover. If every verification condition can be proved, then no run-time errors will occur.

GNATprove's analysis is sound — it will detect all possible instances of run-time exceptions being raised — while also having high precision (i.e., not producing a cascade of "false alarms").

The way to program in SPARK so that GNATprove can guarantee the absence of run-time errors entails:

- declaring variables with precise constraints, and in particular to specify precise ranges for scalars; and
- defining preconditions and postconditions on subprograms, to specify respectively the constraints that callers should respect and the guarantees that the subprogram should provide on exit.

For example, here is a revised version of the previous example, which can guarantee through proof that no possible run-time error can be raised:

```
package No_Runtime_Errors is

  subtype Index_Range is Integer range 0 .. 100;

  type Nat_Array is array (Index_Range range <>) of Natural;

  procedure Update (A : in out Nat_Array;
                    I, J : Index_Range;
                    P, Q : Positive)
  with Pre => I + J in A'Range;

end No_Runtime_Errors;
```

Listing 4: no_runtime_errors.adb
begin
  A (I + J) := P / Q;
end Update;
end No_Runtime_Errors;

83.2. Proof of Absence of Run-Time Errors in SPARK
Chapter 83. Detecting Undefined Behavior
MISRA C defines *unreachable code* as code that cannot be executed, and it defines *dead code* as code that can be executed but has no effect on the functional behavior of the program. (These definitions differ from traditional terminology, which refers to the first category as "dead code" and the second category as "useless code".) Regardless of the terminology, however, both types are actively harmful, as they might confuse programmers and lead to errors during maintenance.

The "Unused code" section of MISRA C contains seven rules that deal with detecting both unreachable code and dead code. The two most important rules are:

- Rule 2.1: "A project shall not contain unreachable code", and
- Rule 2.2: "There shall not be dead code".

Other rules in the same section prohibit unused entities of various kinds (type declarations, tag declarations, macro declarations, label declarations, function parameters).

While some simple cases of unreachable code can be detected by static analysis (typically if a condition in an *if* statement can be determined to be always true or false), most cases of unreachable code can only be detected by performing coverage analysis in testing, with the caveat that code reported as not being executed is not necessarily unreachable (it could simply reflect gaps in the test suite). Note that statement coverage, rather than the more comprehensive decision coverage or modified condition / decision coverage (MC/DC) as defined in the DO-178C standard for airborne software, is sufficient to detect potential unreachable statements, corresponding to code that is not covered during the testing campaign.

The presence of dead code is much harder to detect, both statically and dynamically, as it requires creating a complete dependency graph linking statements in the code and their effect on visible behavior of the program.

SPARK can detect some cases of both unreachable and dead code through its precise construction of a dependency graph linking a subprogram's statements to all its inputs and outputs. This analysis might not be able to detect complex cases, but it goes well beyond what other analyses do in general.

Listing 1: much_ado_about_little.ads

```adaptors
procedure Much_Ado_About_Little (X, Y, Z : Integer; Success : out Boolean);
```

Listing 2: much_ado_about_little.adb

```adaptors
procedure Much_Ado_About_Little (X, Y, Z : Integer; Success : out Boolean) is

  procedure Ok is
  begin
    Success := True;
  end Ok;
```

(continues on next page)
procedure NOk is
begin
  Success := False;
end NOk;

begin
  Success := False;
  for K in Y .. Z loop
    if K < X and not Success then
      Ok;
    end if;
  end loop;
  if X > Y then
    Ok;
  else
    NOk;
  end if;
  if Z > Y then
    NOk;
  else
    Ok;
  end if;
  if Success then
    Success := not Success;
  end if;
end Much_Ado_About_Little;

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Unreachable_And_Dead_Code.Much_Ado_About_Little
MD5: ccccb112fbab169ba964b3f8ef36ec2d

Build output

much_ado_about_little.adb:36:04: warning: unreachable code [enabled by default]

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
  much_ado_about_little.adb:5:15: warning: unused assignment, in call inlined at ...
  much_ado_about_little.adb:5:15: warning: unused assignment, in call inlined at ...
  much_ado_about_little.adb:10:15: warning: unused assignment, in call inlined at ...
  much_ado_about_little.adb:10:15: warning: unused assignment, in call inlined at ...
  much_ado_about_little.adb:14:12: warning: unused assignment
  much_ado_about_little.adb:16:20: warning: statement has no effect
  much_ado_about_little.adb:17:07: warning: statement has no effect
  much_ado_about_little.adb:22:04: warning: statement has no effect
  much_ado_about_little.adb:36:04: warning: unreachable code [enabled by default]
  much_ado_about_little.adb:36:04: warning: this statement is never reached
  much_ado_about_little.adb:37:15: warning: this statement is never reached
  much_ado_about_little.ads:1:34: warning: unused initial value of "X"
The only code in the body of Much_Ado_About_Little that affects the result of the procedure's execution is the if \( Z > Y \)\ldots\ statement, since this statement sets Success to either True or False regardless of what the previous statements did. I.e., the statements preceding this if are dead code in the MISRA C sense. Since both branches of the if \( Z > Y \)\ldots\ statement return from the procedure, the subsequent if Success\ldots\ statement is unreachable. GNATprove detects and issues warnings about both the dead code and the unreachable code.
The C programming language is "close to the metal" and has emerged as a *lingua franca* for the majority of embedded platforms of all sizes. However, its software engineering deficiencies (such as the absence of data encapsulation) and its many traps and pitfalls present major obstacles to those developing critical applications. To some extent, it is possible to put the blame for programming errors on programmers themselves, as Linus Torvalds admonished:

"Learn C, instead of just stringing random characters together until it compiles (with warnings)."

But programmers are human, and even the best would be hard pressed to be 100% correct about the myriad of semantic details such as those discussed in this document. Programming language abstractions have been invented precisely to help developers focus on the "big picture" (thinking in terms of problem-oriented concepts) rather than low-level machine-oriented details, but C lacks these abstractions. As Kees Cook from the Kernel Self Protection Project puts it (during the Linux Security Summit North America 2018):

"Talking about C as a language, and how it's really just a fancy assembler"

Even experts sometimes have problems with the C programming language rules, as illustrated by Microsoft expert David LeBlanc (see *Enforcing Strong Typing for Scalars* (page 1639)) or the MISRA C Committee itself (see the Preface (page 1609)).

The rules in MISRA C represent an impressive collective effort to improve the reliability of C code in critical applications, with a focus on avoiding error-prone features rather than enforcing a particular programming style. The Rationale provided with each rule is a clear and unobjectionable justification of the rule's benefit.

At a fundamental level, however, MISRA C is still built on a base language that was not really designed with the goal of supporting large high-assurance applications. As shown in this document, there are limits to what static analysis can enforce with respect to the MISRA C rules. It's hard to retrofit reliability, safety and security into a language that did not have these as goals from the start.

The SPARK language took a different approach, starting from a base language (Ada) that was designed from the outset to support solid software engineering, and eliminating features that were implementation dependent or otherwise hard to formally analyze. In this document we have shown how the SPARK programming language and its associated formal verification tools can contribute usefully to the goal of producing error-free software, going beyond the guarantees that can be achieved in MISRA C.
86.1 About MISRA C

The official website of the MISRA association https://www.misra.org.uk/ has many freely available resources about MISRA C, some of which can be downloaded after registering on the MISRA Bulletin Board at https://www.misra.org.uk/forum/ (such as the examples from the MISRA C:2012 standard, which includes a one-line description of each guideline).

The following documents are freely available:

- *MISRA Compliance 2016: Achieving compliance with MISRA coding guidelines*, 2016, which explains the rationale and process for compliance, including a thorough discussions of acceptable deviations

- *MISRA C:2012 - Amendment 1: Additional security guidelines for MISRA C:2012*, 2016, which contains 14 additional guidelines focusing on security. This is a minor addition to MISRA C.

The main MISRA C:2012 document can be purchased from the MISRA webstore.

PRQA is the company that first developed MISRA C, and they have been heavily involved in every version since then. Their webpage http://www.prqa.com/coding-standards/misra/ contains many resources about MISRA C: product datasheets, white papers, webinars, professional courses.

The PRQA Resources Library at http://info.prqa.com/resources-library?filter=white_paper has some freely available white papers on MISRA C and the use of static analyzers:


In 2013 ISO standardized a set of 45 rules focused on security, available in the C Secure Coding Rules. A draft is freely available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1624.pdf

86.2 About SPARK

The e-learning website https://learn.adacore.com/ contains a freely available interactive course on SPARK.


A student-oriented textbook on SPARK is Building High Integrity Applications with SPARK by John McCormick and Peter Chapin, published by Cambridge University Press. It covers the latest version of the language, SPARK 2014.

A historical account of the evolution of SPARK technology and its use in industry is covered in the article Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK by Roderick Chapman and Florian Schanda, at http://proteancode.com/keynote.pdf

The website https://www.adacore.com/sparkpro is a portal for up-to-date information and resources on SPARK. AdaCore blog's site https://blog.adacore.com/ contains a number of SPARK-related posts.

The booklet AdaCore Technologies for Cyber Security shows how AdaCore's technology can be used to prevent or mitigate the most common security vulnerabilities in software. See https://www.adacore.com/books/ada-core-tech-for-cyber-security/.

The booklet AdaCore Technologies for CENELEC EN 50128:2011 shows how AdaCore's technology can be used in conjunction with the CENELEC EN 50128:2011 software standard for railway control and protection systems. It describes in particular where the SPARK technology fits best and how it can be used to meet various requirements of the standard. See: https://www.adacore.com/books/cenelec-en-50128-2011/.

The booklet AdaCore Technologies for DO-178C/ED-12C similarly shows how AdaCore's technology can be used in conjunction with the DO-178C/ED-12C standard for airborne software, and describes in particular how SPARK can be used in conjunction with the Formal Methods supplement DO-333/ED-216. See https://www.adacore.com/books/do-178c-tech/.

86.3 About MISRA C and SPARK


The white paper A Comparison of SPARK with MISRA C and Frama-C at https://www.adacore.com/papers/compare-spark-MISRA-C-frama-c compares SPARK to MISRA C and to the formal verification tool Frama-C for C programs.
Part IX

Introduction to the GNAT Toolchain
This course presents an introduction to the GNAT toolchain. The course includes first steps to get started with the toolchain and some details on the project manager (GPRbuild) and the integrated development environment (GNAT Studio).

This document was written by Gustavo A. Hoffmann, with contributions and review from Richard Kenner and Robert Duff.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

345 http://creativecommons.org/licenses/by-sa/4.0
This chapter presents a couple of basic commands from the GNAT toolchain.

### 87.1 Basic commands

Now that the toolchain is installed, you can start using it. From the command line, you can compile a project using `gprbuild`. For example:

```
gprbuild -P project.gpr
```

You can find the binary built with the command above in the `obj` directory. You can run it in the same way as you would do with any other executable on your platform. For example:

```
obj/main
```

A handy command-line option for `gprbuild` you might want to use is `-p`, which automatically creates directories such as `obj` if they aren't in the directory tree:

```
gprbuild -p -P project.gpr
```

Ada source-code are stored in `.ads` and `.adb` files. To view the content of these files, you can use GNAT Studio. To open GNAT Studio, double-click on the `.gpr` project file or invoke GNAT Studio on the command line:

```
gps -P project.gpr
```

To compile your project using GNAT Studio, use the top-level menu to invoke Build → Project → main.adb (or press the keyboard shortcut F4). To run the main program, click on Build → Run → main (or press the keyboard shortcut Shift + F2).

### 87.2 Compiler warnings

One of the strengths of the GNAT compiler is its ability to generate many useful warnings. Some are displayed by default but others need to be explicitly enabled. In this section, we discuss some of these warnings, their purpose, and how you activate them.
87.2.1 -gnatwa switch and warning suppression

Section author: Robert Duff

We first need to understand the difference between a warning and an error. Errors are violations of the Ada language rules as specified in the Ada Reference Manual; warnings don't indicate violations of those rules, but instead flag constructs in a program that seem suspicious to the compiler. Warnings are GNAT-specific, so other Ada compilers might not warn about the same things GNAT does or might warn about them in a different way. Warnings are typically conservative; meaning that some warnings are false alarms. The programmer needs to study the code to determine if each warning is describing a real problem.

Some warnings are produced by default while others are produced only if a switch enables them. Use the -gnatwa switch to turn on (almost) all warnings.

Warnings are useless if you don't do anything about them. If you give your team member some code that causes warnings, how are they supposed to know whether they represent real problems? If you don't address each warning, people will soon start ignoring warnings and there'll be lots of things that generates warnings scattered all over your code. To avoid this, you may want to use the -gnatwae switch to both turn on (almost) all warnings and to treat warnings as errors. This forces you to get a clean (no warnings or errors) compilation.

However, as we said, some warnings are false alarms. Use pragma Warnings (Off) to suppress those warnings. It's best to be as specific as possible and narrow down to a single line of code and a single warning. Then use a comment to explain why the warning is a false alarm if it's not obvious.

Let's look at the following example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Warnings_Example is
   procedure Mumble (X : Integer) is
      begin
         Put_Line ("Mumble processing...");
      end Mumble;

end Warnings_Example;

We compile the above code with -gnatwae:

```

```ada`
```

```
gnat compile -gnatwae ./src/warnings_example.adb
```

This causes GNAT to complain:

```
warnings_example.adb:5:22: warning: formal parameter "X" is not referenced
```

But the following compiles cleanly:

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Warnings_Example is
   pragma Warnings (Off, "formal parameter "X" is not referenced");
   procedure Mumble (X : Integer) is
      pragma Warnings (On, "formal parameter "X" is not referenced");
      -- X is ignored here, because blah blah blah...
      begin
         Put_Line ("Mumble processing...");
```

(continues on next page)```
end Mumble;
end Warnings_Example;

Here we've suppressed a specific warning message on a specific line.

If you get many warnings of a specific type and it's not feasible to fix all of them, you can suppress that type of message so the good warnings won't get buried beneath a pile of bogus ones. For example, you can use the -gnatwaeF switch to silence the warning on the first version of Mumble above: the F suppresses warnings on unreferenced formal parameters. It would be a good idea to use it if you have many of those.

As discussed above, -gnatwa activates almost all warnings, but not all. Refer to the section on warnings of the GNAT User's Guide to get a list of the remaining warnings you could enable in your project. One is -gnatw.o, which displays warnings when the compiler detects modified but unreferenced out parameters. Consider the following example:

```ada
package Warnings_Example is

   procedure Process (X : in out Integer;
                      B : out Boolean);

end Warnings_Example;

package body Warnings_Example is

   procedure Process (X : in out Integer;
                      B : out Boolean) is

      begin
         if X = Integer'First or else X = Integer'Last then
            B := False;
         else
            X := X + 1;
            B := True;
         end if;
      end Process;

end Warnings_Example;

with Ada.Text_IO; use Ada.Text_IO;

with Warnings_Example; use Warnings_Example;

procedure Main  is
   X : Integer := 0;
   Success : Boolean;
begin
   Process (X, Success);
   Put_Line (Integer'image (X));
end Main;
```

If we build the main application using the -gnatw.o switch, the compiler warns us that we didn't reference the Success variable, which was modified in the call to Process:

`main.adb:8:16: warning: "Success" modified by call, but value might not be referenced`

In this case, this actually points us to a bug in our program, since X only contains a valid value if Success is True. The corrected code for Main is:

```ada
package Warnings_Example is

   procedure Process (X : in out Integer;
                      B : out Boolean);

end Warnings_Example;

package body Warnings_Example is

   procedure Process (X : in out Integer;
                      B : out Boolean) is

      begin
         if X = Integer'First or else X = Integer'Last then
            B := False;
         else
            X := X + 1;
            B := True;
         end if;
      end Process;

end Warnings_Example;

with Ada.Text_IO; use Ada.Text_IO;

with Warnings_Example; use Warnings_Example;

procedure Main  is
   X : Integer := 0;
   Success : Boolean;
begin
   Process (X, Success);
   put_line (integer'image (X));
end Main;
```

https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#warning-message-control
begin
  Process (X, Success);

  if Success then
    Put_Line (Integer'Image (X));
  else
    Put_Line ("Couldn't process variable X.");
  end if;
end Main;

We suggest turning on as many warnings as makes sense for your project. Then, when you see a warning message, look at the code and decide if it's real. If it is, fix the code. If it's a false alarm, suppress the warning. In either case, we strongly recommend you make the warning disappear before you check your code into your configuration management system.

87.2.2 Style checking

GNAT provides many options to configure style checking of your code. The main compiler switch for this is -gnaty, which sets almost all standard style check options. As indicated by the section on style checking\(^\text{347}\) of the GNAT User's Guide, using this switch "is equivalent to -gnaty3aAbcefhlmnpqrst, that is all checking options enabled with the exception of -gnatyB, -gnatyE, -gnatyI, -gnatyLnnn, -gnatyO, -gnatyS, -gnatyy, and -gnatyx."

You may find that selecting the appropriate coding style is useful to detect issues at early stages. For example, the -gnaty0 switch checks that overriding subprograms are explicitly marked as such. Using this switch can avoid surprises when you didn't intentionally want to override an operation for some data type. We recommend studying the list of coding style switches and selecting the ones that seem relevant for your project. When in doubt, you can start by using all of them — using -gnatyy and -gnatyBdIL4o0Sux, for example — and deactivating the ones that cause too much noise during compilation.

\(^{347}\) https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#style-checking
This chapter presents a brief overview of **GPRbuild**, the project manager of the GNAT toolchain. It can be used to manage complex builds. In terms of functionality, it's similar to **make** and **cmake**, just to name two examples.


### 88.1 Basic commands

As mentioned in the previous chapter, you can build a project using **gprbuild** from the command line:

```
gprbuild -P project.gpr
```

In order to clean the project, you can use **gprclean**:

```
gprclean -P project.gpr
```

### 88.2 Project files

You can create project files using **GNAT Studio**, which presents many options on its graphical interface. However, you can also edit project files manually as a normal text file in an editor, since its syntax is human readable. In fact, project files use a syntax similar to the one from the Ada language. Let's look at the basic structure of project files and how to customize them.

#### 88.2.1 Basic structure

The main element of a project file is a project declaration, which contains definitions for the current project. A project file may also include other project files in order to compose a complex build. One of the simplest form of a project file is the following:

```plaintext
project Default is
  for Main use ("main");
  for Source_Dirs use ("src");
end Default;
```

In this example, we declare a project named `Default`. The `for Main use` expression indicates that the `main.adb` file is used as the entry point (main source-code file) of the project. The main file doesn't necessary need to be called `main.adb`; we could use any source-code implementing a main application, or even have a list of multiple main files. The `for Source_Dirs use` expression indicates that the `src` directory contains the source-file for the application (including the main file).

### 88.2.2 Customization

GPRbuild support scenario variables, which allow you to control the way binaries are built. For example, you may want to distinguish between debug and optimized versions of your binary. In principle, you could pass command-line options to `gprbuild` that turn debugging on and off, for example. However, defining this information in the project file is usually easier to handle and to maintain. Let's define a scenario variable called `ver` in our project:

```ada
project Default is
  Ver := external ("ver", "debug");
  for Main use ("main");
  for Source_Dirs use ("src");
end Default;
```

In this example, we're specifying that the scenario variable `Ver` is initialized with the external variable `ver`. Its default value is set to `debug`.

We can now set this variable in the call to `gprbuild`:

```bash
gprbuild -P project.gpr -Xver=debug
```

Alternatively, we can simply specify an environment variable. For example, on Unix systems, we can say:

```bash
export ver=debug

# Value from environment variable "ver" used in the following call:

gprbuild -P project.gpr
```

In the project file, we can use the scenario variable to customize the build:

```ada
project Default is
  Ver := external ("ver", "debug");
  for Main use ("main.adb");
  for Source_Dirs use ("src");

-- Using "ver" variable for obj directory
  for Object_Dir use "obj/" & Ver;

package Compiler is
  case Ver is
    when "debug" =>
      for Switches (Ada) use ("-g");
    when "opt" =>
      for Switches (Ada) use ("-O2");
    when others =>
      null;
  end case;
end package;
```

(continues on next page)
end Compiler;
end Default;

We're now using Ver in the for Object_Dir clause to specify a subdirectory of the obj directory that contains the object files. Also, we're using Ver to select compiler options in the Compiler package declaration.

We could also specify all available options in the project file by creating a typed variable. For example:

project Default is

    type Ver_Option is ("debug", "opt");
    Ver : Ver_Option := external ("ver", "debug");

    for Source_Dirs use ("src");
    for Main use ("main.adb");

    -- Using "ver" variable for obj directory
    for Object_Dir use "obj/" & Ver;

    package Compiler is
        case Ver is
            when "debug" =>
                for Switches ("Ada") use ("-g");
            when "opt" =>
                for Switches ("Ada") use ("-O2");
            when others =>
                null;
        end case;
    end Compiler;
end Default;

The advantage of this approach is that gprbuild can now check whether the value that you provide for the ver variable is available on the list of possible values and give you an error if you're entering a wrong value.

### 88.3 Project dependencies

GPRbuild supports project dependencies. This allows you to reuse information from existing projects. Specifically, the keyword with allows you to include another project within the current project.

#### 88.3.1 Simple dependency

Let's look at a very simple example. We have a package called Test_Pkg associated with the project file test_pkg.gpr, which contains:

```ada
project Test_Pkg is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Test_Pkg;
```

This is the code for the Test_Pkg package:
package Test_Pkg is

  type T is record
    X : Integer;
    Y : Integer;
  end record;

  function Init return T;

end Test_Pkg;

package body Test_Pkg is

  function Init return T is
    begin
      return V : T do
        V.X := 0;
        V.Y := 0;
      end return;
    end Init;
  end Test_Pkg;

For this example, we use a directory test_pkg containing the project file and a subdirectory test_pkg/src containing the source files. The directory structure looks like this:

```
|-- test_pkg
  |-- test_pkg.gpr
  |-- src
    |-- test_pkg.adb
    |-- test_pkg.ads
```

Suppose we want to use the Test_Pkg package in a new application. Instead of directly including the source files of Test_Pkg in the project file of our application (either directly or indirectly), we can instead reference the existing project file for the package by using with "test_pkg.gpr". This is the resulting project file:

```
with "../test_pkg/test_pkg.gpr";

project Default is
  for Source_DIRS use ("src");
  for OBJECT_DIR use "obj";
  for MAIN use ("main.adb");
end Default;
```

And this is the code for the main application:

```
with Test_Pkg; use Test_Pkg;

procedure Main is
  A : T;
begin
  A := Init;
end Main;
```

When we build the main project file (default.gpr), we're automatically building all dependent projects. More specifically, the project file for the main application automatically includes the information from the dependent projects such as test_pkg.gpr. Using a with in the main project file is all we have to do for that to happen.
88.3.2 Dependencies to dynamic libraries

We can structure project files to make use of dynamic (shared) libraries using a very similar approach. It's straightforward to convert the project above so that Test_Pkg is now compiled into a dynamic library and linked to our main application. All we need to do is to make a few additions to the project file for the Test_Pkg package:

```ada
library project Test_Pkg is
    for Source_Dirs use "src";
    for Object_Dir use "obj";
    for Library_Name use "test_pkg";
    for Library_Dir use "lib";
    for Library.Kind use "Dynamic";
end Test_Pkg;
```

This is what we had to do:

- We changed the project to library project.
- We added the specification for Library_Name, Library_Dir and Library_Kind.

We don't need to change the project file for the main application because **GPRbuild** automatically detects the dependency information (e.g., the path to the dynamic library) from the project file for the Test_Pkg package. With these small changes, we're able to compile the Test_Pkg package to a dynamic library and link it with our main application.

88.4 Configuration pragma files

Configuration pragma files contain a set of pragmas that modify the compilation of source files according to external requirements. For example, you may use pragmas to either relax or strengthen requirements depending on your environment.

In **GPRbuild**, we can use Local_Configuration_Pragmas (in the Compiler package) to indicate the configuration pragmas file we want **GPRbuild** to use with the source files in our project.

The file gnat.adc shown here is an example of a configuration pragma file:

```ada
pragma Suppress (Overflow_Check);
```

We can use this in our project by declaring a Compiler package. Here's the complete project file:

```ada
project Default is
    for Source_Dirs use "src";
    for Object_Dir use "obj";
    for Main use "main.adb";

    package Compiler is
        for Local_Configuration_Pragmas use "gnat.adc";
    end Compiler;
end Default;
```

Each pragma contained in gnat.adc is used in the compilation of each file, as if that pragma was placed at the beginning of each file.
**88.5 Configuration packages**

You can control the compilation of your source code by creating variants for various cases and selecting the appropriate variant in the compilation package in the project file. One example where this is useful is conditional compilation using Boolean constants, shown in the code below:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Config;

procedure Main is
begin
  if Config.Debug then
    Put_Line ("Debug version");
  else
    Put_Line ("Release version");
  end if;
end Main;
```

In this example, we declared the Boolean constant in the `Config` package. By having multiple versions of that package, we can create different behavior for each usage. For this simple example, there are only two possible cases: either Debug is `True` or `False`. However, we can apply this strategy to create more complex cases.

In our next example, we store the packages in the subdirectories `debug` and `release` of the source code directory. Here's the content of the `src/debug/config.ads` file:

```ada
package Config is
  Debug : constant Boolean := True;
end Config;
```

Here's the `src/release/config.ads` file:

```ada
package Config is
  Debug : constant Boolean := False;
end Config;
```

In this case, **GPRbuild** selects the appropriate directory to look for the `config.ads` file according to information we provide for the compilation process. We do this by using a scenario type called `Mode_Type` in our project file:

```bash
gprbuild -P default.gpr -Xmode=release
```

```ada
project Default is
  type Mode_Type is ("debug", "release");
  Mode : Mode_Type := external ("mode", "debug");
  for Source_Dirs use ("src", "src/" & Mode);
  for Object_Dir use "obj";
  for Main use ("main.adb");
end Default;
```
We declare the scenario variable Mode and use it in the Source_Dirs declaration to add the desired path to the subdirectory containing the config.ads file. The expression "src/" & Mode concatenates the user-specified mode to select the appropriate subdirectory. For more complex cases, we could use either a tree of subdirectories or multiple scenario variables for each aspect that we need to configure.
This chapter presents an introduction to the GNAT Studio, which provides an IDE to develop applications in Ada. For a detailed overview, please refer to the GNAT Studio tutorial\textsuperscript{349}. Also, you can refer to the GNAT Studio product page\textsuperscript{350} for some introductory videos.

In this chapter, all indications using "→" refer to options from the GNAT Studio menu that you can click in order to execute commands.

### 89.1 Start-up

The first step is to start-up the GNAT Studio. The actual step depends on your platform.

#### 89.1.1 Windows

- You may find an icon (shortcut to GNAT Studio) on your desktop.
- Otherwise, start GNAT Studio by typing gnatstudio on the command prompt.

#### 89.1.2 Linux

- Start GNAT Studio by typing gnatstudio on a shell.

### 89.2 Creating projects

After starting-up GNAT Studio, you can create a project. These are the steps:

- Click on Create new project in the welcome window
  - Alternatively, if the wizard (which let's you customize new projects) isn't already opened, click on File → New Project... to open it.
  - After clicking on Create new project, you should see a window with this title: Create Project from Template.
- Select one of the options from the list and click on Next.
  - The simplest one is Basic > Simple Ada Project, which creates a project containing a main application.
- Select the project location and basic settings, and click on Apply.

\textsuperscript{349} https://docs.adacore.com/live/wave/gps/html/gps_tutorial/index.html
\textsuperscript{350} https://www.adacore.com/gnatpro/toolsuite/gps
- If you selected "Simple Ada Project" in the previous step, you may now select the name of the project and of the main file.
- Note that you can select any name for the main file.

You should now have a working project file.

### 89.3 Building

As soon as you've created a project file, you can use it to build an application. These are the required steps:

- Click on Build → Project → Build All
  - You can also click on this icon:

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
<th>Navigate</th>
<th>Find</th>
<th>Code</th>
<th>VCS</th>
<th>Build</th>
<th>SPARK</th>
<th>Analyze</th>
<th>Debug</th>
<th>View</th>
<th>Window</th>
<th>Help</th>
</tr>
</thead>
</table>

- Alternatively, you can click on Build → Project → Build & Run → <name of your main application>
  - You can also click on this icon:

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
<th>Navigate</th>
<th>Find</th>
<th>Code</th>
<th>VCS</th>
<th>Build</th>
<th>SPARK</th>
<th>Analyze</th>
<th>Debug</th>
<th>View</th>
<th>Window</th>
<th>Help</th>
</tr>
</thead>
</table>

- You can also use the keyboard for building and running the main application:
  - Press F4 to open a window that allows you to build the main application and click on Execute.
  - Then, press Shift + F2 to open a window that allows you to run the application, and click on Execute.

### 89.4 Debugging

#### 89.4.1 Debug information

Before you can debug a project, you need to make sure that debugging symbols have been included in the binary build. You can do this by manually adding a debug version into your project, as described in the previous chapter (see *GPRbuild* (page 1687)).

Alternatively, you can change the project properties directly in **GNAT Studio**. In order to do that, click on **Edit → Project Properties...**, which opens the following window:
Click on Build → Switches → Ada on this window, and make sure that the Debug Information option is selected.

### 89.4.2 Improving main application

If you selected "Simple Ada Project" while creating your project in the beginning, you probably still have a very simple main application that doesn't do anything useful. Therefore, in order to make the debugging activity more interesting, please enter some statements to your application. For example:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is begin
  Put_Line ("Hello World!");
  Put_Line ("Hello again!");
end Main;
```
### 89.4.3 Debugging the application

You can now build and debug the application by clicking on Build → Project → Build & Debug → *<name of your main application>*.

You can then click on Debug → Run... to open a window that allows you to start the application. Alternatively, you can press Shift + F9. As soon as the application has started, you can press F5 to step through the application or press F6 to execute until the next line. Both commands are available in the menu by clicking on Debug → Step or Debug → Next.

When you've finished debugging your application, you need to terminate the debugger. To do this, you can click on Debug → Terminate.

### 89.5 Formal verification

In order to see how SPARK can detect issues, let's creating a simple application that accumulates values in a variable A:

```ada
procedure Main
  with SPARK_Mode is
  procedure Acc (A : in out Natural;
                  V : Natural) is
  begin
    A := A + V;
  end Acc;

  A : Natural := 0;
  begin
    Acc (A, Natural'Last);
    Acc (A, 1);
  end Main;
```

You can now click on SPARK → Prove All, which opens a window with various options. For example, on this window, you can select the proof level — varying between 0 and 4 — on the Proof level list. Next, click on Execute. After the prover has completed its analysis, you'll see a list of issues found in the source code of your application.

For the example above, the prover complains about an overflow check that might fail. This is due to the fact that, in the Acc procedure, we're not dealing with the possibility that the result of the addition might be out of range. In order to fix this, we could define a new saturating addition Sat_Add that makes use of a custom type T with an extended range. For example:

```ada
procedure Main
  with SPARK_Mode is
  function Sat_Add (A : Natural;
                    V : Natural) return Natural
  is
    type T is range Natural'First .. Natural'Last * 2;

    A2 : T := T (A);
    V2 : constant T := T (V);
    A_Last : constant T := T (Natural'Last);
  begin
    A2 := A2 + V2;
    -- Saturate result if needed
```

(continues on next page)
if A2 > A_Last then
    A2 := A_Last;
end if;

return Natural (A2);
end Sat_Add;

procedure Acc (A : in out Natural;
                V : Natural) is
begin
    A := Sat_Add (A, V);
end Acc;

A : Natural := 0;
begin
    Acc (A, Natural'Last);
    Acc (A, 1);
end Main;

Now, when running the prover again with the modified code, no issues are found.
In chapter we present a brief overview of some of the tools included in the GNAT toolchain. For further details on how to use these tools, please refer to the GNAT User's Guide.

90.1 gnatchop

gnatchop renames files so they match the file structure and naming convention expected by the rest of the GNAT toolchain. The GNAT compiler expects specifications to be stored in .ads files and bodies (implementations) to be stored in .adb files. It also expects file names to correspond to the content of each file. For example, it expects the specification of a package Pkg.Child to be stored in a file named pkg-child.ads.

However, we may not want to use that convention for our project. For example, we may have multiple Ada packages contained in a single file. Consider a file example.ada containing the following:

```ada
with Ada.Text_IO; use Ada.Text_IO;

package P is
  procedure Test;
end P;

package body P is
  procedure Test is
  begin
    PutLine("Test passed.");
  end Test;
end P;

with P; use P;

procedure P_Main is
begin
  P.Test;
end P_Main;
```

To compile this code, we first pass the file containing our source code to gnatchop before we call gprbuild:

```
gnatchop example.ada

gprbuild p_main
```

This generates source files for our project, extracted from example.ada, that conform to the default naming convention and then builds the executable binary p_main from those

---

files. In this example `gnatchop` created the files `p.ads`, `p.adb`, and `p_main.adb` using the package names in `example.ada`.

When we use this mechanism, any warnings or errors the compiler displays refers to the files generated by `gnatchop`. We can, however, instruct `gnatchop` to instrument the generated files so the compiler refers to the original file (`example.ada` in our case) when displaying messages. We do this by using the `-r` switch:

```
gnatchop -r example.ada
gprbuild p_main
```

If, for example, we had an unused variable in `example.ada`, the compiler warning would now refer to the line in the original file, not in one of the generated ones.

For documentation of other switches available for `gnatchop`, please refer to the `gnatchop` chapter\(^{352}\) of the GNAT User's Guide.

## 90.2 gnatprep

We may want to use conditional compilation in some situations. For example, we might need a customized implementation of a package for a specific platform or need to select a specific version of an algorithm depending on the requirements of the target environment. A traditional way to do this uses a source-code preprocessor. However, in many cases where conditional compilation is needed, we can instead use the syntax of the Ada language or the functionality provided by `GPRbuild` to avoid using a preprocessor in those cases. The **conditional compilation section**\(^{353}\) of the GNAT User's Guide discusses how to do this in detail.

Nevertheless, using a preprocessor is often the most straightforward option in complex cases. When we encounter such a case, we can use `gnatprep`, which provides a syntax that reminds us of the C and C++ preprocessor. However, unlike in C and C++, this syntax is not part of the Ada standard and can only be used with `gnatprep`. Also, you'll notice some differences in the syntax from that preprocessor, such as shown in the example below:

```
#if VERSION'Defined and then (VERSION >= 4) then
   -- Implementation for version 4.0 and above...
#else
   -- Standard implementation for older versions...
#endif;
```

Of course, in this simple case, we could have used the Ada language directly and avoided the preprocessor entirely:

```
package Config is
   Version : constant Integer := 4;
end Config;

with Config;
procedure Do_Something is
begin
   if Config.Version >= 4 then
      null;
      -- Implementation for version 4.0 and above...
   else
      null;
   end if;
```


\(^{353}\) [https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#conditional-compilation](https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#conditional-compilation)
But for the sake of illustrating the use of \texttt{gnatprep}, let's use that tool in this simple case. This is the complete procedure, which we place in file \texttt{do_something.org.adb}:

\begin{verbatim}
procedure Do_Something is
begin
  #if VERSION'Defined and then (VERSION >= 4) then
    -- Implementation for version 4.0 and above...
    null;
  #else
    -- Standard implementation for older versions...
    null;
  #end if;
end Do_Something;
\end{verbatim}

To preprocess this file and build the application, we call \texttt{gnatprep} followed by \texttt{GPRbuild}:

\begin{verbatim}
gnatprep do_something.org.adb do_something.adb
gprbuild do_something
\end{verbatim}

If we look at the resulting file after preprocessing, we see that the \texttt{#else} implementation was selected by \texttt{gnatprep}. To cause it to select the newer "version" of the code, we include the symbol and its value in our call to \texttt{gnatprep}, just like we'd do for C/C++:

\begin{verbatim}
gnatprep -DVERSION=5 do_something.org.adb do_something.adb
\end{verbatim}

However, a cleaner approach is to create a symbol definition file containing all symbols we use in our implementation. Let's create the file and name it \texttt{prep.def}:

\begin{verbatim}
VERSION := 5
\end{verbatim}

Now we just need to pass it to \texttt{gnatprep}:

\begin{verbatim}
gnatprep do_something.org.adb do_something.adb prep.def
gprbuild do_something
\end{verbatim}

When we use \texttt{gnatprep} in that way, the line numbers of the output file differ from those of the input file. To preserve line numbers, we can use one of these command-line switches:

- \texttt{-b}: replace stripped-out code by blank lines
- \texttt{-c}: comment-out the stripped-out code

For example:

\begin{verbatim}
gnatprep -b do_something.org.adb do_something.adb prep.def
gnatprep -c do_something.org.adb do_something.adb prep.def
\end{verbatim}

When we use one of these options, \texttt{gnatprep} ensures that the output file \texttt{do_something.adb} has the same line numbering as the original file (\texttt{do_something.org.adb}).

The \texttt{gnatprep} chapter\textsuperscript{354} of the GNAT User's Guide contains further details about this tool, such as how to integrate \texttt{gnatprep} with project files for \texttt{GPRbuild} and how to replace symbols without using preprocessing directives (using the \$symbol syntax).

\textsuperscript{354} https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#preprocessing-with-gnatprep
90.3 gnatmem

Memory allocation errors involving mismatches between allocations and deallocations are a common source of memory leaks. To test an application for memory allocation issues, we can use gnatmem. This tool monitors all memory allocations in our application. We use this tool by linking our application to a special version of the memory allocation library (libgmem.a).

Let’s consider this simple example:

```
procedure Simple_Mem is
  I_Ptr : access Integer := new Integer;
begin
  null;
end Simple_Mem;
```

To generate a memory report for this code, we need to:

- Build the application, linking it to libgmem.a;
- Run the application, which generates an output file (gmem.out);
- Run gnatmem to generate a report from gmem.out.

For our example above, we do the following:

```
# Build application using gmem
gnatmake -g simple_mem.adb -largs -lgmem

# Run the application and generate gmem.out
./simple_mem

# Call gnatmem to display the memory report based on gmem.out
gnatmem simple_mem
```

For this example, gnatmem produces the following output:

```
Global information
------------------
Total number of allocations : 1
Total number of deallocations : 0
Final Water Mark (non freed mem) : 4 Bytes
High Water Mark : 4 Bytes

Allocation Root # 1
-------------------
Number of non freed allocations : 1
Final Water Mark (non freed mem) : 4 Bytes
High Water Mark : 4 Bytes
Backtrace :
  simple_mem.adb:2 simple_mem
```

This shows all the memory we allocated and tells us that we didn’t deallocate any of it.

Please refer to the chapter on gnatmem of the GNAT User’s Guide for a more detailed discussion of gnatmem.

---

355 https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_and_program_execution.html#the-gnatmem-tool
90.4 gnatmetric

We can use the GNAT metric tool (gnatmetric) to compute various programming metrics, either for individual files or for our complete project.

For example, we can compute the metrics of the body of package P above by running gnatmetric as follows:

```plaintext
gnatmetric p.adb
```

This produces the following output:

```
Line metrics summed over 1 units
  all lines : 13
  code lines : 11
  comment lines : 0
  end-of-line comments : 0
  comment percentage : 0.00
  blank lines : 2

Average lines in body: 4.00

Element metrics summed over 1 units
  all statements : 2
  all declarations : 3
  logical SLOC : 5

  2 subprogram bodies in 1 units

Average cyclomatic complexity: 1.00
```

Please refer to the section on gnatmetric\textsuperscript{356} of the GNAT User's Guide for the many switches available for gnatmetric, including the ability to generate reports in XML format.

90.5 gnatdoc

Use GNATdoc to generate HTML documentation for your project. It scans the source files in the project and extracts information from package, subprogram, and type declarations.

The simplest way to use it is to provide the name of the project or to invoke GNATdoc from a directory containing a project file:

```plaintext
gnatdoc -P some_directory/default.gpr
```

# Alternatively, when the :file:`default.gpr` file is in the same directory

```plaintext
gnatdoc
```

Just using this command is sufficient if your goal is to generate a list of the packages and a list of subprograms in each. However, to create more meaningful documentation, you can annotate your source code to add a description of each subprogram, parameter, and field. For example:

```plaintext
package P is
  -- Collection of auxiliary subprograms
```

\textsuperscript{356} https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-gnat-metrics-tool-gnatmetric
Learning Ada

(continued from previous page)

function Add_One
  (V : Integer
   -- Coefficient to be incremented
  ) return Integer;
  -- @return Coefficient incremented by one
end P;

package body P is

  function Add_One (V : Integer) return Integer is
    begin
      return V + 1;
    end Add_One;

end P;

with P; use P;

procedure Main is
  I : Integer;

begin
  I := Add_One (0);
end Main;

When we run this example, GNATdoc will extract the documentation from the specification of package P and add the description of each element, which we provided as a comment in the line below the actual declaration. It will also extract the package description, which we wrote as a comment in the line right after package P is. Finally, it will extract the documentation of function Add_One (both the description of the V parameter and the return value).

In addition to the approach we've just seen, GNATdoc also supports the tagged format that's commonly found in tools such as Javadoc and uses the @ syntax. We could rewrite the documentation for package P as follows:

package P is
  -- @summary Collection of auxiliary subprograms

    function Add_One
      (V : Integer
       -- @param V Coefficient to be incremented
      ) return Integer;
      -- @return Coefficient incremented by one
    end P;

You can control what parts of the source-code GNATdoc parses to extract the documentation. For example, you can specify the -b switch to request that the package body be parsed for additional documentation and you can use the -p switch to request GNATdoc to parse the private part of package specifications. For a complete list of switches, please refer to the GNATdoc User's Guide\footnote{http://docs.adacore.com/gnatdoc-docs/users_guide/_build/html/index.html}.
The term 'pretty-printing' refers to the process of formatting source code according to a pre-defined convention. **gnatpp** is used for the pretty-printing of Ada source-code files.

Let's look at this example, which contains very messy formatting:

```ada
PROCEDURE Main IS

  FUNCTION Init_2 RETURN INTEGER IS (2);
  I : INTEGER;
  BEGIN
    I := Init_2;
  END Main;

  FUNCTION Init_2 RETURN INTEGER IS (2);
  I : INTEGER;
  BEGIN
    I := Init_2;
  END;
```

We can request **gnatpp** to clean up this file by using the command:

```
gnatpp main.adb
```

**gnatpp** reformats the file in place. After this command, `main.adb` looks like this:

```ada
procedure Main is

  function Init_2 return Integer is (2);
  I : Integer;
  begin
    I := Init_2;
  end Main;
```

We can also process all source code files from a project at once by specifying a project file. For example:

```
gnatpp -P default.gpr
```

**gnatpp** has an extensive list of options, which allow for specifying the formatting of many aspects of the source and implementing many coding styles. These are extensively discussed in the section on **gnatpp**[^358] of the GNAT User's Guide.

[^358]: https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-gnat-pretty-printer-gnatpp
Suppose you’ve created a complex specification of an Ada package. You can create the corresponding package body by copying and adapting the content of the package specification. But you can also have `gnatstub` do much of that job for you. For example, let’s consider the following package specification:

```ada
class Aux is
   function Add_One (V : Integer) return Integer;
   procedure Reset (V : in out Integer);
end Aux;
```

We call `gnatstub`, passing the file containing the package specification:

```
gnatstub aux.ads
```

This generates the file `aux.adb` with the following contents:

```ada
pragma Ada_2012;
package body Aux is

   function Add_One (V : Integer) return Integer is
      begin
         -- Generated stub: replace with real body!
         pragma Compile_Time_Warning (Standard.True, "Add_One unimplemented");
         return raise Program_Error with "Unimplemented function Add_One";
      end Add_One;

   procedure Reset (V : in out Integer) is
      begin
         -- Generated stub: replace with real body!
         pragma Compile_Time_Warning (Standard.True, "Reset unimplemented");
         raise Program_Error with "Unimplemented procedure Reset";
      end Reset;

end Aux;
```

As we can see in this example, not only has `gnatstub` created a package body from all the elements in the package specification, but it also created:

- Headers for each subprogram (as comments);
- Pragmas and exceptions that prevent us from using the unimplemented subprograms in our application.

This is a good starting point for the implementation of the body. Please refer to the section on `gnatstub`\(^359\) of the GNAT User’s Guide for a detailed discussion of `gnatstub` and its options.

\(^359\) https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-body-stub-generator-gnatstub
Part X

Guidelines for Safe and Secure Ada/SPARK
Copyright © 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details on this page.360

This document provides a reasonable set of coding standards to be applied to Ada/SPARK source code. The contents can be used as-is, or customized for a particular project.

This document was originally written by Patrick Rogers, and modified by Michael Frank.

360 http://creativecommons.org/licenses/by-sa/4.0
INTRODUCTION

Ada is a general purpose, high-level programming language designed to support the construction of long-lived, highly-reliable applications. Like all general-purpose languages, only a subset of the full language is appropriate for safety-critical applications because the full language includes facilities that are difficult to analyze and verify to the degree required. This document facilitates identification of subsets appropriate for the highest levels of integrity, including safety-critical applications.

SPARK is a statically verifiable subset of Ada designed specifically for the most critical applications. Ada constructs not amenable to verification are precluded, such as arbitrary use of access types and full tasking. SPARK is also a superset of Ada, with additional contracts for specifying and verifying programs. Many of the guidelines (and more) are implicit in the design of SPARK.

Therefore, this document defines guidelines for the development of high-integrity, safety-critical applications in either the Ada or SPARK programming languages, or both (because the two can be mixed).

91.1 Scope

This document provides guidelines for development decisions, both at the system level and at the unit level, regarding the use of the programming languages Ada and SPARK, as well as related tools, such as static analyzers and unit test generators. It is not concerned with presentation issues such as naming, use of whitespace, or the like.

91.2 Structure

Rather than defining a specific set of rules defining a single subset, this document defines a set of criteria, in the form of guidelines, used by system architects to identify project-specific subsets appropriate to a given project.

The guidelines are separated into related categories, such as storage management, object-oriented programming, concurrency management, and so on. Each guideline is in a separate table, specifying the rule name, a unique identifier, and additional attributes common to each table.
91.3 Enforcement

Detection and enforcement mechanisms are indicated for each guideline. These mechanisms typically consist of the application of a language standard pragma named Restrictions, with policy-specific restriction identifiers given as parameters to the pragma [AdaRM2016]. Violations of the given restrictions are then detected and enforced by the Ada compiler.

Alternatively, the AdaCore GNATcheck utility program has rules precisely corresponding to those restriction identifiers, with the same degree of detection and enforcement. For example, the language restriction identifier No Unchecked Deallocation corresponds to the GNATcheck +RRestrictions:No_Unchecked_Deallocation rule.

The advantage of GNATcheck over the compiler is that all generated messages will be collected in the GNATcheck report that can be used as evidence of the level of adherence to the coding standard. In addition, GNATcheck provides a mechanism to deal with accepted exemptions.

In some cases the enforcement mechanism is the SPARK language and analyzer. Many of the guidelines (and more) are implicit in the design of SPARK and are, therefore, automatically enforced.

In some (very) rare cases the enforcement mechanism is manual program inspection, although alternatives (e.g., SPARK) are usually available and recommended. These guidelines are included because they are considered invaluable in this domain.

91.4 About the Rules

Although we refer to them as rules in the tables for the sake of brevity, these entries should be considered guidance because they require both thought and consideration of project-specific characteristics. For example, in some cases the guidance is to make a selection from among a set of distinct enumerated policies. In other cases a single guideline should be followed but not without some exceptional situations allowing it to be violated. The project lead should consider which guidelines to apply and how best to apply each guideline selected.

Many of these rules can also be considered good programming practices. As such, many of them can be directly correlated to the ISO/IEC Guidance to Avoiding Vulnerabilities in Programming Languages [TR24772]. When a rule addresses one of these vulnerabilities, it is listed in the appropriate subsection.
DEFINITIONS

This section contains terms and values used in the definitions of the rules set forth in this chapter.

92.1 Level

**Level** is the compliance level for the rule. Possible values are:

- **Mandatory**
  Non-compliance with a *Mandatory* recommendation level corresponds to a **high risk** of a software bug. There would need to be a good reason for non-conformity to a mandatory rule and, although it is accepted that exceptional cases may exist, any non-conformance should be accompanied by a clear technical explanation of the exceptional circumstance.

- **Required**
  Non-compliance with a *Required* recommendation level corresponds to a **medium to high risk** of a software bug. Much like a *Mandatory* recommendation, there would need to be a good reason for non-conformity to a required rule. Although it is accepted that more exceptional cases may exist, non-conformance should be accompanied by a clear technical explanation of the exceptional circumstance.

- **Advisory**
  Failure to follow an *Advisory* recommendation does not necessarily result in a software bug; the risk of a direct correlation between non-conformance of an advisory rule and a software bug is low. Non-compliance with an advisory recommendation level does not require a supporting technical explanation, however, as the quality of the code may be impacted, the reason for the non-conformance should be understood.

92.2 Remediation

**Remediation** indicates the the level of difficulty to modify/update code that does not follow this particular rule.

- **High**
  Failure to follow this rule will likely cause an unreasonable amount of modifications/updates to bring the code base into compliance.

- **Medium**
  Failure to follow this rule will likely cause a large amount of modifications/updates to bring the code base into compliance, but those changes may still be cost-effective.
Low
Failure to follow this rule may cause a small amount of modifications/updates to bring the code base into compliance, but those changes will be minor compared to the benefit.

N/A
This rule is more of a design decision (as opposed to a coding flaw) and therefore, if the rule is violated, it is done so with a specific purpose.
CHAPTER
NINETYTHREE

DYNAMIC STORAGE MANAGEMENT (DYN)

Goal

- Maintainability ✓
- Reliability ✓
- Portability
- Performance ✓
- Security ✓

Description

Have a plan for managing dynamic memory allocation and deallocation.

Rules

DYN01, DYN02, DYN03, DYN04, DYN05, DYN06

In Ada, objects are created by being either declared or allocated. Declared objects may be informally thought of as being created "on the stack" although such details are not specified by the language. Allocated objects may be thought of as being allocated "from the heap", which is, again, an informal term. Allocated objects are created by the evaluation of allocators represented by the reserved word new and, unlike declared objects, have lifetimes independent of scope.

The terms static and dynamic tend to be used in place of declared and allocated, although in traditional storage management terminology all storage allocation in Ada is dynamic. In the following discussion, the term dynamic allocation refers to storage that is allocated by allocators. Static object allocation refers to objects that are declared. Deallocation refers to the reclamation of allocated storage.

Unmanaged dynamic storage allocation and deallocation can lead to storage exhaustion; the required analysis is difficult under those circumstances. Furthermore, access values can establish aliases that complicate verification, and explicit deallocation of dynamic storage can lead to specific errors (e.g., "double free", "use after free") having unpredictable results. As a result, the prevalent approach to storage management in high-integrity systems is to disallow dynamic management techniques completely. [SEI-C] [MISRA2013] [Holzmann2006] [ISO2000]

However, restricted forms of storage management and associated feature usage can support the necessary reliability and analyzability characteristics while retaining sufficient expressive power to justify the analysis expense. The following sections present possible approaches, including the traditional approach in which no dynamic behavior is allowed. Individual projects may then choose which storage management approach best fits their requirements and apply appropriate tailoring, if necessary, to the specific guidelines.

Realization

There is a spectrum of management schemes possible, trading ease of analysis against
increasing expressive power. At one end there is no dynamic memory allocation (and hence, deallocation) allowed, making analysis trivial. At the other end, nearly the full expressive power of the Ada facility is available, but with analyzability partially retained. In the latter, however, the user must create the allocators in such a manner as to ensure proper behavior.

Rule DYN01 is Required, as it avoids problematic features whatever the strategy chosen. Rules DYN02-05 are marked as Advisory, because one of them should be chosen and enforced throughout a given software project.

93.1 Common High Integrity Restrictions (DYN01)

Level → Required
Category
  Safety ✓
  Cyber ✓
Goal
  Maintainability ✓
  Reliability ✓
  Portability
  Performance
  Security ✓
Remediation → Low
Verification Method → Compiler restrictions

93.1.1 Reference

Ada Reference Manual: H.4 High Integrity Restrictions

93.1.2 Description

The following restrictions must be in effect:
  • No_Anonymous_Allocators
  • No_Coextensions
  • No_Access_Parameter_Allocators
  • Immediate_Reclamation

The first three restrictions prevent problematic usage that, for example, may cause unreclaimed (and unreclaimable) storage. The last restriction ensures any storage allocated by the compiler at run-time for representing objects is reclaimed at once. (That restriction does not apply to objects created by allocators in the application.)

93.1.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

93.1.4 Noncompliant Code Example

For No_Anonymous_Allocators:

```ada
X : access String := new String'("Hello");
...
X := new String'("Hello");
```

For No_Coextensions:

```ada
type Object (Msg : access String) is ...
Obj : Object (Msg => new String'("Hello"));
```

For No_Access_Parameter_Allocators:

```ada
procedure P (Formal : access String);
...
P (Formal => new String'("Hello"));
```

93.1.5 Compliant Code Example

For No_Anonymous_Allocators, use a named access type:

```ada
type String_Reference is access all String;
S : constant String_Reference := new String'("Hello");
X : access String := S;
...
X := S;
```

For No_Coextensions, use a variable of a named access type:

```ada
type Object (Msg : access String) is ...
type String_Reference is access all String;
S : String_Reference := new String'("Hello");
Obj : Object (Msg => S);
```

For No_Access_Parameter_Allocators, use a variable of a named access type:

```ada
procedure P (Formal : access String);
type String_Reference is access all String;
S : String_Reference := new String'("Hello");
...
P (Formal => S);
```
93.1.6 Notes

The compiler will detect violations of the first three restrictions. Note that GNATcheck can detect violations in addition to the compiler.

The fourth restriction is a directive for implementation behavior, not subject to source-based violation detection.

93.2 Traditional Static Allocation Policy (DYN02)

Level → Advisory
Category
  Safety ✓
  Cyber ✓
Goal
  Maintainability ✓
  Reliability ✓
  Portability
  Performance
  Security ✓
Remediation → Low
Verification Method → Compiler restrictions

93.2.1 Reference

MISRAC Dir 4.12: "Dynamic memory allocation shall not be used."

93.2.2 Description

The following restrictions must be in effect:
  • No_Allocators
  • No_Task_Allocators

Under the traditional approach, no dynamic allocations and no deallocations occur. Only declared objects are used and no access types of any kind appear in the code.

Without allocations there is no issue with deallocation as there would be nothing to deallocate. Heap storage exhaustion and fragmentation are clearly prevented although storage may still be exhausted due to insufficient stack size allotments.

In this approach the following constructs are not allowed:
  • Allocators
  • Access-to-constant access types
• Access-to-variable access types
• User-defined storage pools
• Unchecked Deallocations

93.2.3 Applicable Vulnerability within ISO TR 24772-2

• 4.10 Storage Pool

93.2.4 Noncompliant Code Example

Any code using the constructs listed above.

93.2.5 Compliant Code Example

N/A

93.2.6 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

93.3 Access Types Without Allocators Policy (DYN03)

Level → Advisory
Category
  Safety ✓
  Cyber ✓
Goal
  Maintainability ✓
  Reliability ✓
  Portability
  Performance
  Security ✓
Remediation → Low
Verification Method → Compiler restrictions
93.3.1 Reference

MISRA C Rule 21.3: “The memory allocation and deallocation functions of <stdlib.h> shall not be used.”

93.3.2 Description

The following restrictions must be in effect:

- No_Allocators
- No_Dependence => Ada.Unchecked_Deallocation

In this approach dynamic access values are only created via the attribute 'Access applied to aliased objects. Allocation and deallocation never occur. As a result, storage exhaustion cannot occur because no dynamic allocations occur. Fragmentation cannot occur because there are no deallocations.

In this approach the following constructs are not allowed:

- Allocators
- User-defined storage pools
- Unchecked Deallocations

Aspects should be applied to all access types in this approach, specifying a value of zero for the storage size. Although the restriction No_Allocators is present, such clauses may be necessary to prevent any default storage pools from being allocated for the access types, even though the pools would never be used. A direct way to accomplish this is to use pragma Default_Storage_Pool with a parameter of null like so:

```
pragma Default_Storage_Pool (null);
```

The above would also ensure no allocations can occur with access types that have the default pool as their associated storage pool (per Ada Reference Manual: 13.11.3 (6.1/3) Default Storage Pools).³⁶²

93.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.14 Dangling reference to heap [XYK]

93.3.4 Noncompliant Code Example

Any code using the constructs listed above.

93.3.5 Compliant Code Example

```ada
type Descriptor is ...;
type Descriptor_Ref is access all Descriptor;
...
Device : aliased Descriptor;
...
P : Descriptor_Ref := Device'Access;
...
```

93.3.6 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

93.4 Minimal Dynamic Allocation Policy (DYN04)

**Level** → Advisory  

**Category**  

- **Safety** ✓  
- **Cyber** ✓  

**Goal**  

- **Maintainability** ✓  
- **Reliability** ✓  
- **Portability**  
- **Performance**  
- **Security**  

**Remediation** → Low  

**Verification Method** → Compiler restrictions

93.4.1 Reference

Power of Ten rule 3: "Do not use dynamic memory allocation after initialization."
93.4.2 Description

The following restrictions must be in effect:

- No_Local_Allocators
- No_Dependence => Ada.Unchecked_Deallocation

In this approach dynamic allocation is only allowed during "start-up" and no later. Deallocation never occur. As a result, storage exhaustion should never occur assuming the initial allotment is sufficient. This assumption is as strong as when using only declared objects on the "stack" because in that case a sufficient initial storage allotment for the stack must be made.

In this approach the following constructs are not allowed:

- Unchecked Deallocations

Note that some operating systems intended for this domain directly support this policy.

93.4.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

93.4.4 Noncompliant Code Example

Any code using the constructs listed above.

93.4.5 Compliant Code Example

Code performing dynamic allocations any time prior to an arbitrary point designated as the end of the "startup" interval.

93.4.6 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

93.5 User-Defined Storage Pools Policy (DYN05)

Level → Advisory

Category

- Safety ✓
- Cyber ✓

Goal

- Maintainability ✓
- Reliability ✓
93.5.1 Reference

MISRA C Rule 21.3: "The memory allocation and deallocation functions of <stdlib.h> shall not be used."

93.5.2 Description

There are two issues that make storage utilization analysis difficult:

1. the predictability of the allocation and deallocation implementation, and
2. how access values are used by the application.

The behavior of the underlying implementation is largely undefined and may, for example, consist of calls to the operating system (if present). Application code can manipulate access values beyond the scope of analysis.

Under this policy, the full expressive power of access-to-object types is provided but one of the two areas of analysis difficulty is removed. Specifically, predictability of the allocation and deallocation implementation is achieved via user-defined storage pools. With these storage pools, the implementation of allocation (new) and deallocation (instances of Ada.Unchecked_Deallocation) is defined by the pool type.

If the pool type is implemented with fixed-size blocks on the stack, allocation and deallocation timing behavior are predictable.

Such an implementation would also be free from fragmentation.

Given an analysis providing the worst-case allocations and deallocations, it would be possible to verify that pool exhaustion does not occur. However, as mentioned such analysis can be quite difficult. A mitigation would be the use of the "owning" access-to-object types provided by SPARK.

In this approach no storage-related constructs are disallowed unless the SPARK subset is applied.

93.5.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool
93.5.4 Noncompliant Code Example

Allocation via an access type not tied to a user-defined storage pool.

93.5.5 Compliant Code Example

```ada
Heap : Sequential_Fixed_Blocks.Storage_Pool
   (Storage_Size => Required_Storage_Size,
    Element_Size => Representable_Obj_Size,
    Alignment => Representation_Alignment);

type Pointer is access all Unsigned_Longword with
   Storage_Pool => Heap;
Ptr : Pointer;
...
Ptr := new Unsigned_Longword; -- from Heap
```

93.5.6 Notes

Enforcement of this approach can only be provided by manual code review unless SPARK is used.

However, the User-Defined Storage Pools Policy can be enforced statically by specifying Default_Storage_Pool (null). This essentially requires all access types to have a specified storage pool if any allocators are used with the access type.

93.6 Statically Determine Maximum Stack Requirements (DYN06)

Level → Required

Category

- Safety ✓
- Cyber ✓

Goal

- Maintainability ✓
- Reliability ✓
- Portability
- Performance
- Security

Remediation → Low

Verification Method → Static analysis tools
93.6.1 Reference

N/A

93.6.2 Description

Each Ada application task has a stack, as does the "environment task" that elaborates library packages and calls the main subprogram. A tool to statically determine the maximum storage required for these stacks must be used, per task.

This guideline concerns another kind of dynamic memory utilization. The previous guidelines concerned the management of storage commonly referred to as the "heap." This guideline concerns the storage commonly referred to as the "stack." (Neither term is defined by the language, but both are commonly recognized and are artifacts of the underlying run-time library or operating system implementation.)

93.6.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

93.6.4 Noncompliant Code Example

N/A

93.6.5 Compliant Code Example

N/A

93.6.6 Notes

The GNATstack\textsuperscript{363} tool can statically determine the maximum requirements per task.

\textsuperscript{363} http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html
SAFE RECLAMATION (RCL)

Goal

Maintainability ✓
Reliability ✓
Portability
Performance ✓
Security ✓

Description
Related to managing dynamic storage at the system (policy) level, these statement-level rules concern the safe reclamation of access (pointer) values.

Rules
RCL01, RCL02, RCL03

94.1 No Multiple Reclamations (RCL01)

Level → Mandatory
Category

Safety ✓
Cyber ✓

Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance
Security ✓
Remediation → High
Verification Method → Code inspection

94.1.1 Reference

[CWE2019] CWE-415: Double Free

94.1.2 Description

Never deallocate the storage designated by a given access value more than once.

94.1.3 Applicable Vulnerability within ISO TR 24772-2

• 6.39 Memory leak and heap fragmentation [XYL]

94.1.4 Noncompliant Code Example

```ada
type String_Reference is access all String;
procedure Free is new Ada.Unchecked_Decommissioning
   (Object => String, Name => String_Reference);
S : String_Reference := new String('Hello');
Y : String_Reference;
begin
   Y := S;
   Free (S);
   Free (Y);
```

94.1.5 Compliant Code Example

Remove the call to Free (Y).

94.1.6 Notes

Enforcement of this rule can be provided by manual code review, unless deallocation is forbidden via No_Unchecked_Deallocorion or SPARK is used, as ownership analysis in SPARK detects such cases. Note that storage utilization analysis tools such as Valgrind can usually find this sort of error. In addition, a GNAT-defined storage pool is available to help debug such errors.
94.2 Only Reclaim Allocated Storage (RCL02)

Level → Mandatory
Category
  
Safety ✓

Cyber ✓

Goal
  
Maintainability ✓

Reliability ✓

Portability ✓

Performance

Security ✓

Remediation → High

Verification Method → Code inspection

94.2.1 Reference

[SEI-C] MEM34-C: Only Free Memory Allocated Dynamically

94.2.2 Description

Only deallocate storage that was dynamically allocated by the evaluation of an allocator (i.e., new).

This is possible because Ada allows creation of access values designating declared (aliased) objects.

94.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

94.2.4 Noncompliant Code Example

```ada
type String_Reference is access all String;
procedure Free is new Ada.Unchecked_Deallocation
  (Object => String, Name => String_Reference);
S : aliased String := "Hello";
Y : String_Reference := S'Access;
begin
  Free (Y);
```

94.2. Only Reclaim Allocated Storage (RCL02)
94.2.5 Compliant Code Example

Remove the call to Free (Y).

94.2.6 Notes

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via No_Unchecked_Deallocation.

94.3 Only Reclaim to the Same Pool (RCL03)

Level → Mandatory

Category

Safety ✓
Cyber ✓

Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance Security ✓

Remediation → High

Verification Method → Code inspection

94.3.1 Reference

N/A

94.3.2 Description

When deallocating, ensure that the pool to which the storage will be returned is the same pool from which it was allocated. Execution is erroneous otherwise, meaning anything can happen (Ada Reference Manual: 13.11.2 (16) Unchecked Storage Deallocation364).

Each access type has an associated storage pool, either implicitly by default, or explicitly with a storage pool specified by the programmer. The implicit default pool might not be the same pool used for another access type, even an access type designating the same subtype.

94.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

94.3.4 Noncompliant Code Example

```ada
type Pointer1 is access all Integer;
type Pointer2 is access all Integer;
P1 : Pointer1;
P2 : Pointer2;
procedure Free is new Ada.Unchecked_Deallocation
    (Object => Integer, Name => Pointer2);
begin
    P1 := new Integer;
P2 := Pointer2 (P1);
    Call_Something ( P2.all );
    ...
    Free (P2);
```

In the above, P1.all was allocated from Pointer1'Storage_Pool, but, via the type conversion, the code above is attempting to return it to Pointer2'Storage_Pool, which may be a different pool.

94.3.5 Compliant Code Example

```ada
type Pointer1 is access all Integer;
type Pointer2 is access all Integer;
P1 : Pointer1;
P2 : Pointer2;
procedure Free is new Ada.Unchecked_Deallocation
    (Object => Integer, Name => Pointer1);
begin
    P1 := new Integer;
P2 := Pointer2 (P1);
    Call_Something ( P2.all );
    ...
    Free (P1);
```

94.3.6 Notes

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via No_Unchecked_Deallocation.
Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance ✓
Security

Description
Have a plan for managing the use of concurrency in high-integrity applications having real-time requirements.

Rules
CON01, CON02, CON03

The canonical approach to applications having multiple periodic and aperiodic activities is to map those activities onto independent tasks, i.e., threads of control. The advantages for the application are both a matter of software engineering and also ease of implementation. For example, when the different periods are not harmonics of one another, the fact that each task executes independently means that the differences are trivially represented. In contrast, such periods are not easily implemented in a cyclic scheduler, which, by definition, involves only one (implicit) thread of control with one frame rate.

High integrity applications are subject to a number of stringent analyses, including, for example, safety analyses and certification against rigorous industry standards. In addition, high integrity applications with real-time requirements must undergo timing analysis because they must be shown to meet deadlines prior to deployment — failure to meet hard deadlines is unacceptable in this domain.

These analyses are applied both to the application and to the implementation of the underlying run-time library. However, analysis of the complete set of general Ada tasking features is not tractable, neither technically nor in terms of cost. A subset of the language is required.

The Ravenscar profile [AdaRM2016] is a subset of the Ada concurrency facilities that supports determinism, schedulability analysis, constrained memory utilization, and certification to the highest integrity levels. Four distinct application domains are specifically intended:

- Hard real-time applications requiring predictability;
- Safety-critical systems requiring formal, stringent certification;
- High-integrity applications requiring formal static analysis and verification;
• Embedded applications requiring both a small memory footprint and low execution overhead.

Those tasking constructs that preclude analysis at the source level or analysis of the tasking portion of the underlying run-time library are disallowed.

The Ravenscar profile is necessarily strict in terms of what it removes so that it can support the stringent analyses, such as safety analysis, that go beyond the timing analysis required for real-time applications. In addition, the strict subset facilitates that timing analysis in the first place.

However, not all high-integrity applications are amenable to expression in the Ravenscar profile subset. The Jorvik profile [AdaRM2020] is an alternative subset of the Ada concurrency facilities. It is based directly on the Ravenscar profile but removes selected restrictions in order to increase expressive power, while retaining analyzability and performance. As a result, typical idioms for protected objects can be used, for example, and relative delays statements are allowed. Timing analysis is still possible but slightly more complicated, and the underlying run-time library is slightly larger and more complex.

When the most stringent analyses are required and the tightest timing is involved, use the Ravenscar profile. When a slight increase in complexity is tolerable, i.e., in those cases not undergoing all of these stringent analyses, consider using the Jorvik profile.

**95.1 Use the Ravenscar Profile (CON01)**

**Level** → Advisory

**Category**

Safety ✓

Cyber ✓

**Goal**

Maintainability ✓

Reliability ✓

Portability ✓

Performance ✓

Security

**Remediation** → High

**Verification Method** → GNATcheck rule: uses_profile: ravenscar

**Mutually Exclusive** → CON02
95.1.1 Reference


95.1.2 Description

The following profile must be in effect:

```ada
pragma Profile (Ravenscar);
```

The profile is equivalent to the following set of pragmas:

```ada
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (  
  No_Abort_Statements,  
  No_Dynamic_Attachment,  
  No_Dynamic_CPU_Assignment,  
  No_Dynamic_Priorities,  
  No_Implicit_Heap_Allocations,  
  No_Local_Protected_Objects,  
  No_Local_Timing_Events,  
  No_Protected_Type_Allocators,  
  No_Relative_Delay,  
  No_Requeue_Statements,  
  No_Select_Statements,  
  No_Specific_Termination_Handlers,  
  No_Task_Allocators,  
  No_Task_Hierarchy,  
  No_Task_Termination,  
  Simple_Barriers,  
  Max_Entry_Queue_Length => 1,  
  Max_Protected_Entries => 1,  
  Max_Task_Entries => 0,  
  No_Dependence => Ada.Asynchronous_Task_Control,  
  No_Dependence => Ada.Calendar,  
  No_Dependence => Ada.Execution_Time.Group_Budgets,  
  No_Dependence => Ada.Execution_Time.Timers,  
  No_Dependence => Ada.Synchronous_Barriers,  
  No_Dependence => Ada.Task_Attributes,  
  No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

95.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.59 Concurrency - Activation [GGA]
- 6.60 Concurrency - Directed termination [CGT]
- 6.61 Concurrent data access [CGX]
- 6.62 Concurrency - Premature termination [CGS]
- 6.63 Lock protocol errors [CGM]

95.1.4 Noncompliant Code Example

Any code disallowed by the profile. Remediation is high because use of the facilities outside the subset can be difficult to retrofit into compliance.

```ada
task body Task_T is
begin
  loop
    -- Error: No_Relative_Delay
    delay 1.0;
    Put_Line ("Hello World");
  end loop;
end Task_T;
```

95.1.5 Compliant Code Example

```ada
task body Task_T is
  Period : constant Time_Span := Milliseconds (10);
  Activation : Time := Clock;
begin
  loop
    delay until Activation;
    Put_Line ("Hello World");
    Activation := Activation + Period;
  end loop;
end Task_T;
```

95.1.6 Notes

The Ada builder will detect violations if the programmer specifies this profile or corresponding pragmas. GNATcheck also can detect violations of profile restrictions.

95.2 Use the Jorvik Profile (CON02)

**Level** → Advisory

**Category**

- **Safety**
  - ✓
- **Cyber**
  - ✓

**Goal**

- **Maintainability**
  - ✓
- **Reliability**
  - ✓
- **Portability**
  - ✓
- **Performance**
  - ✓
Security

Remediation → High

Verification Method → GNATcheck rule: uses_profile:jorvik

Mutually Exclusive → CON01

95.2.1 Reference


95.2.2 Description

The following profile must be in effect:

```
pragma Profile (Jorvik);
```

The profile is equivalent to the following set of pragmas:

```
pragma TaskDispatchingPolicy (FIFO_Within_Priorities);
pragma LockingPolicy (Ceiling_Locking);
pragma DetectBlocking;
pragma Restrictions (  
  No_Abort_Statements,  
  No_Dynamic_Attachment,  
  No_Dynamic_CPU_Assignment,  
  No_Dynamic_Priorities,  
  No_Local_Protected_Objects,  
  No_Local_Timing_Events,  
  No_Protected_Type_Allocators,  
  No_Requeue_Statements,  
  No_Select_Statements,  
  No_Specific_Termination_Handlers,  
  No_Task_Allocators,  
  No_Task_Hierarchy,  
  No_Task_Termination,  
  Pure_Barriers,  
  Max_Task_Entries => 0,  
  No_Dependence => Ada.Asynchronous_Task_Control,  
  No_Dependence => Ada.Execution_Time.Group_Budgets,  
  No_Dependence => Ada.Execution_Time.Timers,  
  No_Dependence => Ada.Task_Attributes,  
  No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

The following restrictions are part of the Ravenscar profile but not part of the Jorvik profile.

```
No_Implicit_Heap_Allocations
No_Relative_Delay
Max_Entry_Queue_Length => 1
Max_Protected_Entries => 1
No_Dependence => Ada.Calendar
No_Dependence => Ada.Synchronous_Barriers
```

Jorvik also replaces restriction SimpleBarItemriers with Pure_Barriers (a weaker requirement than the restriction Simple_Barriers).

---


95.2. Use the Jorvik Profile (CON02)
95.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.59 Concurrency - Activation [GGA]
- 6.60 Concurrency - Directed termination [CGT]
- 6.61 Concurrent data access [CGX]
- 6.62 Concurrency - Premature termination [CGS]
- 6.63 Lock protocol errors [CGM]

95.2.4 Noncompliant Code Example

Any code disallowed by the profile. Remediation is **high** because use of the facilities outside the subset can be difficult to retrofit into compliance.

```ada
task body Task_T is
begin
    -- Error: Max_Task_Entries => 0
    accept Entry_Point do
        Put_Line ("Hello World");
    end Entry_Point;
    loop
        delay 1.0;
        Put_Line ("Ping");
    end loop;
end Task_T;
```

95.2.5 Compliant Code Example

```ada
task body Task_T is
begin
    delay 1.0;
    Put_Line ("Hello World");
    loop
        delay 1.0;
        Put_Line ("Ping");
    end loop;
end Task_T;
```

95.2.6 Notes

The Ada builder will detect violations. GNATcheck can also detect violations.

95.3 Avoid Shared Variables for Inter-task Communication (CON03)

**Level** → Advisory  
**Category**

**Safety** ✓
Cyber ✓

Goal

Maintainability ✓

Reliability ✓

Portability ✓

Performance ✓

Security

Remediation → High

Verification Method → GNATCheck rule: Volatile_Objects_Without_Address_Clauses

95.3.1 Reference


95.3.2 Description

Although the Ravenscar and Jorvik profiles allow the use of shared variables for inter-task communication, such use is less robust and less reliable than encapsulating shared variables within protected objects.

95.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.56 Undefined behaviour [EWF]

95.3.4 Noncompliant Code Example

```ada
Global_Object : Integer
   with Volatile;
function Get return Integer is (Global_Object);
```

Note that variables marked as Atomic are also Volatile, per the Ada Reference Manual: C.6 (8/3) Shared Variable Control

---

368 http://www.ada-auth.org/standards/12rm/html/RM-C-6.html
95.3.5 Compliant Code Example

When assigned to a memory address, a Volatile variable can be used to interact with a memory-mapped device, among other similar usages.

```ada
Global_Object : Integer
   with Volatile,
   Address => To_Address (16#1234_5678#);
function Get return Integer is (Global_Object);
```

95.3.6 Notes

In addition to GNATcheck, SPARK and CodePeer can also detect conflicting access to unprotected variables.
ROBUST PROGRAMMING PRACTICE (RPP)

Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance ✓
Security ✓

Description

These rules promote the production of robust software.

Rules

RPP01, RPP02, RPP03, RPP04, RPP05, RPP06, RPP07, RPP08, RPP09, RPP10, RPP11, RPP12, RPP13, RPP14

96.1 No Use of "others" in Case Constructs (RPP01)

Level → Required

Category

Safety ✓
Cyber ✓

Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance
Security
Remediation → Low
Verification Method → GNATcheck rule: OTHERS_In_CASE_Statements

96.1.1 Reference

[SEI-C] MSC01-C

96.1.2 Description

Case statement alternatives and case-expressions must not include use of the others discrete choice option. This rule prevents accidental coverage of a choice added after the initial case statement is written, when an explicit handler was intended for the addition.

Note that this is opposite to typical C guidelines such as [SEI-C] MSC01-C. The reason is that in C, the default alternative plays the role of defensive code to mitigate the switch statement's non-exhaustivity. In Ada, the case construct is exhaustive: the compiler statically verifies that for every possible value of the case expression there is a branch alternative, and there is also a dynamic check against invalid values which serves as implicit defensive code. As a result, Ada's others alternative doesn't play C's defensive code role and therefore a stronger guideline can be adopted.

96.1.3 Applicable Vulnerability within ISO TR 24772-2

• 6.27 Switch statements and static analysis [CLL]

96.1.4 Noncompliant Code Example

```ada
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'succ (C);
  when others =>
    C := Character'pred (C);
end case;
```

96.1.5 Compliant Code Example

```ada
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'succ (C);
  when '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' =>
    C := Character'pred (C);
end case;
```
96.1.6 Notes

N/A

96.2 No Enumeration Ranges in Case Constructs (RPP02)

Level → Required
Category
   Safety
      ✓
   Cyber
      ✓
Goal
   Maintainability
      ✓
   Reliability
      ✓
   Portability
      ✓
   Performance
   Security
Remediation → Low
Verification Method → GNATcheck rule: Enumeration_Ranges_In_CASE_Statements

96.2.1 Reference

Similar to RPP01

96.2.2 Description

A range of enumeration literals must not be used as a choice in a case statement or a case expression. This includes explicit ranges (A .. B), subtypes, and the 'Range attribute. Much like the use of others in case statement alternatives, the use of ranges makes it possible for a new enumeration value to be added but not handled with a specific alternative, when a specific alternative was intended.
96.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.5 Enumerator issues [CCB]

96.2.4 Noncompliant Code Example

```ada
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'SucC (C);
  when '1' .. '8' =>
    C := Character'Pred (C);
end case;
```

96.2.5 Compliant Code Example

```ada
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'SucC (C);
  when '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' =>
    C := Character'Pred (C);
end case;
```

96.2.6 Notes

N/A

96.3 Limited Use of "others" in Aggregates (RPP03)

**Level** → Advisory

**Category**

- Safety ✓
- Cyber ✓

**Goal**

- Maintainability ✓
- Reliability ✓
- Portability ✓
- Performance
- Security

**Remediation** → Low

**Verification Method** → GNATcheck rule: OTHERS_In_Aggregates
96.3.1 Reference

Similar to RPP01

96.3.2 Description

Do not use an `others` choice in an extension aggregate. In `record` and `array` aggregates, do not use an `others` choice unless it is used either to refer to all components, or to all but one component.

This guideline prevents accidental provision of a general value for a `record` component or `array` component, when a specific value was intended. This possibility includes the case in which new components are added to an existing composite type.

96.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.5 Enumerator issues [CCB]
- 6.27 Switch statements and static analysis [CLL]

96.3.4 Noncompliant Code Example

```ada
type Record_T is record
  Field1 : Integer   := 1;
  Field2 : Boolean   := False;
  Field3 : Character := ' ';
end record;
type Array_T is array (Character) of Boolean;
Rec : Record_T := (Field1 => 1,
                   Field3 => '2',
                   others => <>);
Arr : Array_T := ('0' .. '9' => True,
                   others => False);
```

96.3.5 Compliant Code Example

```ada
type Record_T is record
  Field1 : Integer   := 1;
  Field2 : Boolean   := False;
  Field3 : Character := ' ';
end record;
type Array_T is array (Character) of Boolean;
Rec : Record_T := (Field1 => 1,
                   others => <>);
Arr : Array_T := (others => False);
```
96.3.6 Notes

N/A

96.4 No Unassigned Mode-Out Procedure Parameters (RPP04)

Level → Required
Category
  Safety ✓
  Cyber ✓
Goal
  Maintainability ✓
  Reliability ✓
  Portability ✓
  Performance
  Security
Remediation → High
Verification Method → GNATcheck rule: Unassigned_OUT_Parameters

96.4.1 Reference
MISRA C Rule 9.1: "The value of an object with automatic storage duration shall not be read before it has been set."

96.4.2 Description
For any procedure, all formal parameters of mode out must be assigned a value if the procedure exits normally. This rule ensures that, upon a normal return, the corresponding actual parameter has a defined value. Ensuring a defined value is especially important for scalar parameters because they are passed by value, such that some value is copied out to the actual. These undefined values can be especially difficult to locate because evaluation of the actual parameter's value might not occur immediately after the call returns.
96.4.3 Applicable Vulnerability within ISO TR 24772-2

- 6.32 Passing parameters and return values [CSJ]

96.4.4 Noncompliant Code Example

```ada
for Value_T use
    (Alpha => 2#1#,
     Baker => 2#10#,
     Charlie => 2#100#,
     Dog => 2#1000#,
     Invalid => 2#1111#);

procedure Noncompliant (Register : Character;
                        Registera : out Value_T;
                        Registerb : out Value_T) is
begin
    if Register = 'A' then
        Registera := Alpha;
    end if;
end Noncompliant;
```

In the above example, some value is copied back for an output parameter as specified by Register. The other parameter is not assigned, and on return the value copied to the actual parameter may not be a valid representation for a value of the type. (We give the enumeration values a non-standard representation for the sake of illustration, i.e., to make it more likely that the undefined value is not valid.)

96.4.5 Compliant Code Example

```ada
procedure Compliant (Register : Character;
                     Registera : out Value_T;
                     Registerb : out Value_T) is
begin
    Registera := Invalid;
    Registerb := Invalid;
    if Register = 'A' then
        Registera := Alpha;
    end if;
end Compliant;
```

96.4.6 Notes

The GNATcheck rule specified above only detects a trivial case of an unassigned variable and doesn't provide a guarantee that there is no uninitialized access. It is not a replacement for a rigorous check for uninitialized access provided by advanced static analysis tools such as SPARK and CodePeer.

Note that the GNATcheck rule does not check function parameters (as of Ada 2012 functions can have `out` parameters). As a result, the better choice is either SPARK or CodePeer.
96.5 No Use of "others" in Exception Handlers (RPP05)

Level → Required

Category

- Safety ✓
- Cyber ✓

Goal

- Maintainability ✓
- Reliability ✓
- Portability ✓
- Performance
- Security

Remediation → Low

Verification Method → GNATcheck rule: OTHERS_In_Exception_Handlers

96.5.1 Reference

N/A

96.5.2 Description

Much like the situation with others in case statements and case expressions, the use of others in exception handlers makes it possible to omit an intended specific handler for an exception, especially a new exception added to an existing set of handlers. As a result, a subprogram could return normally without having applied any recovery for the specific exception occurrence, which is likely a coding error.

96.5.3 Applicable Vulnerability within ISO TR 24772-2

N/A

96.5.4 Noncompliant Code Example

```ada
procedure Noncompliant (X : in out Integer) is
begin
  X := X * X;
exception
  when others =>
    X := -1;
end Noncompliant;
```
96.5.5 Compliant Code Example

```ada
procedure Compliant (X : in out Integer) is
begin
    X := X * X;
exception
    when Constraint_Error =>
        X := -1;
end Compliant;
```

96.5.6 Notes

ISO TR 24772-2: 6.50.2 slightly contradicts this when applying exception handlers around calls to library routines:

- Put appropriate exception handlers in all routines that call library routines, including the catch-all exception handler `when others =>`
- Put appropriate exception handlers in all routines that are called by library routines, including the catch-all exception handler `when others =>`

ISO TR 24772-2 also recommends "All tasks should contain an exception handler at the outer level to prevent silent termination due to unhandled exceptions."

96.6 Avoid Function Side-Effects (RPP06)

**Level** → Advisory

**Category**

- **Safety** ✓
- **Cyber** ✓

**Goal**

- **Maintainability** ✓
- **Reliability** ✓
- **Portability** ✓
- **Performance**
- **Security**

**Remediation** → Medium

**Verification Method** → Code inspection
96.6.1 Reference

MISRA C Rule 13.2: “The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.”

96.6.2 Description

Functions cannot update an actual parameter or global variable.

A side effect occurs when evaluation of an expression updates an object. This rule applies to function calls, a specific form of expression.

Side effects enable one form of parameter aliasing (see below) and evaluation order dependencies. In general they are a potential point of confusion because the reader expects only a computation of a value.

There are useful idioms based on functions with side effects. Indeed, a random number generator expressed as a function must use side effects to update the seed value. So-called "memo" functions are another example, in which the function tracks the number of times it is called. Therefore, exceptions to this rule are anticipated but should only be allowed on a per-instance basis after careful analysis.

96.6.3 Applicable Vulnerability within ISO TR 24772-2

- 6.24 Side-effects and order of evaluation [SAM]

96.6.4 Noncompliant Code Example

```ada
Call_Count : Integer := 0;
function F return Boolean is
  Result : Boolean;
begin
  ... Call_Count := Call_Count + 1;
  return Result;
end F;
```

96.6.5 Compliant Code Example

Remove the update to Call_Count, or change the function into a procedure with a parameter for Call_Count.

96.6.6 Notes

Violations are detected by SPARK as part of a rule disallowing side effects on expression evaluation.
96.7 Functions Only Have Mode "in" (RPP07)

Level → Required

Category

Safety ✓
Cyber ✓

Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: function_out_parameters

96.7.1 Reference

N/A

96.7.2 Description

Functions must have only mode in.
As of Ada 2012, functions are allowed to have the same modes as procedures. However, this can lead to side effects and aliasing.
This rule disallows all modes except mode in for functions.

96.7.3 Applicable Vulnerability within ISO TR 24772-2

- 6.24 Side-effects and order of evaluation [SAM]

96.7.4 Noncompliant Code Example

```ada
function Noncompliant (Value : in out Integer) return Integer is
begin
  if Value < Integer'last then
    Value := Value + 1;
  end if;
  return Value;
end Noncompliant;
```
96.7.5 Compliant Code Example

```ada
function Compliant (Value : Integer) return Integer is
begin
    return Value + 1;
end Compliant;
```

OR

```ada
procedure Compliant (Value : in out Integer) is
begin
    if Value < Integer'last then
        Value := Value + 1;
    end if;
end Compliant;
```

96.7.6 Notes

Violations are detected by SPARK.

96.8 Limit Parameter Aliasing (RPP08)

Level → Required

Category

- Safety ✓
- Cyber ✓

Goal

- Maintainability ✓
- Reliability ✓
- Portability ✓
- Performance
- Security

Remediation → High

Verification Method → Code inspection
96.8.1 Reference


SPARK Reference Manual: Anti-Aliasing

96.8.2 Description

In software, an alias is a name which refers to the same object as another name. In some cases, it is an error in Ada to reference an object through a name while updating it through another name in the same subprogram. Most of these cases cannot be detected by a compiler. Even when not an error, the presence of aliasing makes it more difficult to understand the code for both humans and analysis tools, and thus it may lead to errors being introduced during maintenance.

This rule is meant to detect problematic cases of aliasing that are introduced through the actual parameters and between actual parameters and global variables in a subprogram call. It is a simplified version of the SPARK rule for anti-aliasing defined in SPARK Reference Manual, section 6.4.2: Anti-Aliasing.

A formal parameter is said to be immutable when the subprogram cannot modify its value or modify the value of an object by dereferencing a part of the parameter of access type (at any depth in the case of SPARK). In Ada and SPARK, this corresponds to either an anonymous access-to-constant parameter or a parameter of mode in and not of an access type. Otherwise, the formal parameter is said to be mutable.

A procedure call shall not pass two actual parameters which potentially introduce aliasing via parameter passing unless either:

- both of the corresponding formal parameters are immutable; or
- at least one of the corresponding formal parameters is immutable and is of a by-copy type that is not an access type.

If an actual parameter in a procedure call and a global variable referenced by the called procedure potentially introduce aliasing via parameter passing, then:

- the corresponding formal parameter shall be immutable; and
- if the global variable is written in the subprogram, then the corresponding formal parameter shall be of a by-copy type that is not an access type.

Where one of the rules above prohibits the occurrence of an object or any of its subcomponents as an actual parameter, the following constructs are also prohibited in this context:

- A type conversion whose operand is a prohibited construct;
- A call to an instance of Unchecked_Conversion whose operand is a prohibited construct;
- A qualified expression whose operand is a prohibited construct;
- A prohibited construct enclosed in parentheses.

\[369 \text{ http://www.ada-auth.org/standards/12rm/html/RM-6-2.html}\]
\[370 \text{ https://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#anti-aliasing}\]
\[371 \text{ https://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#anti-aliasing}\]
96.8.3 Applicable Vulnerability within ISO TR 24772-2

- 6.32 Passing parameters and return values [CSJ]

96.8.4 Noncompliant Code Example

```ada
type R is record
    Data : Integer := 0;
end record;

procedure Detect_Aliasing (Val_1 : in out R;
                           Val_2 : in R)
    is
    begin
        null;
    end Detect_Aliasing;

Obj : R;
begin
    Detect_Aliasing (Obj, Obj);
```

96.8.5 Compliant Code Example

Do not pass Obj as the actual parameter to both formal parameters.

96.8.6 Notes

All violations are detected by SPARK. The GNAT compiler switch `-gnateA[1]` enables detection of some cases, but not all.

96.9 Use Precondition and Postcondition Contracts (RPP09)

Level → Advisory

Category

Safety ✓

Cyber ✓

Goal

Maintainability ✓

Reliability ✓

Portability ✓
96.9.1 Reference

Power of Ten rule 5: "The assertion density of the code should average to a minimum of two assertions per function."

96.9.2 Description

Subprograms should declare Pre and/or Post contracts. Developers should consider specifying the Global contract as well, when the default does not apply.

Subprogram contracts complete the Ada notion of a specification, enabling clients to know what the subprogram does without having to know how it is implemented.

Preconditions define those logical (Boolean) conditions required for the body to be able to provide the specified behavior. As such, they are obligations on the callers. These conditions are checked at run-time in Ada, prior to each call, and verified statically in SPARK.

Postconditions define those logical (Boolean) conditions that will hold after the call returns normally. As such, they express obligations on the implementer, i.e., the subprogram body. The implementation must be such that the postcondition holds, either at run-time for Ada, or statically in SPARK.

Not all subprograms will have both a precondition and a postcondition, some will have neither.

The Global contract specifies interactions with those objects not local to the corresponding subprogram body. As such, they help complete the specification because, otherwise, one would need to examine the body of the subprogram itself and all those it calls, directly or indirectly, to know whether any global objects were accessed.

96.9.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle or the contract model [BLP]

96.9.4 Noncompliant Code Example

```ada
type Stack is private;
procedure Push (This : in out Stack; Item : Element);
```
96.9.5 Compliant Code Example

```ada
type Stack is private;
procedure Push (This : in out Stack; Item : Element) with
  Pre => not Full (This),
  Post => not Empty (This)
    and Top_Element (This) = Item
    and Extent (This) = Extent (This)'Old + 1
    and Unchanged (This'Old, Within => This),
  Global => null;
```

96.9.6 Notes

This rule must be enforced by manual inspection.

Moreover, the program must be compiled with enabled assertions (GNAT -gnata switch) to ensure that the contracts are executed, or a sound static analysis tool such as CodePeer or SPARK toolset should be used to prove that the contracts are always true.

96.10 Do Not Re-Verify Preconditions in Subprogram Bodies (RPP10)

**Level** → Advisory

**Category**

- Safety ✓
- Cyber ✓

**Goal**

- Maintainability ✓
- Reliability ✓
- Portability ✓
- Performance
- Security

**Remediation** → Low

**Verification Method** → Static analysis tools
96.10.1 Reference

N/A

96.10.2 Description

Do not re-verify preconditions in the corresponding subprogram bodies. It is a waste of cycles and confuses the reader as well.

96.10.3 Applicable Vulnerability within ISO TR 24772-2

N/A

96.10.4 Noncompliant Code Example

```ada
type Stack is private;
procedure Push (This : in out Stack; Item : Element) with
  Pre => not Full (This),
  Post => ...
begin
  if Full (This) then -- redundant check
    raise Overflow;
  end if;
  This.Top := This.Top + 1;
  This.Values (This.Top) := Item;
end Push;
```

96.10.5 Compliant Code Example

```ada
type Stack is private;
procedure Push (This : in out Stack; Item : Element) with
  Pre => not Full (This),
  Post => ...
begin
  This.Top := This.Top + 1;
  This.Values (This.Top) := Item;
end Push;
```
96.10.6 Notes

This rule can be enforced by CodePeer or SPARK, via detection of dead code.

96.11 Always Use the Result of Function Calls (RPP11)

Level → Advisory
Category
   Safety ✓
   Cyber ✓
Goal
   Maintainability ✓
   Reliability ✓
   Portability ✓
   Performance
   Security
Remediation → Low
Verification Method → Compiler restrictions

96.11.1 Reference

MISRA C Rule 17.7: "The value returned by a function having non-void return type shall be used," and
Directive 4.7: "If a function returns error information, that error information shall be tested."

96.11.2 Description

In Ada and SPARK, it is not possible to ignore the object returned by a function call. The call must be treated as a value, otherwise the compiler will reject the call. For example, the value must be assigned to a variable, or passed as the actual parameter to a formal parameter of another call, and so on.

However, that does not mean that the value is actually used to compute some further results. Although almost certainly a programming error, one could call a function, assign the result to a variable (or constant), and then not use that variable further.

Note that functions will not have side-effects (due to RPP06) so it is only the returned value that is of interest here.
96.11.3 Applicable Vulnerability within ISO TR 24772-2

- 6.47 Inter-language calling [DJS]

96.11.4 Noncompliant Code Example

N/A

96.11.5 Compliant Code Example

N/A

96.11.6 Notes

The GNAT compiler warning switch -gnatwu (or the more general -gnatwa warnings switch) will cause the compiler to detect variables assigned but not read. CodePeer will detect these unused variables as well. SPARK goes further by checking that all computations contribute all the way to subprogram outputs.

96.12 No Recursion (RPP12)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance

Security

Remediation → Low

Verification Method → GNATcheck rule: Recursive_Subprograms
96.12.1 Reference

MISRA C Rule 17.2: “Functions shall not call themselves, either directly or indirectly.”

96.12.2 Description

No subprogram shall be invoked, directly or indirectly, as part of its own execution. In addition to making static analysis more complex, recursive calls make static stack usage analysis extremely difficult, requiring, for example, manual supply of call limits.

96.12.3 Applicable Vulnerability within ISO TR 24772-2

- 6.35 Recursion [GDL]

96.12.4 Noncompliant Code Example

```ada
function Noncompliant (N : Positive) return Positive is
begin
  if N = 1 then
    return 1;
  else
    return N * Noncompliant (N - 1); -- could overflow
  end if;
end Noncompliant;
```

96.12.5 Compliant Code Example

```ada
function Compliant (N : Positive) return Positive is
  Result : Positive := 1;
begin
  for K in 2 .. N loop
    Result := Result * K; -- could overflow
  end loop;
  return Result;
end Compliant;
```

96.12.6 Notes

The compiler will detect violations with the restriction No_Recursion in place. Note this is a dynamic check.

The GNATCheck rule specified above is a static check, subject to the limitations described in GNATCheck Reference Manual: Recursive Subprograms\(^{372}\).

\(^{372}\) https://docs.adacore.com/live/wave/lkqi/html/gnatcheck_rm/gnatcheck_rm/predefined_rules.html#recursive-subprograms
96.13 No Reuse of Standard Typemarks (RPP13)

**Level** → Advisory

**Category**
- Safety ✓
- Cyber ✓

**Goal**
- Maintainability ✓
- Reliability ✓
- Portability ✓
- Performance
- Security

**Remediation** → Low

**Verification Method** → GNATcheck rule: overrides_standard_name

---

96.13.1 Reference

N/A

96.13.2 Description

Do not reuse the names of standard Ada typemarks (e.g. `type Integer is range -1_000 .. 1_000;`)

When a developer uses an identifier that has the same name as a standard typemark, such as `Integer`, a subsequent maintainer might be unaware that this identifier does not actually refer to Standard. `Integer` and might unintentionally use the locally-scoped `Integer` rather than the original Standard. `Integer`. The locally-scoped `Integer` can have different attributes (and may not even be of the same base type).

96.13.3 Applicable Vulnerability within ISO TR 24772-2

N/A
96.13.4 Noncompliant Code Example

```ada
type Boolean is range 0 .. 1 with Size => 1;
type Character is ('A', 'E', 'I', 'O', 'U');
```

96.13.5 Compliant Code Example

```ada
type Boolean_T is range 0 .. 1 with Size => 1;
type Character_T is ('A', 'E', 'I', 'O', 'U');
```

96.13.6 Notes

N/A

96.14 Use Symbolic Constants for Literal Values (RPP14)

Level → Advisory

Category

Safety ✓

Cyber ✓

Goal

Maintainability ✓

Reliability ✓

Portability ✓

Performance

Security

Remediation → Low

Verification Method → GNATcheck rule: Numeric_Literals

96.14.1 Reference

N/A
96.14.2 Description

Extensive use of literals in a program can lead to two problems. First, the meaning of the literal is often obscured or unclear from the context. Second, changing a frequently used literal requires searching the entire program source for that literal and distinguishing the uses that must be modified from those that should remain unmodified.

Avoid these problems by declaring objects with meaningfully named constants, setting their values to the desired literals, and referencing the constants instead of the literals throughout the program. This approach clearly indicates the meaning or intended use of each literal. Furthermore, should the constant require modification, the change is limited to the declaration; searching the code is unnecessary.

Some literals can be replaced with attribute values. For example, when iterating over an array, it is better to use `Array_Object'First .. Array_Object'Last` than using `1 .. Array_Object'Length`.

96.14.3 Applicable Vulnerability within ISO TR 24772-2

N/A

96.14.4 Noncompliant Code Example

```ada
type Array_T is array (0 .. 31) of Boolean;
function Any_Set (X : Array_T) return Boolean is
  (for some Flag in 0 .. 31 => X (Flag));
```

96.14.5 Compliant Code Example

```ada
Number_Of_Bits : constant := 32;
type Array_T is array (0 .. Number_Of_Bits - 1) of Boolean;
function Any_Set (X : Array_T) return Boolean is
  (for some Flag in X'Range => X (Flag));
```

96.14.6 Notes

N/A
CHAPTER
NINETYSEVEN

EXCEPTION USAGE (EXU)

Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance ✓
Security ✓

Description
Have a plan for managing the use of Ada exceptions at the application level.

Rules
EXU01, EXU02, EXU03, EXU04

Exceptions in modern languages present the software architect with a dilemma. On one hand, exceptions can increase integrity by allowing components to signal specific errors in a manner that cannot be ignored, and, in general, allow residual errors to be caught. (Although there should be no unexpected errors in high integrity code, there may be some such errors due, for example, to unforeseeable events such as radiation-induced single-event upsets.) On the other hand, unmanaged use of exceptions increases verification expense and difficulty, especially flow analysis, perhaps to an untenable degree. In that case overall integrity is reduced or unwarranted.

In addition, programming languages may define some system-level errors in terms of language-defined exceptions. Such exceptions may be unavoidable, at least at the system level. For example, in Ada, stack overflow is signalled with the language-defined Storage_Error exception. Other system events, such as bus error, may also be mapped to language-defined or vendor-defined exceptions.

Complicating the issue further is the fact that, if exceptions are completely disallowed, there will be no exception handling code in the underlying run-time library. The effects are unpredictable if any exception actually does occur.

Therefore, for the application software the system software architect must decide whether to allow exceptions at all, and if they are to be used, decide the degree and manner of their usage. At the system level, the architect must identify the exceptions that are possible and how they will be addressed.
97.1 Do Not Raise Language-Defined Exceptions (EXU01)

Level → Required

Category
  Safety ✓
  Cyber ✓

Goal
  Maintainability ✓
  Reliability ✓
  Portability ✓
  Performance
  Security

Remediation → Low

Verification Method → GNATcheck rule: Raising_Predefined_Exceptions

97.1.1 Reference

[SEI-Java] ERR07-

97.1.2 Description

In no case should the application explicitly raise a language-defined exception.

The Ada language-defined exceptions are raised implicitly in specific circumstances defined by the language standard. Explicitly raising these exceptions would be confusing to application developers. The potential for confusion increases as the exception is propagated up the dynamic call chain, away from the point of the raise statement, because this increases the number of paths and thus corresponding language-defined checks that could have been the cause.

97.1.3 Applicable Vulnerability within ISO TR 24772-2

N/A
97.1.4 Noncompliant Code Example

```ada
procedure Noncompliant (X : in out Integer) is
begin
  if X < Integer'Last / 2 then
    X := X * 2;
  else
    raise Constraint_Error;
  end if;
end Noncompliant;
```

97.1.5 Compliant Code Example

```ada
procedure Compliant (X : in out Integer) is
begin
  if X < Integer'Last / 2 then
    X := X * 2;
  else
    raise Math_Overflow;
  end if;
end Compliant;
```

97.1.6 Notes

N/A

97.2 No Unhandled Application-Defined Exceptions (EXU02)

**Level** → Required

**Category**
- Safety ✓
- Cyber ✓

**Goal**
- Maintainability ✓
- Reliability ✓
- Portability ✓
- Performance
- Security

**Remediation** → Low

**Verification Method** → GNATcheck rule: Unhandled_Exceptions
97.2.1 Reference

N/A

97.2.2 Description

All application-defined exceptions must have at least one corresponding handler that is applicable. Otherwise, if an exception is raised, undesirable behavior is possible. The term applicable means that there is no dynamic call chain that can reach the active exception which does not also include a handler that will be invoked for that exception, somewhere in that chain.

When an unhandled exception occurs in the sequence of statements of an application task and propagates to task's body, the task terminates abnormally. No notification of some sort is required or defined by the language, although some vendors' implementations may print out a log message or provide some other non-standard response. (Note that such a notification implies an external persistent environment, such as an operating system, that may not be present in all platforms.) The task failure does not affect any other tasks unless those other tasks attempt to communicate with it. In short, failure is silent.

Although the language-defined package Ada.Task_Termination can be used to provide a response using standard facilities, not all run-time libraries provide that package. For example, under the Ravenscar profile, application tasks are not intended to terminate, neither normally nor abnormally, and the language does not define what happens if they do. A run-time library for a memory-constrained target, especially a bare-metal target without an operating system, might not include any support for task termination when the tasking model is Ravenscar. The effects of task termination in that case are not defined by the language.

When an unhandled exception occurrence reaches the main subprogram and is not handled there, the exception occurrence is propagated to the environment task, which then completes abnormally. Even if the main subprogram does handle the exception, the environment task still completes (normally in that case).

When the environment task completes (normally or abnormally) it waits for the completion of dependent application tasks, if any. Those dependent tasks continue executing normally, i.e., they do not complete as a result of the environment task completion. Alternatively, however, instead of waiting for them, the implementation has permission to abort the dependent application tasks, per Ada Reference Manual: 10.2 (30) Program Execution The resulting application-specific effect is undefined.

Finally, whether the environment task waited for the dependent tasks or aborted them, the semantics of further execution beyond that point are undefined. There is no concept of a calling environment beyond the environment task (Ada Reference Manual: 10.2 (30) Program Execution). In some systems there is no calling environment, such as bare-metal platforms with only an Ada run-time library and no operating system.

97.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.36 Ignored error status and unhandled exceptions [OYB]

97.2.4 Noncompliant Code Example

```ada
procedure Main is
begin
  if Argument_Count = 0 then
    raise Cli_Exception;
  else
    begin
      Start_Application (Argument (1));
    exception
      when Application_Exception =>
        Put_Line ("Application failed");
    end if;
  end if;
end Main;
```

97.2.5 Compliant Code Example

```ada
procedure Main is
begin
  if Argument_Count = 0 then
    raise Cli_Exception;
  else
    begin
      Start_Application (Argument (1));
    exception
      when Application_Exception =>
        Put_Line ("Application failed");
    end if;
  exception
    when Cli_Exception =>
      Put_Line ("Failure");
end Main;
```

97.2.6 Notes

SPARK can prove that no exception will be raised (or fail to prove it and indicate the failure).

97.3 No Exception Propagation Beyond Name Visibility (EXU03)

**Level** → Required

**Category**

- **Safety**
  

---

97.3. No Exception Propagation Beyond Name Visibility (EXU03) 1771
Learning Ada

Cyber ✓

Goal
Maintainability ✓
Reliability ✓
Portability ✓
Performance ✓
Security

Remediation → Low

Verification Method → GNATcheck rule: Non_Visible_Exceptions

97.3.1 Reference
RPP05

97.3.2 Description
An active exception can be propagated dynamically past the point where the name of the exception is visible (the scope of the declaration). The exception can only be handled via others past that point. That situation prevents handling the exception specifically, and violates RPP05.

97.3.3 Applicable Vulnerability within ISO TR 24772-2
N/A

97.3.4 Noncompliant Code Example

```ada
procedure Noncompliant (Param : in out Integer) is
  Noncompliant_Exception : exception;
begins
  Param := Param * Param;
except
    when others =>
      raise Noncompliant_Exception;
  end Noncompliant;
```

As a result the exception name cannot be referenced outside the body:

```ada
procedure Bad_Call (Param : in out Integer) is
begin
  Noncompliant (Param);
except
    when Noncompliant_Exception => -- compile error
      null;
  end Bad_Call;
```
97.3.5 Compliant Code Example

```ada
Compliant_Exception : exception;
procedure Compliant (Param : in out Integer) is
  begin
    Param := Param * Param;
  exception
    when others =>
      raise Compliant_Exception;
end Compliant;

procedure Good_Call (Param : in out Integer) is
  begin
    Compliant (Param);
  exception
    when Compliant_Exception =>
      null;
end Good_Call;
```

97.3.6 Notes

N/A

97.4 Prove Absence of Run-time Exceptions (EXU04)

Level → Required

Category

Safety ✓

Cyber ✓

Goal

Maintainability ✓

Reliability ✓

Portability ✓

Performance

Security

Remediation → Low

Verification Method → Compiler restrictions
97.4.1 Reference

MISRA C Rule 1.3: “There shall be no occurrence of undefined or critical unspecified behaviour.”

97.4.2 Description

In many high-integrity systems the possible responses to an exception are limited or non-existent. In these cases the only approach is to prove exceptions cannot occur in the first place. Additionally, the cost of proving exceptions cannot happen may be less than the cost of analyzing code in which they are allowed to be raised.

The restriction No_Exceptions can be used with pragma Restrictions to enforce this approach. Specifically, the restriction ensures that raise statements and exception handlers do not appear in the source code and that language-defined checks are not emitted by the compiler. However, a run-time check performed automatically by the hardware is permitted because it typically cannot be prevented. An example of such a check would be traps on invalid addresses. If a hardware check fails, or if an omitted language-defined check would have failed, execution is unpredictable. As a result, enforcement with the restriction is not ideal. However, proof of the absence of run-time errors is possible using the SPARK subset of Ada.

97.4.3 Applicable Vulnerability within ISO TR 24772-2

N/A

97.4.4 Noncompliant Code Example

N/A

97.4.5 Compliant Code Example

N/A

97.4.6 Notes

This restriction is detected by SPARK, in which any statements explicitly raising an exception must be proven unreachable (or proof fails and the failure is indicated), and any possibility of run-time exception should be proved not to happen.
CHAPTER
NINETY-EIGHT

OBJECT-ORIENTED PROGRAMMING (OOP)

Goal

Maintainability ✓

Reliability ✓

Portability

Performance

Security ✓

Description
Have a plan for selecting the OOP facilities of the language to use.

Rules
OOP01, OOP02, OOP03, OOP04, OOP05, OOP06, OOP07

There are many issues to consider when planning the use of Object Oriented features in a high-integrity application. Choices should be made based on the desired expressive power of the OO features and the required level of certification or safety case.

For example, the use of inheritance can provide abstraction and separation of concerns. However, the extensive use of inheritance, particularly in deep hierarchies, can lead to fragile code bases.

Similarly, when new types of entities are added, dynamic dispatching provides separation of the code that must change from the code that manipulates those types and need not be changed to handle new types. However, analysis of dynamic dispatching must consider every candidate object type and analyze the associated subprogram for appropriate behavior.

Therefore, the system architect has available a range of possibilities for the use of OOP constructs, with tool enforcement available for the selections. Note that full use of OOP, including dynamic dispatching, may not be unreasonable.

The following rules assume use of tagged types, a requirement for full OOP in Ada.
98.1 No Class-wide Constructs Policy (OOP01)

Level → Advisory
Category
  Safety ✓
  Cyber ✓
Goal
  Maintainability ✓
  Reliability ✓
  Portability
  Performance
  Security ✓
Remediation → N/A
Verification Method → Compiler restrictions
Mutually Exclusive → OOP02

98.1.1 Reference

N/A

98.1.2 Description

In this approach, tagged types are allowed and type extension (inheritance) is allowed, but there are no class-wide constructs.

This restriction ensures there are no class-wide objects or formal parameters, nor access types designating class-wide types.

In this approach there are no possible dynamic dispatching calls because such calls can only occur when a class-wide value is passed as the parameter to a primitive operation of a tagged type.

98.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.43 Redispatching [PPH]
98.1.4 Noncompliant Code Example

```
X : Object'Class := Some_Object;
```

98.1.5 Compliant Code Example

```
X : Object := Some_Object;
```

98.1.6 Notes

The compiler will detect violations with the standard Ada restriction No_Dispatch applied.

98.2 Static Dispatching Only Policy (OOP02)

**Level** → Advisory

**Category**

- Safety ✓
- Cyber ✓

**Goal**

- Maintainability ✓
- Reliability ✓
- Portability
- Performance
- Security ✓

**Remediation** → N/A

**Verification Method** → Compiler restrictions

**Mutually Exclusive** → OOP01

98.2.1 Reference

N/A
98.2.2 Description

In this approach, class-wide constructs are allowed, as well as tagged types and type extension (inheritance), but dynamic dispatching remains disallowed (i.e., as in OOP01).

This rule ensures there are no class-wide values passed as the parameter to a primitive operation of a tagged type, hence there are no dynamically dispatched calls.

Note that this rule should not be applied without due consideration.

98.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.43 Redispatching [PPH]

98.2.4 Noncompliant Code Example

```ada
Some_Primitive (Object'Class (X));
```

98.2.5 Compliant Code Example

```ada
Some_Primitive (X);
```

98.2.6 Notes

N/A

98.3 Limit Inheritance Hierarchy Depth (OOP03)

**Level** → Advisory

**Category**

- Safety ✓
- Cyber ✓

**Goal**

- Maintainability ✓
- Reliability ✓
- Portability
- Performance
- Security ✓

**Remediation** → High

**Verification Method** → GNATcheck rule: Deep_Inheritance_Hierarchies:2
98.3.1 Reference

[AdaOOP2016] section 5.1

98.3.2 Description

A class inheritance hierarchy consists of a set of types related by inheritance. Each class, other than the root class, is a subclass of other classes, and each, except for "leaf" nodes, is a base class for those that are derived from it.

Improperly designed inheritance hierarchies complicate system maintenance and increase the effort in safety certification, in any programming language.

A common characteristic of problematic hierarchies is "excessive" depth, in which a given class is a subclass of many other classes. Depth can be a problem because a change to a class likely requires inspection, modification, recompilation, and retesting/reverification of all classes below it in the hierarchy. The extent of that effect increases as we approach the root class. This rippling effect is known as the fragile base class problem. Clearly, the greater the depth the more subclasses there are to be potentially affected. In addition, note that a change to one class may cause a cascade of other secondary changes to subclasses, so the effect is often not limited to a single change made to all the subclasses in question.

Deep inheritance hierarchies also contribute to complexity, rather than lessening it, by requiring the reader to understand multiple superclasses in order to understand the behavior of a given subclass.

98.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.41 Inheritance [RIP]

98.3.4 Noncompliant Code Example

The threshold for "too deep" is inexact, but beyond around 4 or 5 levels the complexity accelerates rapidly.

```ada
type Shape_T is tagged private;
procedure Set_Name (Shape : Shape_T; Name : String);
function Get_Name (Shape : Shape_T) return String;

type Quadrilateral_T is new Shape_T with private;
type Trapezoid_T is new Quadrilateral_T with private;
type Parallelogram_T is new Trapezoid_T with private;
type Rectangle_T is new Parallelogram_T with private;
type Square_T is new Rectangle_T with private;
```
98.3.5 Compliant Code Example

```ada
type Shape_T is tagged private;
procedure Set_Name (Shape : Shape_T; Name : String);
function Get_Name (Shape : Shape_T) return String;

type Quadrilateral_T is new Shape_T with private;
type Rectangle_T is new Quadrilateral_T with private;
type Square_T is new Rectangle_T with private;
```

98.3.6 Notes

N/A

98.4 Limit Statically-Dispatched Calls to Primitive Operations (OOP04)

**Level** → Advisory

**Category**

- Safety ✓
- Cyber ✓

**Goal**

- Maintainability ✓
- Reliability ✓
- Portability ✓
- Performance
- Security

**Remediation** → Medium (easy fix, but a difficult to detect bug)

**Verification Method** → GNATcheck rule: Direct_Calls_To_Primitives

98.4.1 Reference

N/A
98.4.2 Description

This rule applies only to tagged types, when visibly tagged at the point of a call from one primitive to another of that same type.

By default, subprogram calls are statically dispatched. Dynamic dispatching only occurs when a class-wide value is passed to a primitive operation of a specific type. Forcing an otherwise optional dynamic dispatching call in this case is known as redispatching.

When one primitive operation of a given tagged type invokes another distinct primitive operation of that same type, use redispatching so that an overriding version of that other primitive will be invoked if it exists. Otherwise an existing overridden version would not be invoked, which is very likely an error.

This rule does not apply to the common case in which an overriding of a primitive operation calls the "parent" type's version of the overridden operation. Such calls occur in the overridden body when the new version is not replacing, but rather, is augmenting the parent type's version. In this case the new version must do whatever the parent version did, and can then add functionality specific to the new type.

By default, this rule applies to another common case in which static calls from one primitive operation to another make sense. Specifically, constructors are often implemented in Ada as functions that create a new value of the tagged type. As constructors, these functions are type-specific. They must call the primitive operations of the type being created, not operations that may be overridden for some type later derived from it. (Note that there is a GNATcheck rule parameter to not flag this case.)

Typically constructor functions only have the tagged type as the result type, not as the type for formal parameters, if any, because actual parameters of the tagged type would themselves likely require construction. This specific usage is the case ignored by the GNATcheck rule parameter.

Note that constructors implemented as procedures also call primitive operations of the specific type, for the same reasons as constructor functions. This usage is allowed by this rule and does not require the GNATcheck parameter. (The difference between function and procedure constructors is that these procedures will have a formal parameter of the tagged type, of mode out.)

98.4.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle of the contract model [BLP]
- 6.43 Redispatching [PPH]
- 6.44 Polymorphic variables [BKK]

98.4.4 Noncompliant Code Example

Class constructs

```ada
package Root is
  type Root_T is tagged null record;
  procedure Noncompliant (X : in out Root_T) is null;
  procedure Compliant (X : in out Root_T) is null;
  procedure Other_Prim (X : in out Root_T) is null;
end Root;

package Child is
  use Root;
```

(continues on next page)
type Child_T is new Root_T with null record;
procedure Noncompliant (X : in out Child_T);
procedure Compliant (X : in out Child_T);
procedure Other_Prim (X : in out Child_T);
end Child;

procedure Not_A_Primitive (X : in out Child.Child_T) is null;

Noncompliant Code

procedure Noncompliant (X : in out Child_T) is
begin
  Other_Prim (Root_T (X));
  Other_Prim (X);
end Noncompliant;

98.4.5 Compliant Code Example

procedure Compliant (X : in out Child_T) is
begin
  Compliant (Root_T (X)); -- constructor style is OK
  Not_A_Primitive (X);
end Compliant;

98.4.6 Notes

N/A

98.5 Use Explicit Overriding Annotations (OOP05)

Level → Required
Category

Safety  ✓
Cyber   ✓

Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance
Security

Remediation → Low
Verification Method → GNATcheck rule: Style_Checks:O
98.5.1 Reference

[AdaOOP2016] section 4.3

98.5.2 Description

The declaration of a primitive operation that overrides an inherited operation must include an explicit **overriding** annotation.

The semantics of inheritance in mainstream object-oriented languages may result in two kinds of programming errors: 1) intending, but failing, to override an inherited subprogram, and 2) intending not to override an inherited subprogram, but doing so anyway. Because an overridden subprogram may perform subclass-specific safety or security checks, the invocation of the parent subprogram on a subclass instance can introduce a vulnerability.

The first issue (intending but failing to override) typically occurs when the subprogram name is misspelled. In this case a new or overloaded subprogram is actually declared.

The second issue (unintended overriding) can arise when a new subprogram is added to a parent type in an existing inheritance hierarchy. The new subprogram happens to cause one or more inherited subprograms below it to override the new superclass version. This mistake typically happens during program maintenance.

In Ada, much like other modern languages, one can annotate a subprogram declaration (and body) with an indication that the subprogram is an overriding of an inherited version. This is done with the **overriding** reserved word preceding the subprogram specification.

Similarly, in Ada one can explicitly indicate that a subprogram is not an overriding. To do so, the programmer includes the reserved words not **overriding** immediately prior to the subprogram specification.

Of course, incorrect marking errors are flagged by the compiler. If a subprogram is explicitly marked as overriding but is not actually overriding, the compiler will reject the code. Likewise, if a primitive subprogram is explicitly marked as not overriding, but actually is overriding, the compiler will reject the code.

However, most subprograms are not overriding so it would be a heavy burden on the programmer to make them explicitly indicate that fact. That’s not to mention the relatively heavy syntax required.

In addition, both annotations are optional for the sake of compatibility with prior versions of the language. Therefore, a subprogram without either annotation might or might not be overriding. A legal program could contain some explicitly annotated subprograms and some that are not annotated at all. But because the compiler will reject explicit annotations that are incorrect, all we require is that one of the two cases be explicitly indicated for all such subprograms. Any unannotated subprograms not flagged as errors are then necessarily not that case, they must be the other one.

Since overriding is less common and involves slightly less syntax to annotate, the guideline requires explicit annotations only for overriding subprograms. It follows that any subprograms not flagged as errors by the compiler are not overriding, so they need not be marked explicitly as such.

This guideline is implemented by compiler switches, or alternatively, by a GNATcheck rule (specified below the table). With this guideline applied and enforced, the two inheritance errors described in the introduction cannot happen.

Note that the compiler switches will also require the explicit overriding indicator when overriding a language-defined operator. The switches also apply to inherited primitive subprograms for non-tagged types.

98.5. Use Explicit Overriding Annotations (OOP05)
98.5.3 Applicable Vulnerability within ISO TR 24772-2

- 6.34 Subprogram signature mismatch [OTR]
- 6.41 Inheritance [RIP]

98.5.4 Noncompliant Code Example

```ada
type Root_T is tagged null record;
procedure Primitive (X : in out Root_T) is null;

type Noncompliant_Child_T is new Root_T with null record;
procedure Primitive (X : in out Noncompliant_Child_T) is null;
```

98.5.5 Compliant Code Example

```ada
type Compliant_Child_T is new Root_T with null record;
overriding procedure Primitive (X : in out Compliant_Child_T) is null;
```

98.5.6 Notes

This rule requires the GNAT compiler switches -gnatyO and -gnatwe in order for the compiler to flag missing overriding annotations as errors. The first causes the compiler to generate the warnings, and the second causes those warnings to be treated as errors.

98.6 Use Class-wide Pre/Post Contracts (OOP06)

Level → Required
Category
- Safety ✓
- Cyber ✓
Goal
- Maintainability ✓
- Reliability ✓
- Portability
- Performance
- Security ✓
Remediation → Low
Verification Method → GNATcheck rule: Specific_Pre_Post
98.6.1 Reference

[AdaOOP2016] section 6.1.4
SPARK User's Guide, section 7.5.2

98.6.2 Description

For primitive operations of tagged types, use only class-wide pre/post contracts, if any.

The class-wide form of precondition and postcondition expresses conditions that are intended to apply to any version of the subprogram. Therefore, when a subprogram is derived as part of inheritance, only the class-wide form of those contracts is inherited from the parent subprogram, if any are defined. As a result, it only makes sense to use the class-wide form in this situation.

(The same semantics and recommendation applies to type invariants.)

Note: this approach will be required for OOP07 (Ensure Local Type Consistency).

98.6.3 Applicable Vulnerability within ISO TR 24772-2

• 6.42 Violations of the Liskov substitution principle or the contract model [BLP]

98.6.4 Noncompliant Code Example

```ada
type Root_T is tagged null record;
procedure Set_Name (X : Root_T;
                   Name : String)
   with Pre => Name'length > 0;
function Get_Name (X : Root_T) return String
   with Post => Get_Name'result'length > 0;
```

98.6.5 Compliant Code Example

```ada
type Root_T is tagged null record;
procedure Set_Name (X : Root_T;
                   Name : String)
   with Pre'class => Name'length > 0;
function Get_Name (X : Root_T) return String
   with Post'class => Get_Name'result'length > 0;
```

98.6.6 Notes

N/A

98.7 Ensure Local Type Consistency (OOP07)

Level → Required
Category
  Safety ✓
  Cyber ✓
Goal
  Maintainability ✓
  Reliability ✓
  Portability
  Performance
  Security ✓
Remediation → N/A
Verification Method → Software test

98.7.1 Reference

GNAT User’s Guide, section 5.10.11

98.7.2 Description

Either:
  • Formally verify local type consistency, or
  • Ensure that each tagged type passes all the tests of all the parent types which it can replace.

Rationale:

One of the fundamental benefits of OOP is the ability to manipulate objects in a class inheritance hierarchy without “knowing” at compile-time the specific classes of the objects being manipulated. By manipulate we mean invoking the primitive operations, the methods defined by the classes.

We will use the words class and type interchangeably, because classes are composed in Ada and SPARK using a combination of building blocks, especially type declarations. However, we will use the term subclass rather than subtype because the latter has a specific meaning in Ada and SPARK that is unrelated to OOP.

https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_utility_programs.html
The objects whose operations are being invoked can be of types anywhere in the inheritance tree, from the root down to the bottom. The location, i.e., the specific type, is transparent to the manipulating code. This type transparency is possible because the invoked operations are dynamically dispatched at run-time, rather than statically dispatched at compile-time.

Typically, the code manipulating the objects does so in terms of superclasses closer to the root of the inheritance tree. Doing so increases generality because it increases the number of potential subclasses that can be manipulated. The actual superclass chosen will depend on the operations required by the manipulation. In Ada and SPARK, subclasses can add operations but can never remove them, so more operations are found as we move down from the root. That is the nature of specialization. Note that the guarantee of an invoked operations’ existence is essential for languages used in this domain.

However, for this transparent manipulation to be functionally correct — to accomplish what the caller intends — the primitive operations of subclasses must be functionally indistinguishable from those of the superclasses. That doesn't mean the subclasses cannot make changes. Indeed, the entire point of subclasses is to make changes. In particular, functional changes can be either introduction of new operations, or overridings of inherited operations. It is these overridings that must be functionally transparent to the manipulating code. (Of course, for an inherited operation that is not overridden, the functionality is inherited as-is, and is thus transparent trivially.)

The concept of functional transparency was introduced, albeit with different terminology, by Liskov and Wing in 1994 [LiskovWing1994] and is, therefore, known as the Liskov Substitution Principle, or LSP. The substitution in LSP means that a subclass must be substitutable for its superclass, i.e., a subclass instance should be usable whenever a superclass instance is required.

Unfortunately, requirements-based testing will not detect violations of LSP because unit-level requirements do not concern themselves with superclass substitutability.

However, the OO supplement of DO-178C [DO178C] offers solutions, two of which are in fact the options recommended by this guideline.

Specifically, the supplement suggests adding a specific verification activity it defines as Local Type Consistency Verification. This activity ensures LSP is respected and, in so doing, addresses the vulnerability.

Verification can be accomplished statically with formal methods in SPARK, or dynamically via a modified form of testing.

For the static approach, type consistency is verified by examining the properties of the overriding operation's preconditions and postconditions. These are the properties required by Design by Contract, as codified by Bertrand Meyer [Meyer1997]. Specifically, an overridden primitive may only replace the precondition with one weaker than that of the parent version, and may only replace the postcondition with one stronger. In essence, relative to the parent version, an overridden operation can only require the same or less, and provide the same or more. Satisfying that requirement is sufficient to ensure functional transparency because the manipulating code only "knows" the contracts of the class it manipulates, i.e., the view presented by the type, which may very well be a superclass of the one actually invoked.

In Ada and SPARK, the class-wide form of preconditions and postconditions are inherited by overridden primitive operations of tagged types. The inherited precondition and that of the overriding (if any) are combined into a conjunction. All must hold, otherwise the precondition fails. Likewise, the inherited postcondition is combined with the overriding postcondition into a disjunction. At least one of them must hold. In Ada these are tested at run-time. In SPARK, they are verified statically (or not, in which case proof fails and an error is indicated).

To verify substitutability via testing, all the tests for all superclass types are applied to objects of the given subclass type. If all the parent tests pass, this provides a high degree
of confidence that objects of the new tagged type can properly substitute for parent type objects. Note that static proof of consistency provides an even higher degree of confidence.

98.7.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle of the contract model [BLP]
- 6.43 Redispetching [PPH]
- 6.44 Polymorphic variables [BKK]

98.7.4 Noncompliant Code Example

```ada
package P is
  pragma Elaborate_Body;
  type Rectangle is tagged private;
  procedure Set_Width (This : in out Rectangle;
      Value : Positive)
  with
      Post => Width (This) = Value and
      Height (This) = Height (This'Old);
  function Width (This : Rectangle) return Positive;
  procedure Set_Height (This : in out Rectangle;
      Value : Positive)
  with
      Post => Height (This) = Value and
      Width (This) = Width (This'Old);
  function Height (This : Rectangle) return Positive;
private
  ...
end P;
```

The postcondition for Set_Width states that the Height is not changed. Likewise, for Set_Height, the postcondition asserts that the Width is not changed. However, these postconditions are not class-wide so they are not inherited by subclasses.

Now, in a subclass Square, the operations are overridden so that setting the width also sets the height to the same value, and vice versa. Thus the overridden operations do not maintain type consistency, but this fact is neither detected at run-time, nor could SPARK verify it statically (and SPARK is not used at all in these versions of the packages).

```ada
with P; use P;
package Q is
  pragma Elaborate_Body;
  type Square is new Rectangle with private;

  overriding
  procedure Set_Width (This : in out Square;
      Value : Positive)
  with
      Post => Width (This) = Height (This);

  overriding
  procedure Set_Height (This : in out Square;
      Value : Positive)
```

(continues on next page)
98.7.5 Compliant Code Example

```ada
package P with SPARK_Mode is
    pragma Elaborate_Body;
    type Rectangle is tagged private;

    procedure Set_Width (This : in out Rectangle;
        Value : Positive)
    with
        Post'Class => Width (This) = Value and
                Height (This) = Height (This'Old);

    function Width (This : Rectangle) return Positive;

    procedure Set_Height (This : in out Rectangle;
        Value : Positive)
    with
        Post'Class => Height (This) = Value and
                Width (This) = Width (This'Old);

    function Height (This : Rectangle) return Positive;

    private
        ...
    end P;
```

Now the postconditions are class-wide so they are inherited by subclasses. In the subclass Square, the postconditions will not hold at run-time. Likewise, SPARK can now prove that type consistency is not verified because the postconditions are weaker than those inherited:

```ada
with P; use P;
package Q with SPARK_Mode is
    pragma Elaborate_Body;
    type Square is new Rectangle with private;

    overriding procedure Set_Width (This : in out Square;
        Value : Positive)
    with
        Post'Class => Width (This) = Height (This);

    overriding procedure Set_Height (This : in out Square;
        Value : Positive)
    with
        Post'Class => Width (This) = Height (This);

    private
        type Square is new Rectangle with null record;
    end Q;
```
98.7.6 Notes

Verification can be achieved dynamically with the GNATtest tool, using the --validate-type-extensions switch. SPARK enforces this rule.
Goal

Maintainability ✓
Reliability ✓
Portability ✓
Performance ✓
Security ✓

Description
These rules promote "best practices" for software development.

Rules
SWE01, SWE02, SWE03, SWE04

99.1 Use SPARK Extensively (SWE01)

Level → Advisory
Category
Safety ✓
Cyber ✓

Goal
Maintainability ✓
Reliability ✓
Portability ✓
Performance ✓
Security ✓
Learning Ada

Remediation → High, as retrofit can be extensive
Verification Method → Compiler restrictions

99.1.1 Reference


99.1.2 Description

SPARK has proven itself highly effective, both in terms of low defects, low development costs, and high productivity. The guideline advises extensive use of SPARK, especially for the sake of formally proving the most critical parts of the source code. The rest of the code can be in SPARK as well, even if formal proof is not intended, with some parts in Ada when features outside the SPARK subset are essential.

99.1.3 Applicable Vulnerability within ISO TR 24772-2

N/A

99.1.4 Noncompliant Code Example

Any code outside the (very large) SPARK subset is flagged by the compiler.

99.1.5 Compliant Code Example

N/A

99.1.6 Notes

Violations are detected by the SPARK toolset.

99.2 Enable Optional Warnings and Treat As Errors (SWE02)

Level → Required
Category
  Safety ✓
  Cyber ✓
Goal
  Maintainability ✓

99.2.1 Reference

Power of 10 rule #10: "All code must be compiled, from the first day of development, with all compiler warnings enabled at the most pedantic setting available. All code must compile without warnings."

99.2.2 Description

The Ada compiler does a degree of static analysis itself, and generates many warnings when they are enabled. These warnings likely indicate very real problems so they should be examined and addressed, either by changing the code or disabling the warning for the specific occurrence flagged in the source code.

To ensure that warnings are examined and addressed one way or the other, the compiler must be configured to treat warnings as errors, i.e., preventing object code generation.

Note that warnings will occasionally be given for code usage that is intentional. In those cases the warnings should be disabled by using `pragma Warnings` with the parameter Off, and a string indicating the error message to be disabled. In other cases, a different mechanism might be appropriate, such as aspect (or pragma) Unreferenced.

99.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.18 Dead Store [WXQ]
- 6.19 Unused variable [YZS]
- 6.20 Identifier name reuse [YOW]
- 6.22 Initialization of variables [LAV]

99.2.4 Noncompliant Code Example

```ada
procedure P (This : Obj) is
begin
    ... code not referencing This
end P;
```

The formal parameter controls dispatching for the sake of selecting the subprogram to be called but does not participate in the implementation of the body.
99.2.5 Compliant Code Example

```ada
procedure P (This : Obj) is
  pragma Unreferenced (This);
begin
  ... code not referencing This
end P;
```

The compiler will no longer issue a warning that the formal parameter This is not referenced. Of course, if that changes and This becomes referenced, the compiler will flag the `pragma`.

99.2.6 Notes

This rule can be applied via the GNAT -gnatwae compiler switch, which both enables warnings and treats them as errors. Note that the switch enables almost all optional warnings, but not all. Some optional warnings correspond to very specific circumstances, and would otherwise generate too much noise for their value.

99.3 Use a Static Analysis Tool Extensively (SWE03)

**Level** → Mandatory

**Category**

- **Safety**
  - ✓
- **Cyber**
  - ✓

**Goal**

- **Maintainability**
  - ✓
- **Reliability**
  - ✓
- **Portability**
  - ✓
- **Performance**
  - ✓
- **Security**
  - ✓

**Remediation** → High

**Verification Method** → Static analysis tools
99.3.1 Reference

Power of 10 rule #10: "All code must also be checked daily with at least one, but preferably more than one, strong static source code analyzer and should pass all analyses with zero warnings."

99.3.2 Description

If not using SPARK for regular development, use a static analyzer, such as CodePeer, extensively. No warnings or errors should remain unresolved at the given level adopted for analysis (which can be selected to adjust the false positive ratio).

Specifically, any code checked into the configuration management system must be checked by the analyzer and be error-free prior to check-in. Similarly, each nightly build should produce a CodePeer baseline for the project.

99.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.6 Conversion errors [FLC]
- 6.18 Dead store [WXQ]
- 6.19 Unused variable [YZS]
- 6.20 Identifier name reuse [YOW]
- 6.24 Side-effects and order of evaluation [SAM]
- 6.25 Likely incorrect expression [KOA]

99.3.4 Noncompliant Code Example

N/A

99.3.5 Compliant Code Example

N/A

99.3.6 Notes

CodePeer is the recommended static analyzer. Note that CodePeer can detect GNATcheck rule violations (via the -gnatcheck CodePeer switch and a rules file).
99.4 Hide Implementation Artifacts (SWE04)

Level → Advisory

Category

Safety ✓

Cyber ✓

Goal

Maintainability ✓

Reliability ✓

Portability

Performance

Security ✓

Remediation → High, as retrofit can be extensive

Verification Method → GNATcheck rule: Visible_Components

99.4.1 Reference

MISRA C Rule 8.7: "Functions and objects should not be defined with external linkage if they are referenced in only one translation unit."

99.4.2 Description

Do not make implementation artifacts compile-time visible to clients. Only make available those declarations that define the abstraction presented to clients by the component. In other words, define Abstract Data Types and use the language to enforce the abstraction. This is a fundamental Object-Oriented Design principle.

This guideline minimizes client dependencies and thus allows the maximum flexibility for changes in the underlying implementation. It minimizes the editing changes required for client code when implementation changes are made.

This guideline also limits the region of code required to find any bugs to the package and child packages, if any, defining the abstraction.

This guideline is to be followed extensively as the design default for components. Once the application code size becomes non-trivial, the cost of retrofit is extremely high.
99.4.3 Applicable Vulnerability within ISO TR 24772-2

N/A

99.4.4 Noncompliant Code Example

```ada
package Noncompliant is
    type Content_T is array (Capacity_T range <>) of Integer;
    type Stack_T (Capacity : Capacity_T) is tagged record
        Content : Content_T (1 .. Capacity);
        Top : Capacity_T := 0;
    end record;
    procedure Push (Stack : in out Stack_T; Item : Integer);
    procedure Pop (Stack : in out Stack_T; Item : out Integer);
end Noncompliant;
```

Note that both type Content_T, as well as the record type components of type Stack_T, are visible to clients. Client code may declare variables of type Content_T and may directly access and modify the record components. Bugs introduced via this access could be anywhere in the entire client codebase.

99.4.5 Compliant Code Example

```ada
package Compliant is
    type Stack_T (Capacity : Capacity_T) is tagged private;
    procedure Push (Stack : in out Stack_T; Item : Integer);
    procedure Pop (Stack : in out Stack_T; Item : out Integer);
private
    type Content_T is array (Capacity_T range <>) of Integer;
    type Stack_T (Capacity : Capacity_T) is tagged record
        Content : Content_T (1 .. Capacity);
        Top : Capacity_T := 0;
    end record;
end Compliant;
```

Type Content_T, as well as the record type components of type Stack_T, are no longer visible to clients. Any bugs in the stack processing code must be in this package, or its child packages, if any.
99.4.6 Notes

The GNATcheck rule specified above is not exhaustive.
REFERENCES

- AdaCore. SPARK 2014 User's Guide.\textsuperscript{379}
- AdaCore. GNAT User's Guide for Native Platforms\textsuperscript{380}
- AdaCore. “GNATstack User's Guide”\textsuperscript{381}

\textsuperscript{379} \url{http://docs.adacore.com/spark2014-docs/html/ug/index.html}
\textsuperscript{380} \url{http://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html}
\textsuperscript{381} \url{http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html}
Part XI

Introduction to Ada: Laboratories
These labs contain exercises for the *Introduction to Ada* (page 5) course.

This document was written by Gustavo A. Hoffmann and reviewed by Michael Frank.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

---

382 http://creativecommons.org/licenses/by-sa/4.0
CHAPTER ONE

IMPERATIVE LANGUAGE

For the exercises below (except for the first one), don't worry about the details of the Main procedure. You should just focus on implementing the application in the subprogram specified by the exercise.

101.1 Hello World

Goal: create a "Hello World!" application.

Steps:
1. Complete the Main procedure.

Requirements:
1. The application must display the message "Hello World!".

Listing 1: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
begin
  -- Implement the application here!
  null;
end Main;
```

101.2 Greetings

Goal: create an application that greets a person.

Steps:
1. Complete the Greet procedure.

Requirements:
1. Given an input string <name>, procedure Greet must display the message "Hello <name>!".
   1. For example, if the name is "John", it displays the message "Hello John!".

Remarks:
1. You can use the concatenation operator (&).
Listing 2: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    procedure Greet (Name : String) is
    begin
        -- Implement the application here!
        null;
    end Greet;

    begin
        if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
        end if;
        Greet (Argument (1));
    end Main;
```

101.3 Positive Or Negative

**Goal:** create an application that classifies integer numbers.

**Steps:**

1. Complete the Classify_Number procedure.

**Requirements:**

1. Given an integer number X, procedure Classify_Number must classify X as positive, negative or zero and display the result:
   1. If X > 0, it displays Positive.
   2. If X < 0, it displays Negative.
   3. If X = 0, it displays Zero.

Listing 3: classify_number.ads

```ada
procedure Classify_Number (X : Integer);
```

Listing 4: classify_number.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Classify_Number (X : Integer) is
begin
    -- Implement the application here!
    null;
end Classify_Number;
```
101.4 Numbers

Goal: create an application that displays numbers in a specific order.

Steps:
1. Complete the Display_Numbers procedure.

Requirements:
1. Given two integer numbers, Display_Numbers displays all numbers in the range starting with the smallest number.
return;
elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments...");
end if;

A := Integer‘Value (Argument (1));
B := Integer‘Value (Argument (2));
Display_Numbers (A, B);
end Main;
### 102.1 Subtract procedure

**Goal:** write a procedure that subtracts two numbers.

**Steps:**
1. Complete the procedure Subtract.

**Requirements:**
1. Subtract performs the operation $A - B$.

---

#### Listing 1: subtract.ads

```plaintext
-- Write the correct parameters for the procedure below.

procedure Subtract;
```

#### Listing 2: subtract.adb

```plaintext
procedure Subtract is
begin
  -- Implement the procedure here.
  null;
end Subtract;
```

#### Listing 3: main.adb

```plaintext
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Subtract;

procedure Main is
  type Test_Case_Index is
    (Sub_10_1_Chk,
     Sub_10_100_Chk,
     Sub_0_5_Chk,
     Sub_0_Minus_5_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Sub_10_1_Chk =>
        Subtract (10, 1, Result);
        Put_Line ("Result: " & Integer'Image (Result));
      when Sub_10_100_Chk =>
        Subtract (10, 100, Result);
      (continues on next page)
```
102.2 Subtract function

Goal: write a function that subtracts two numbers.

Steps:
1. Rewrite the Subtract procedure from the previous exercise as a function.

Requirements:
1. Subtract performs the operation \(A - B\) and returns the result.
procedure Check (TC : Test_Case_Index) is
  Result : Integer;
begin
  case TC is
  when Sub_10_1_Chk =>
    Result := Subtract (10, 1);
    Put_Line ("Result: " & Integer'Image (Result));
  when Sub_10_100_Chk =>
    Result := Subtract (10, 100);
    Put_Line ("Result: " & Integer'Image (Result));
  when Sub_0_5_Chk =>
    Result := Subtract (0, 5);
    Put_Line ("Result: " & Integer'Image (Result));
  when Sub_0_Minus_5_Chk =>
    Result := Subtract (0, -5);
    Put_Line ("Result: " & Integer'Image (Result));
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

102.3 Equality function

Goal: write a function that compares two values and returns a flag.

Steps:
1. Complete the Is_Equal subprogram.

Requirements:
1. Is_Equal returns a flag as a Boolean value.
2. The flag must indicate whether the values are equal (flag is True) or not (flag is False).

Listing 7: is_equal.ads
--- Write the correct signature for the function below.
--- Don't forget to replace the keyword "procedure" by "function."
procedure Is_Equal;

Listing 8: is_equal.adb

procedure Is_Equal is
begin
  -- Implement the function here!
  null;
end Is_Equal;
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Is_Equal;

procedure Main is
  type Test_Case_Index is
    (Equal_Chk, Inequal_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Display_Equal (A, B : Integer; Equal : Boolean) is
      begin
        Put (Integer'Image (A));
        if Equal then
          Put (" is equal to ");
        else
          Put (" isn't equal to ");
        end if;
        Put_Line (Integer'Image (B) & ".");
      end Display_Equal;

      Result : Boolean;
    begin
      case TC is
        when Equal_Chk =>
          for I in 0 .. 10 loop
            Result := Is_Equal (I, I);
            Display_Equal (I, I, Result);
          end loop;
        when Inequal_Chk =>
          for I in 0 .. 10 loop
            Result := Is_Equal (I, I - 1);
            Display_Equal (I, I - 1, Result);
          end loop;
      end case;
    end Check;
  begin
    if Argument_Count < 1 then
      Put_Line("ERROR: missing arguments! Exiting...");
      return;
    elsif Argument_Count > 1 then
      Put_Line("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
  end Main;
102.4 States

**Goal:** write a procedure that displays the state of a machine.

**Steps:**

1. Complete the procedure `Display_State`.

**Requirements:**

1. The states can be set according to the following numbers:

<table>
<thead>
<tr>
<th>Number</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Off</td>
</tr>
<tr>
<td>1</td>
<td>On: Simple Processing</td>
</tr>
<tr>
<td>2</td>
<td>On: Advanced Processing</td>
</tr>
</tbody>
</table>

2. The procedure `Display_State` receives the number corresponding to a state and displays the state (indicated by the table above) as a user message.

**Remarks:**

1. You can use a case statement to implement this procedure.

Listing 10: display_state.ads

```ada
procedure Display_State (State : Integer);
```

Listing 11: display_state.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_State (State : Integer) is
begin
null;
end Display_State;
```

Listing 12: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_State;

procedure Main is
State : Integer;
beg
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting...");
return;
elif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;
State := Integer'Value (Argument (1));
Display_State (State);
end Main;
```
102.5 States #2

**Goal:** write a function that returns the state of a machine.

**Steps:**
1. Implement the function `Get_State`.

**Requirements:**
1. Implement same state machine as in the previous exercise.
2. Function `Get_State` must return the state as a string.

**Remarks:**
1. You can implement a function returning a string by simply using quotes in a return statement. For example:

```
Listing 13: get_hello.ads
function Get_Hello return String;
```

```
Listing 14: get_hello.adb
function Get_Hello return String is
begin
return "Hello";
end Get_Hello;
```

```
Listing 15: main.adb
with Ada.Text_IO; use Ada.Text_IO;
with Get_Hello;
procedure Main is
S : constant String := Get_Hello;
begin
Put_Line (S);
end Main;
```

2. You can reuse your previous implementation and replace it by a case expression.
   1. For values that do not correspond to a state, you can simply return an empty string (""").

```
Listing 16: get_state.ads
function Get_State (State : Integer) return String;
```

```
Listing 17: get_state.adb
function Get_State (State : Integer) return String is
begin
return "";
end Get_State;
```

```
Listing 18: main.adb
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Get_State;
```

procedure Main is
  State : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  State := Integer'Value (Argument (1));
  Put_Line (Get_State (State));
end Main;

102.6 States #3

Goal: implement an on/off indicator for a state machine.

Steps:
1. Implement the function Is_On.
2. Implement the procedure Display_On_Off.

Requirements:
1. Implement same state machine as in the previous exercise.
2. Function Is_On returns:
   • True if the machine is on;
   • otherwise, it returns False.
3. Procedure Display_On_Off displays the message
   • "On" if the machine is on, or
   • "Off" otherwise.
4. Is_On must be called in the implementation of Display_On_Off.

Remarks:
1. You can implement both subprograms using if expressions.

Listing 19: is_on.ads
function Is_On (State : Integer) return Boolean;

Listing 20: is_on.adb
function Is_On (State : Integer) return Boolean is
begin
  return False;
end Is_On;

Listing 21: display_on_off.ads
procedure Display_On_Off (State : Integer);
### Learning Ada

#### Listing 22: display_on_off.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Is_On;

procedure Display_On_Off (State : Integer) is
  begin
    Put_Line ("");
  end Display_On_Off;
```

#### Listing 33: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_On_Off;
with Is_On;

procedure Main is
  State : Integer;
  begin
    if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting..."
               & return);
    elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
    end if;
    State := Integer'Value (Argument (1));
    Display_On_Off (State);
    Put_Line (Boolean'Image (Is_On (State)));
  end Main;
```

## 102.7 States #4

**Goal:** implement a procedure to update the state of a machine.

**Steps:**
1. Implement the procedure `Set_Next`.

**Requirements:**
1. Implement the same state machine as in the previous exercise.
2. Procedure `Set_Next` updates the machine's state with the next one in a *circular* manner:
   - In most cases, the next state of `N` is simply the next number (`N + 1`).
   - However, if the state is the last one (which is 2 for our machine), the next state must be the first one (in our case: 0).

**Remarks:**
1. You can use an `if` expression to implement `Set_Next`. 
Listing 24: set_next.ads

```ada
procedure Set_Next (State : in out Integer);
```

Listing 25: set_next.adb

```ada
procedure Set_Next (State : in out Integer) is
begin
    null;
end Set_Next;
```

Listing 26: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Set_Next;

procedure Main is
    State : Integer;
begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
    end if;
    State := Integer'Value (Argument (1));
    Set_Next (State);
    Put_Line (Integer'Image (State));
end Main;
```
103.1 Months

Goal: create a package to display the months of the year.

Steps:
1. Convert the Months procedure below to a package.
2. Create the specification and body of the Months package.

Requirements:
1. Months must contain the declaration of strings for each month of the year, which are stored in three-character constants based on the month's name.
   • For example, the string "January" is stored in the constant Jan. These strings are then used by the Display_Months procedure, which is also part of the Months package.

Remarks:
1. The goal of this exercise is to create the Months package.
   1. In the code below, Months is declared as a procedure.
      • Therefore, we need to convert it into a real package.
   2. You have to modify the procedure declaration and implementation in the code below, so that it becomes a package specification and a package body.

Listing 1: months.ads

```
-- Create specification for Months package, which includes
-- the declaration of the Display_Months procedure.
--
procedure Months;
```

Listing 2: months.adb

```
-- Create body of Months package, which includes
-- the implementation of the Display_Months procedure.
--
procedure Months is

  procedure Display_Months is
  begin
    Put_Line ("Months:");
    Put_Line ("- " & Jan);
    Put_Line ("- " & Feb);
    Put_Line ("- " & Mar);

(continues on next page)```
12   Put_Line ("- " & Apr);
13   Put_Line ("- " & May);
14   Put_Line ("- " & Jun);
15   Put_Line ("- " & Jul);
16   Put_Line ("- " & Aug);
17   Put_Line ("- " & Sep);
18   Put_Line ("- " & Oct);
19   Put_Line ("- " & Nov);
20   Put_Line ("- " & Dec);
21   end Display_Months;
22
23   begin
24      null;
25   end Months;

Listing 3: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Months; use Months;

procedure Main is

   type Test_Case_Index is
      (Months_Chk);

   procedure Check (TC : Test_Case_Index) is
   begin
      case TC is
      when Months_Chk =>
         Display_Months;
      end case;
   end Check;

begin
   if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
   elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
   end if;
   Check (Test_Case_Index'Value (Argument (1)));
end Main;

103.2 Operations

Goal: create a package to perform basic mathematical operations.

Steps:
1. Implement the Operations package.
   1. Declare and implement the Add function.
   2. Declare and implement the Subtract function.
   3. Declare and implement the Multiply: function.
4. Declare and implement the Divide function.

2. Implement the Operations.Test package
   1. Declare and implement the Display procedure.

Requirements:
1. Package Operations contains functions for each of the four basic mathematical operations for parameters of \texttt{Integer} type:
   1. Function \texttt{Add} performs the addition of \texttt{A} and \texttt{B} and returns the result;
   2. Function \texttt{Subtract} performs the subtraction of \texttt{A} and \texttt{B} and returns the result;
   3. Function \texttt{Multiply} performs the multiplication of \texttt{A} and \texttt{B} and returns the result;
   4. Function \texttt{Divide} performs the division of \texttt{A} and \texttt{B} and returns the result.

2. Package Operations.Test contains the test environment:
   1. Procedure \texttt{Display} must use the functions from the parent (Operations) package as indicated by the template in the code below.

\begin{verbatim}
1 package Operations is
2    -- Create specification for Operations package, including the
3    -- declaration of the functions mentioned above.
4    --
5 end Operations;

1 package body Operations is
2    -- Create body of Operations package.
3    --
4 end Operations;

1 package Operations.Test is
2    -- Create specification for Operations package, including the
3    -- declaration of the Display procedure:
4    --
5    -- procedure Display (A, B : Integer);
6    --
7 end Operations.Test;

1 package body Operations.Test is
2    -- Implement body of Operations.Test package.
3    --
4    procedure Display (A, B : Integer) is
5        A_Str : constant String := Integer'Image (A);
6        B_Str : constant String := Integer'Image (B);
7    end Display;
8 end Operations.Test;
\end{verbatim}
begin
Put_Line ("Operations:");
Put_Line (A_Str & " + " & B_Str & " = \\
& " & Integer'Image (Add (A, B)));
-- Use the line above as a template and add the rest of the 
-- implementation for Subtract, Multiply and Divide.
end Display;
end Operations.Test;

Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Operations;
with Operations.Test; use Operations.Test;

procedure Main is

   type Test_Case_Index is
      (Operations_Chk,
       Operations_Display_Chk);

   procedure Check (TC : Test_Case_Index) is
      begin
         case TC is
            when Operations_Chk =>
               Put_Line ("Add (100, 2) = " \\
                & Integer'Image (Operations.Add (100, 2)));
               Put_Line ("Subtract (100, 2) = " \\
                & Integer'Image (Operations.Subtract (100, 2)));
               Put_Line ("Multiply (100, 2) = " \\
                & Integer'Image (Operations.Multiply (100, 2)));
               Put_Line ("Divide (100, 2) = " \\
                & Integer'Image (Operations.Divide (100, 2)));
            when Operations_Display_Chk =>
               Display (10, 5);
               Display (1, 2);
         end case;
      end Check;
   begin
      if Argument_Count < 1 then
         Put_Line ("ERROR: missing arguments! Exiting...");
         return;
      elsif Argument_Count > 1 then
         Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
end Main;
104.1 Colors

**Goal:** create a package to represent HTML colors in hexadecimal form and its corresponding names.

**Steps:**
1. Implement the Color_Types package.
   1. Declare the HTML_Color enumeration.
   2. Declare the Basic_HTML_Color enumeration.
   3. Implement the To_Integer function.
   4. Implement the To_HTML_Color function.

**Requirements:**
1. Enumeration HTML_Color has the following colors:
   - Salmon
   - Firebrick
   - Red
   - Darkred
   - Lime
   - Forestgreen
   - Green
   - Darkgreen
   - Blue
   - Mediumblue
   - Darkblue

2. Enumeration Basic_HTML_Color has the following colors: Red, Green, Blue.
3. Function To_Integer converts from the HTML_Color type to the HTML color code — as integer values in hexadecimal notation.
   - You can find the HTML color codes in the table below.
4. Function To_HTML_Color converts from Basic_HTML_Color to HTML_Color.
5. This is the table to convert from an HTML color to a HTML color code in hexadecimal notation:
### Learning Ada

#### Color

<table>
<thead>
<tr>
<th>Color</th>
<th>HTML color code (hexa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salmon</td>
<td>#FA8072</td>
</tr>
<tr>
<td>Firebrick</td>
<td>#B22222</td>
</tr>
<tr>
<td>Red</td>
<td>#FF0000</td>
</tr>
<tr>
<td>Darkred</td>
<td>#8B0000</td>
</tr>
<tr>
<td>Lime</td>
<td>#00FF00</td>
</tr>
<tr>
<td>Forestgreen</td>
<td>#228B22</td>
</tr>
<tr>
<td>Green</td>
<td>#008000</td>
</tr>
<tr>
<td>Darkgreen</td>
<td>#006400</td>
</tr>
<tr>
<td>Blue</td>
<td>#0000FF</td>
</tr>
<tr>
<td>Mediumblue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>Darkblue</td>
<td>#00008B</td>
</tr>
</tbody>
</table>

#### Remarks:

1. In order to express the hexadecimal values above in Ada, use the following syntax: `16#<hex_value>`. (e.g.: `16#FFFFFF`).

2. For function `To_Integer`, you may use a **case** for this.

Listing 1: color_types.ads

```ada
package Color_Types is

  -- Include type declaration for HTML_Color!
  -- type HTML_Color is [...] 
  --

  -- Include function declaration for:
  -- function To_Integer (C : HTML_Color) return Integer;
  --

  -- Include type declaration for Basic_HTML_Color!
  --
  -- type Basic_HTML_Color is [...] 
  --

  -- Include function declaration for:
  -- Basic_HTML_Color => HTML_Color
  --
  -- function To_HTML_Color [...] ;

end Color_Types;
```

Listing 2: color_types.adb

```ada
package body Color_Types is

  -- Implement the conversion from HTML_Color to Integer here!
  --
  -- function To_Integer (C : HTML_Color) return Integer is
  -- begin
  --   -- Hint: use 'case' for the HTML colors;
  --   -- use 16#....# for the hexadecimal values.
  -- end To_Integer;

  -- Implement the conversion from Basic_HTML_Color to HTML_Color here!
  --
  -- function To_HTML_Color [...] is
```

(continues on next page)
end Color_Types;

Listing 3: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO;
with Color_Types; use Color_Types;

procedure Main is

   type Test_Case_Index is
      (HTML_Color_Range,
       HTML_Color_To_Integer,
       Basic_HTML_Color_To_HTML_Color);

   procedure Check (TC : Test_Case_Index) is
     begin
       case TC is
         when HTML_Color_Range =>
           for I in HTML_Color'Range loop
             Put_Line (HTML_Color'Image (I));
           end loop;
         when HTML_Color_To_Integer =>
           for I in HTML_Color'Range loop
             Ada.Integer_Text_IO.Put (Item => To_Integer (I),
                                        Width => 6,
                                        Base => 16);
             New_Line;
           end loop;
         when Basic_HTML_Color_To_HTML_Color =>
           for I in Basic_HTML_Color'Range loop
             Put_Line (HTML_Color'Image (To_HTML_Color (I)));
           end loop;
       end case;
     end case;
   end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

104.1. Colors
104.2 Integers

Goal: implement a package with various integer types.

Steps:
1. Implement the Int_Types package.
   1. Declare the integer type I_100.
   2. Declare the modular type U_100.
   3. Implement the To_I_100 function to convert from the U_100 type.
   4. Implement the To_U_100 function to convert from the I_100 type.
   5. Declare the derived type D_50.
   6. Declare the subtype S_50.
   7. Implement the To_D_50 function to convert from the I_100 type.
   8. Implement the To_S_50 function to convert from the I_100 type.
   9. Implement the To_I_100 function to convert from the D_50 type.

Requirements:
1. Types I_100 and U_100 have values between 0 and 100.
   1. Type I_100 is an integer type.
   2. Type U_100 is a modular type.
2. Function To_I_100 converts from the U_100 type to the I_100 type.
3. Function To_U_100 converts from the I_100 type to the U_100 type.
4. Types D_50 and S_50 have values between 10 and 50 and use I_100 as a base type.
   1. D_50 is a derived type.
   2. S_50 is a subtype.
5. Function To_D_50 converts from the I_100 type to the D_50 type.
6. Function To_S_50 converts from the I_100 type to the S_50 type.
7. Functions To_D_50 and To_S_50 saturate the input values if they are out of range.
   • If the input is less than 10 the output should be 10.
   • If the input is greater than 50 the output should be 50.
8. Function To_I_100 converts from the D_50 type to the I_100 type.

Remarks:
1. For the implementation of functions To_D_50 and To_S_50, you may use the type attributes D_50'First and D_50'Last:
   1. D_50'First indicates the minimum value of the D_50 type.
   2. D_50'Last indicates the maximum value of the D_50 type.
   3. The same attributes are available for the S_50 type (S_50'First and S_50'Last).
2. We could have implemented a function To_I_100 as well to convert from S_50 to I_100. However, we skip this here because explicit conversions are not needed for subtypes.
Listing 4: int_types.ads

package Int_Types is
  -- Include type declarations for I_100 and U_100!
  --
  -- type I_100 is [...]
  -- type U_100 is [...]
  --
  function To_I_100 (V : U_100) return I_100;
  function To_U_100 (V : I_100) return U_100;
  -- Include type declarations for D_50 and S_50!
  -- [...]
  -- [...]
  --
  function To_D_50 (V : I_100) return D_50;
  function To_S_50 (V : I_100) return S_50;
  function To_I_100 (V : D_50) return I_100;
end Int_Types;

Listing 5: int_types.adb

package body Int_Types is
  function To_I_100 (V : U_100) return I_100 is
    begin
      -- Implement the conversion from U_100 to I_100 here!
      null;
    end To_I_100;
  function To_U_100 (V : I_100) return U_100 is
    begin
      -- Implement the conversion from I_100 to U_100 here!
      null;
    end To_U_100;
  function To_D_50 (V : I_100) return D_50 is
    Min : constant I_100 := I_100 (D_50'First);
    Max : constant I_100 := I_100 (D_50'Last);
    begin
      -- Implement the conversion from I_100 to D_50 here!
      --
      -- Hint: using the constants above simplifies the checks needed for
      -- this function.
      null;
    end To_D_50;
  function To_S_50 (V : I_100) return S_50 is
    begin
      -- Implement the conversion from I_100 to S_50 here!
    end To_S_50;
end Int_Types;

104.2. Integers
null;
end To_S_50;

function To_I_100 (V : D_50) return I_100 is
begin
-- Implement the conversion from I_100 to D_50 here!
-- Remark: don't forget to verify whether an explicit conversion like
-- I_100 (V) is needed.
null;
end To_I_100;
end Int_Types;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Int_Types; use Int_Types;

procedure Main is
package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
package D_50_IO is new Ada.Text_IO.Integer_IO (D_50);

use I_100_IO;
use U_100_IO;
use D_50_IO;

type Test_Case_Index is
(I_100_Range,
 U_100_Range,
 U_100_Wraparound,
 U_100_To_I_100,
 I_100_To_U_100,
 D_50_Range,
 S_50_Range,
 I_100_To_D_50,
 I_100_To_S_50,
 D_50_To_I_100,
 S_50_To_I_100);

procedure Check (TC : Test_Case_Index) is
begin
I_100_IO.Default_Width := 1;
U_100_IO.Default_Width := 1;
D_50_IO.Default_Width := 1;

case TC is
when I_100_Range =>
  Put (I_100'First);
  New_Line;
  Put (I_100'Last);
  New_Line;
when U_100_Range =>
(continues on next page)
Put (U_100'First);
New_Line;
Put (U_100'Last);
New_Line;
when U_100_WRAPAROUND =>
Put (U_100'First - 1);
New_Line;
Put (U_100'Last + 1);
New_Line;
when U_100_TO_I_100 =>
for I in U_100'RANGE loop
 I_100_IO.Put (TO_I_100 (I));
 New_Line;
end loop;
when I_100_TO_U_100 =>
for I in I_100'RANGE loop
 Put (TO_U_100 (I));
 New_Line;
end loop;
when D_50_RANGE =>
Put (D_50'First);
New_Line;
Put (D_50'Last);
New_Line;
when S_50_RANGE =>
Put (S_50'First);
New_Line;
Put (S_50'Last);
New_Line;
when I_100_TO_D_50 =>
for I in I_100'RANGE loop
 Put (TO_D_50 (I));
 New_Line;
end loop;
when I_100_TO_S_50 =>
for I in I_100'RANGE loop
 Put (TO_S_50 (I));
 New_Line;
end loop;
when D_50_TO_I_100 =>
for I in D_50'RANGE loop
 Put (TO_I_100 (I));
 New_Line;
end loop;
when S_50_TO_I_100 =>
for I in S_50'RANGE loop
 Put (I);
 New_Line;
end loop;
end case;
end Check;

begin
 if Argument_Count < 1 then
  Put_Line ("ERROR: missing arguments! Exiting...");
 return;
 elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
 end if;
Check (Test_Case_Index'Value (Argument (1)));
104.3 Temperatures

Goal: create a package to handle temperatures in Celsius and Kelvin.

Steps:
1. Implement the Temperature_Types package.
   1. Declare the Celsius type.
   2. Declare the Int_Celsius type.
   3. Implement the To_Celsius function.
   4. Implement the To_Int_Celsius function.
   5. Declare the Kelvin type.
   6. Implement the To_Celsius function to convert from the Kelvin type.
   7. Implement the To_Kelvin function.

Requirements:
1. The custom floating-point types declared in Temperature_Types must use a precision of six digits.
2. Types Celsius and Int_Celsius are used for temperatures in Celsius:
   1. Celsius is a floating-point type with a range between -273.15 and 5504.85.
   2. Int_Celsius is an integer type with a range between -273 and 5505.
3. Functions To_Celsius and To_Int_Celsius are used for type conversion:
   1. To_Celsius converts from Int_Celsius to Celsius type.
   2. To_Int_Celsius converts from Celsius and Int_Celsius types:
4. Kelvin is a floating-point type for temperatures in Kelvin using a range between 0.0 and 5778.0.
5. The functions To_Celsius and To_Kelvin are used to convert between temperatures in Kelvin and Celsius.
   1. In order to convert temperatures in Celsius to Kelvin, you must use the formula \( K = C + 273.15 \), where:
      • \( K \) is the temperature in Kelvin, and
      • \( C \) is the temperature in Celsius.

Remarks:
1. When implementing the To_Celsius function for the Int_Celsius type:
   1. You’ll need to check for the minimum and maximum values of the input values because of the slightly different ranges.
   2. You may use variables of floating-point type (Float) for intermediate values.
2. For the implementation of the functions To_Celsius and To_Kelvin (used for converting between Kelvin and Celsius), you may use a variable of floating-point type (Float) for intermediate values.
Listing 7: temperature_types.ads

```ada
package Temperature_Types is

-- Include type declaration for Celsius!
-- Celsius is [...];
-- Int_Celsius is [...];

-- function To_Celsius (T : Int_Celsius) return Celsius;
-- function To_Int_Celsius (T : Celsius) return Int_Celsius;

-- Include type declaration for Kelvin!
-- type Kelvin is [...];
-- Include function declarations for:
-- - Kelvin => Celsius
-- - Celsius => Kelvin
-- function To_Celsius [...];
-- function To_Kelvin [...];

end Temperature_Types;
```

Listing 8: temperature_types.adb

```ada
package body Temperature_Types is

-- function To_Celsius (T : Int_Celsius) return Celsius is
begin
  null;
end To_Celsius;

-- function To_Int_Celsius (T : Celsius) return Int_Celsius is
begin
  null;
end To_Int_Celsius;

-- Include function implementation for:
-- - Kelvin => Celsius
-- - Celsius => Kelvin
-- function To_Celsius [...] is
-- function To_Kelvin [...] is

end Temperature_Types;
```

Listing 9: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Temperature_Types; use Temperature_Types;

procedure Main is

  package Celsius_IO is new Ada.Text_IO.Float_IO (Celsius);
  package Kelvin_IO is new Ada.Text_IO.Float_IO (Kelvin);
```

(continues on next page)
package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);

use Celsius_IO;
use Kelvin_IO;
use Int_Celsius_IO;

type Test_Case_Index is (
  Celsius_Range,
  Celsius_To_Int_Celsius,
  Int_Celsius_To_Celsius,
  Celsius_To_Kelvin);

procedure Check (TC : Test_Case_Index) is begin
  Celsius_IO.Default_Fore := 1;
  Kelvin_IO.Default_Fore := 1;
  Int_Celsius_IO.Default_Width := 1;

  case TC is
    when Celsius_Range =>
      Put (Celsius'First);
      New_Line;
      Put (Celsius'Last);
      New_Line;
    when Celsius_To_Int_Celsius =>
      Put (To_Int_Celsius (Celsius'First));
      New_Line;
      Put (To_Int_Celsius (0.0));
      New_Line;
      Put (To_Int_Celsius (Celsius'Last));
      New_Line;
    when Int_Celsius_To_Celsius =>
      Put (To_Celsius (Int_Celsius'First));
      New_Line;
      Put (To_Celsius (0));
      New_Line;
      Put (To_Celsius (Int_Celsius'Last));
      New_Line;
    when Kelvin_To_Celsius =>
      Put (To_Celsius (Kelvin'First));
      New_Line;
      Put (To_Celsius (0));
      New_Line;
      Put (To_Celsius (Kelvin'Last));
      New_Line;
    when Celsius_To_Kelvin =>
      Put (To_Kelvin (Celsius'First));
      New_Line;
      Put (To_Kelvin (Celsius'Last));
      New_Line;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;

(continues on next page)
Check (Test_Case_Index'Value (Argument (1)));
end Main;
Chapter 104. Strongly typed language
105.1 Directions

**Goal**: create a package that handles directions and geometric angles.

**Steps**: 
1. Implement the Directions package.
   1. Declare the Ext_Angle record.
   2. Implement the Display procedure.
   3. Implement the To Ext_Angle function.

**Requirements**: 
1. Record Ext_Angle stores information about the extended angle (see remark about extended angles below).
2. Procedure Display displays information about the extended angle.
   1. You should use the implementation that has been commented out (see code below) as a starting point.
3. Function To Ext_Angle converts a simple angle value to an extended angle (Ext_Angle type).

**Remarks**:
1. We make use of the algorithm implemented in the Check_Direction procedure (*chapter on imperative language* (page 9)).
2. For the sake of this exercise, we use the concept of extended angles. This includes the actual geometric angle and the corresponding direction (North, South, Northwest, and so on).

Listing 1: directions.ads

```
package Directions is
  type Angle_Mod is mod 360;
  type Direction is
    (North,
     Northeast,
     East,
     Southeast,
     South,
     Southwest,
     West,
     Northwest);
```
function To_Direction (N: Angle_Mod) return Direction;

-- Include type declaration for Ext_Angle record type:
--
-- NOTE: Use the Angle_Mod and Direction types declared above!
--
-- type Ext_Angle is [...]
--

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;

procedure Display (N : Ext_Angle);
end Directions;

Listing 2: directions.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Directions is

procedure Display (N : Ext_Angle) is
    -- Uncomment the code below and fill the missing elements
    --
    -- Put_Line ("Angle: 
    -- & Angle_Mod'Image (____)
    -- & " => "
    -- & Direction'Image (____)
    -- & ".");
    null;
end Display;

function To_Direction (N : Angle_Mod) return Direction is
    begin
        case N is
        when 0 => return North;
        when 1 .. 89 => return Northeast;
        when 90 => return East;
        when 91 .. 179 => return Southeast;
        when 180 => return South;
        when 181 .. 269 => return Southwest;
        when 270 => return West;
        when 271 .. 359 => return Northwest;
        end case;
    end To_Direction;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
    begin
        -- Implement the conversion from Angle_Mod to Ext_Angle here!
        --
        -- Hint: you can use a return statement and an aggregate.
        --
        null;
    end To_Ext_Angle;
end Directions;
105.2 Colors

**Goal:** create a package to represent HTML colors in RGB format using the hexadecimal form.

**Steps:**

1. Implement the Color_Types package.
   1. Declare the RGB record.
   2. Implement the To_RGB function.
   3. Implement the Image function for the RGB type.

**Requirements:**

1. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:
2. The hexadecimal information of each HTML color can be mapped to three color elements: red, green and blue.
   1. Each color element has a value between 0 and 255, or 00 and FF in hexadecimal.
   2. For example, for the color salmon, the hexadecimal value of the color elements are:
      • red = FA,
      • green = 80, and
      • blue = 72.
3. Record RGB stores information about HTML colors in RGB format, so that we can retrieve the individual color elements.
4. Function To_RGB converts from the HTML_Color enumeration to the RGB type based on the information from the table above.
5. Function Image returns a string representation of the RGB type in this format:
   • "(Red => 16#..#, Green => 16#...#, Blue => 16#...# )"

Remarks:
1. We use the exercise on HTML colors from the previous lab on Strongly typed language (page 1823) as a starting point.

Listing 4: color_types.ads

```
package Color_Types is

    type HTML_Color is
        (Salmon,
         Firebrick,
         Red,
         Darkred,
         Lime,
         Forestgreen,
         Green,
         Darkgreen,
         Blue,
         Mediumblue,
         Darkblue);

    function To_Integer (C : HTML_Color) return Integer;

    type Basic_HTML_Color is
        (Red,
         ...,
         Darkblue);
```

(continues on next page)
function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;

subtype Int_Color is Integer range 0 .. 255;

-- Replace type declaration for RGB record below
-- - NOTE: Use the Int_Color type declared above!
--
type RGB is null record;

function To_RGB (C : HTML_Color) return RGB;

function Image (C : RGB) return String;
end Color_Types;

with Ada.Integer_Text_IO;

package body Color_Types is

function To_Integer (C : HTML_Color) return Integer is
begin
  case C is
    when Salmon => return 16#FA8072#;
    when Firebrick => return 16#B22222#;
    when Red => return 16#FF0000#;
    when Darkred => return 16#8B0000#;
    when Lime => return 16#00FF00#;
    when Forestgreen => return 16#228B22#;
    when Green => return 16#008000#;
    when Darkgreen => return 16#006400#;
    when Blue => return 16#0000FF#;
    when Mediumblue => return 16#0000CD#;
    when Darkblue => return 16#00008B#;
  end case;
end To_Integer;

function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
begin
  case C is
    when Red => return Red;
    when Green => return Green;
    when Blue => return Blue;
  end case;
end To_HTML_Color;

function To_RGB (C : HTML_Color) return RGB is
begin
  -- Implement the conversion from HTML_Color to RGB here!
  --
  return (null record);
end To_RGB;

105.2. Colors
function Image (C : RGB) return String is
subtype Str_Range is Integer range 1 .. 10;
SR : String (Str_Range);
SG : String (Str_Range);
SB : String (Str_Range);
begina
  -- Replace argument in the calls to Put below
  -- with the missing elements (red, green, blue)
  -- from the RGB record
  --
  Ada.Integer_Text_IO.Put (To => SR,
    Item => 0, -- REPLACE!
    Base => 16);
  Ada.Integer_Text_IO.Put (To => SG,
    Item => 0, -- REPLACE!
    Base => 16);
  Ada.Integer_Text_IO.Put (To => SB,
    Item => 0, -- REPLACE!
    Base => 16);
  return
    "(Red => " & SR
    & ", Green => " & SG
    & ", Blue => " & SB
    & ")";
end Image;
end Color_Types;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;
procedure Main is
type Test_Case_Index is
  (HTML_Color_To_RGB);

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when HTML_Color_To_RGB =>
      for I in HTML_Color'Rangeloop
        Put_Line (HTML_Color'Image (I) & 
          " => "
        & Image (To_RGB (I)) & ").");
      end loop;
  end case;
end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting..."");
  return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
105.3 Inventory

**Goal:** create a simplified inventory system for a store to enter items and keep track of assets.

**Steps:**

1. Implement the Inventory_Pkg package.
   1. Declare the Item record.
   2. Implement the Init function.
   3. Implement the Add procedure.

**Requirements:**

1. Record Item collects information about products from the store.
   1. To keep it simple, this record only contains the name, quantity and price of each item.
   2. The record components are:
      - Name of `Item_Name` type;
      - Quantity of `Natural` type;
      - Price of `Float` type.
2. Function `Init` returns an initialized item (of Item type).
   1. Function `Init` must also display the item name by calling the `To_String` function for the `Item_Name` type.
      - This is already implemented in the code below.
   3. Procedure `Add` adds an item to the assets.
      1. Since we want to keep track of the assets, the implementation must accumulate the total value of each item's inventory, the result of multiplying the item quantity and its price.

Listing 7: inventory_pkg.ads

```ada
package Inventory_Pkg is

  type Item_Name is
    (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);

  function To_String (I : Item_Name) return String;

  -- Replace type declaration for Item record:
  --
  type Item is null record;

  function Init (Name : Item_Name;
      Quantity : Natural;
      Price : Float) return Item;

  procedure Add (Assets : in out Float;
      I : Item);

end Inventory_Pkg;
```

Listing 8: inventory_pkg.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Inventory_Pkg is

  function To_String (I : Item_Name) return String is
  begin
    case I is
      when Ballpoint_Pen => return "Ballpoint Pen";
      when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
      when Feather_Quill_Pen => return "Feather Quill Pen";
    end case;
  end To_String;

  function Init (Name : Item_Name; Quantity : Natural; Price : Float) return Item is
  begin
    Put_Line ("Item: " & To_String (Name) & ".");
    -- Replace return statement with the actual record initialization!
    return (null record);
  end Init;

  procedure Add (Assets : in out Float; I : Item) is
  begin
    -- Implement the function that adds an item to the inventory here!
    null;
  end Add;

end Inventory_Pkg;
```

Listing 9: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Inventory_Pkg; use Inventory_Pkg;

procedure Main is
  -- Remark: the following line is not relevant.
  F : array (1 .. 10) of Float := (others => 42.42);

  type Test_Case_Index is
    (Inventory_Chk);

  procedure Display (Assets : Float) is
    package F_IO is new Ada.Text_IO.Float_IO (Float);
    use F_IO;
  begin
    Put ("Assets: ");
    Put (Assets, 1, 2, 0);
    New_Line;
  end Display;
```

(continues on next page)
procedure Check (TC : Test_Case_Index) is
    I : Item;
    Assets : Float := 0.0;

    -- Please ignore the following three lines!
    pragma Warnings (Off, "default initialization");
    for Assets'Address use F'Address;
    pragma Warnings (On, "default initialization");

begin
    case TC is
        when Inventory_Chk =>
            I := Init (Ballpoint_Pen, 185, 0.15);
            Add (Assets, I);
            Display (Assets);

            I := Init (Oil_Based_Pen_Marker, 100, 9.0);
            Add (Assets, I);
            Display (Assets);

            I := Init (Feather_Quill_Pen, 2, 40.0);
            Add (Assets, I);
            Display (Assets);
        end case;
    end Check;

begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
    end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;
106.1 Constrained Array

**Goal:** declare a constrained array and implement operations on it.

**Steps:**
1. Implement the `Constrained_Arrays` package.
   1. Declare the range type `My_Index`.
   2. Declare the array type `My_Array`.
   3. Declare and implement the `Init` function.
   4. Declare and implement the `Double` procedure.
   5. Declare and implement the `FirstElem` function.
   6. Declare and implement the `LastElem` function.
   7. Declare and implement the `Length` function.
   8. Declare the object `A` of `My_Array` type.

**Requirements:**
1. Range type `My_Index` has a range from 1 to 10.
2. `My_Array` is a constrained array of `Integer` type.
   1. It must make use of the `My_Index` type.
   2. It is therefore limited to 10 elements.
3. Function `Init` returns an array where each element is initialized with the corresponding index.
4. Procedure `Double` doubles the value of each element of an array.
5. Function `FirstElem` returns the first element of the array.
6. Function `LastElem` returns the last element of the array.
7. Function `Length` returns the length of the array.
8. Object `A` of `My_Array` type is initialized with:
   1. the values 1 and 2 for the first two elements, and
   2. 42 for all other elements.
package Constrained_Arrays is

-- Complete the type and subprogram declarations:
--
-- type My_Index is [...]
--
-- type My_Array is [...]
--
-- function Init ...
--
-- procedure Double ...
--
-- function First_Element ...
--
-- function Last_Element ...
--
-- function Length ...
--
-- A : ...

end Constrained_Arrays;

package body Constrained_Arrays is

-- Create the implementation of the subprograms!
--

end Constrained_Arrays;

procedure Main is

type Test_Case_Index is
  (Range_CHK, Array_Range_CHK, A_Obj_CHK, Init_CHK, Double_CHK, First_Elem_CHK, Last_Elem_CHK, Length_CHK);

procedure Check (TC : Test_Case_Index) is
  AA : My_Array;

  procedure Display (A : My_Array) is
  begin
    for I in A'Range loop
      Put_Line (Integer'Image (A (I)));
    end loop;
  end Display;

  procedure Local_Init (A : in out My_Array) is

end Main;
begin
  A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
end Local_Init;
begin
  case TC is
  when Range_Chk =>
    for I in My_Index loop
      Put_Line (My_Index'Image (I));
    end loop;
  when Array_Range_Chk =>
    for I in My_Array'Range loop
      Put_Line (My_Index'Image (I));
    end loop;
  when A_Obj_Crk =>
    Display (A);
  when Init_Crk =>
    AA := Init;
    Display (AA);
  when Double_Crk =>
    Local_Init (AA);
    Double (AA);
    Display (AA);
  when First_Elem_Crk =>
    Local_Init (AA);
    Put_Line (Integer'Image (First_Elem (AA)));
  when Last_Elem_Crk =>
    Local_Init (AA);
    Put_Line (Integer'Image (Last_Elem (AA)));
  when Length_Crk =>
    Put_Line (Integer'Image (Length (AA)));
  end case;
end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

106.2 Colors: Lookup-Table

Goal: rewrite a package to represent HTML colors in RGB format using a lookup table.

Steps:

1. Implement the Color_Types package.
   1. Declare the array type HTML_Color_RGB.
   2. Declare the To_RGB_Lookup_Table object and initialize it.
   3. Adapt the implementation of the To_RGB function.

Requirements:

1. Array type HTML_Color_RGB is used for the table.
2. The To_RGB_Lookup_Table object of HTML_Color_RGB type contains the lookup table.
   - This table must be implemented as an array of constant values.

3. The implementation of the To_RGB function must use the To_RGB_Lookup_Table object.

Remarks:
1. This exercise is based on the HTML colors exercise from a previous lab (Records (page 1835)).
2. In the previous implementation, you could use a case statement to implement the To_RGB function. Here, you must rewrite the function using a look-up table.
   1. The implementation of the To_RGB function below includes the case statement as commented-out code. You can use this as your starting point: you just need to copy it and convert the case statement to an array declaration.
   2. Don't use a case statement to implement the To_RGB function. Instead, write code that accesses To_RGB_Lookup_Table to get the correct value.
3. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:

<table>
<thead>
<tr>
<th>Color</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salmon</td>
<td>#FA</td>
<td>#80</td>
<td>#72</td>
</tr>
<tr>
<td>Firebrick</td>
<td>#B2</td>
<td>#22</td>
<td>#22</td>
</tr>
<tr>
<td>Red</td>
<td>#FF</td>
<td>#00</td>
<td>#00</td>
</tr>
<tr>
<td>Darkred</td>
<td>#8B</td>
<td>#00</td>
<td>#00</td>
</tr>
<tr>
<td>Lime</td>
<td>#00</td>
<td>#FF</td>
<td>#00</td>
</tr>
<tr>
<td>Forestgreen</td>
<td>22</td>
<td>8B</td>
<td>#22</td>
</tr>
<tr>
<td>Green</td>
<td>#00</td>
<td>#80</td>
<td>#00</td>
</tr>
<tr>
<td>Darkgreen</td>
<td>#00</td>
<td>#64</td>
<td>#00</td>
</tr>
<tr>
<td>Blue</td>
<td>#00</td>
<td>#00</td>
<td>#FF</td>
</tr>
<tr>
<td>Mediumblue</td>
<td>#00</td>
<td>#00</td>
<td>#CD</td>
</tr>
<tr>
<td>Darkblue</td>
<td>#00</td>
<td>#00</td>
<td>#8B</td>
</tr>
</tbody>
</table>

Listing 4: color_types.ads

```ada
package Color_Types is

  type HTML_Color is
    (Salmon,
     Firebrick,
     Red,
     Darkred,
     Lime,
     Forestgreen,
     Green,
     Darkgreen,
     Blue,
     Mediumblue,
     Darkblue);

  subtype Int_Color is Integer range 0 .. 255;

  type RGB is record
    Red : Int_Color;
    Green : Int_Color;
    Blue : Int_Color;

```

(continues on next page)
end record;

function To_RGB (C : HTML_Color) return RGB;

function Image (C : RGB) return String;

with Ada.Integer_Text_IO;
package body Color_Types is
begin

function To_RGB (C : HTML_Color) return RGB is
begin
  -- Implement To_RGB using To_RGB_Lookup_Table
  return (0, 0, 0);
  -- Use the code below from the previous version of the To_RGB
  -- function to declare the To_RGB_Lookup_Table:
  -- case C is
  -- when Salmon => return (16#FA#, 16#80#, 16#72#);
  -- when Firebrick => return (16#B2#, 16#22#, 16#22#);
  -- when Red => return (16#FF#, 16#00#, 16#00#);
  -- when Darkred => return (16#8B#, 16#00#, 16#00#);
  -- when Lime => return (16#00#, 16#FF#, 16#00#);
  -- when Forestgreen => return (16#22#, 16#8B#, 16#22#);
  -- when Green => return (16#00#, 16#80#, 16#00#);
  -- when Darkgreen => return (16#00#, 16#64#, 16#00#);
  -- when Blue => return (16#00#, 16#00#, 16#FF#);
  -- when Mediumblue => return (16#00#, 16#00#, 16#CD#);
  -- when Darkblue => return (16#00#, 16#00#, 16#8B#);
  -- end case;
  end To_RGB;

function Image (C : RGB) return String is
subtype Str_Range is Integer range 1 .. 10;

SR : String (Str_Range);
SG : String (Str_Range);
SB : String (Str_Range);

begin
  Ada.Integer_Text_IO.Put (To => SR,
                          Item => C.Red,
                          Base => 16);
  Ada.Integer_Text_IO.Put (To => SG,
                          Item => C.Green,
                          Base => 16);
  Ada.Integer_Text_IO.Put (To => SB,
                          Item => C.Blue,
                          Base => 16);

  return "(Red => " & SR
& " Green => " & SG
& " Blue => " & SB
end Image;
end Color_Types;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is
    (Color_Table_Chk,
     HTML_Color_To_Integer_Chk);
  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Color_Table_Chk =>
        Put_Line ("Size of HTML_Color_RGB: 
          & Integer'Image (HTML_Color_RGB'Length));
        Put_Line ("Firebrick: 
          & Image (To_RGB_Lookup_Table (Firebrick)));
      when HTML_Color_To_Integer_Chk =>
        for I in HTML_Color'Range loop
          Put_Line (HTML_Color'Image (I) & " => " 
            & Image (To_RGB (I)) & ".");
        end loop;
    end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

106.3 Unconstrained Array

Goal: declare an unconstrained array and implement operations on it.
Steps:

1. Implement the Unconstrained_Arrays package.
   1. Declare the My_Array type.
   2. Declare and implement the Init procedure.
   3. Declare and implement the Init function.
4. Declare and implement the Double procedure.
5. Declare and implement the Diff_Prev_ELEM function.

Requirements:
1. My_Array is an unconstrained array (with a Positive range) of Integer elements.
2. Procedure Init initializes each element with the index starting with the last one.
   • For example, for an array of 3 elements where the index of the first element is 1 (My_Array (1 .. 3)), the values of these elements after a call to Init must be (3, 2, 1).
3. Function Init returns an array based on the length L and start index I provided to the Init function.
   1. I indicates the index of the first element of the array.
   2. L indicates the length of the array.
   3. Both I and L must be positive.
   4. This is its declaration: function Init (I, L : Positive) return My_Array;
   5. You must initialize the elements of the array in the same manner as for the Init procedure described above.
4. Procedure Double doubles each element of an array.
5. Function Diff_Prev_ELEM returns — for each element of an input array A — an array with the difference between an element of array A and the previous element.
   1. For the first element, the difference must be zero.
   2. For example:
      • INPUT: (2, 5, 15)
      • RETURN of Diff_Prev_ELEM: (0, 3, 10), where
        - 0 is the constant difference for the first element;
        - 5 - 2 = 3 is the difference between the second and the first elements of the input array;
        - 15 - 5 = 10 is the difference between the third and the second elements of the input array.

Remarks:
1. For an array A, you can retrieve the index of the last element with the attribute 'Last.
   1. For example: Y : Positive := A'Last;
   2. This can be useful during the implementation of procedure Init.
2. For the implementation of the Init function, you can call the Init procedure to initialize the elements. By doing this, you avoid code duplication.
3. Some hints about attributes:
   1. You can use the range attribute (A'Range) to retrieve the range of an array A.
   2. You can also use the range attribute in the declaration of another array (e.g.: B : My_Array (A'Range)).
   3. Alternatively, you can use the A'First and A'Last attributes in an array declaration.
package Unconstrained_Arrays is
  -- Complete the type and subprogram declarations:
  -- type My_Array is ...
  -- procedure Init ...;

  function Init (I, L : Positive) return My_Array;

  -- procedure Double ...
  -- function Diff_Prev_Elem ...
end Unconstrained_Arrays;

package body Unconstrained_Arrays is
  -- Implement the subprograms:
  -- procedure Init is...
  -- function Init (L : Positive) return My_Array is...
  -- procedure Double ... is...
  -- function Diff_Prev_Elem ... is...
end Unconstrained_Arrays;

procedure Main is
  type Test_Case_Index is
    (Init_Chk, Init_Proc_Chk, Double_Chk, Diff_Prev_Chk, Diff_Prev_Single_Chk);

  procedure Check (TC : Test_Case_Index) is
    AA : My_Array (1 .. 5);
    AB : My_Array (5 .. 9);

    procedure Display (A : My_Array) is
      begin
        for I in A'Range loop
          Put_Line (Integer'Image (A (I)));
        end loop;
      end Display;

    (continues on next page)
procedure Local_Init (A : in out My_Array) is
begin
    A := (1, 2, 5, 10, -10);
end Local_Init;

begin
    case TC is
        when Init_Chk =>
            AA := Init (AA'First, AA'Length);
            AB := Init (AB'First, AB'Length);
            Display (AA);
            Display (AB);
        when Init_Proc_Chk =>
            Init (AA);
            Init (AB);
            Display (AA);
            Display (AB);
        when Double_Chk =>
            Local_Init (AB);
            Double (AB);
            Display (AB);
        when Diff_Previous_Chk =>
            Local_Init (AB);
            AB := Diff_Previous (AB);
            Display (AB);
        when Diff_Previous_Single_Chk =>
            declare
                A1 : My_Array (1 .. 1) := (1 => 42);
            begin
                A1 := Diff_Previous (A1);
                Display (A1);
            end;
    end case;
end Check;

begin
    if Argument_Count < 1 then
        Put_Line("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line("Ignoring additional arguments...");
    end if;

    Check (Test_Case_Index'Value (Argument (1)));
end Main;

106.4 Product info

Goal: create a system to keep track of quantities and prices of products.

Steps:
1. Implement the Product_Info_Pkg package.
   1. Declare the array type ProductInfos.
   2. Declare the array type Currency_Array.
2. Implement the Total procedure.
3. Implement the Total function returning an array of Currency_Array type.
5. Implement the Total function returning a single value of Currency type.

Requirements:

1. Quantity of an individual product is represented by the Quantity subtype.
2. Price of an individual product is represented by the Currency subtype.
3. Record type Product_Info deals with information for various products.
4. Array type Product_Infos is used to represent a list of products.
5. Array type Currency_Array is used to represent a list of total values of individual products (see more details below).
6. Procedure Total receives an input array of products.
   1. It outputs an array with the total value of each product using the Currency_Array type.
   2. The total value of an individual product is calculated by multiplying the quantity for this product by its price.
7. Function Total returns an array of Currency_Array type.
   1. This function has the same purpose as the procedure Total.
   2. The difference is that the function returns an array instead of providing this array as an output parameter.
8. The second function Total returns a single value of Currency type.
   1. This function receives an array of products.
   2. It returns a single value corresponding to the total value for all products in the system.

Remarks:

1. You can use Currency (Q) to convert from an element Q of Quantity type to the Currency type.
   1. As you might remember, Ada requires an explicit conversion in calculations where variables of both integer and floating-point types are used.
   2. In our case, the Quantity subtype is based on the Integer type and the Currency subtype is based on the Float type, so a conversion is necessary in calculations using those types.

Listing 10: product_info_pkg.ads

```ada
package Product_Info_Pkg is

   subtype Quantity is Natural;

   subtype Currency is Float;

   type Product_Info is record
      Units : Quantity;
      Price : Currency;
   end record;

   -- Complete the type declarations:
   -- type Product_Infos is ...
   -- type Currency_Array is ...

   procedure Total (P : Product_Info;
```
package body Product_Info_Pkg is

-- Complete the subprogram implementations:
--
-- procedure Total (P : Product_Infos; Tot : out Currency_Array) is ...
-- function Total (P : Product_Infos) return Currency_Array is ...
-- function Total (P : Product_Infos) return Currency is ...

end Product_Info_Pkg;

procedure Main is

package Currency_IO is new Ada.Text_IO.Float_IO (Currency);

type Test_Case_Index is
  (Total_Func_Chk, Total_Proc_Chk, Total_Value_Chk);

procedure Check (TC : Test_Case_Index) is
  subtype Test_Range is Positive range 1 .. 5;
  P : Product_Infos (Test_Range);
  Tots : Currency_Array (Test_Range);
  Tot : Currency;

procedure Display (Tots : Currency_Array) is
  begin
    for I in Tots'Range loop
      Currency_IO.Put (Tots (I));
      New_Line;
    end loop;
  end Display;

procedure Local_Init (P : in out Product_Infos) is
  begin
    P := ((1, 0.5),
          (2, 10.0),
          (5, 40.0),
          (10, 10.0),
          (106.4, Product info 1855)
          (continues on next page)
35  \[(10, 20.0)\];
36  end Local_Init;
37
38  begin
39      Currency_IO.Default_Fore := 1;
40      Currency_IO.Default_Aft := 2;
41      Currency_IO.Default_Exp := 0;
42
43      case TC is
44         when Total_Func_Chk =>
45            Local_Init (P);
46            Tots := Total (P);
47            Display (Tots);
48         when Total_Proc_Chk =>
49            Local_Init (P);
50            Total (P, Tots);
51            Display (Tots);
52         when Total_Value_Chk =>
53            Local_Init (P);
54            Tot := Total (P);
55            Currency_IO.Put (Tot);
56            New_Line;
57      end case;
58   end case;
59   end begin;
60
61   begin
62      if Argument_Count < 1 then
63         Put_Line ("ERROR: missing arguments! Exiting...");
64      return;
65      elsif Argument_Count > 1 then
66         Put_Line ("Ignoring additional arguments...");
67      end if;
68   end Check;
69
70  end Main;

106.5 String_10

Goal: work with constrained string types.

Steps:
1. Implement the Strings_10 package.
   1. Declare the String_10 type.
   2. Implement the To_String_10 function.

Requirements:
1. The constrained string type String_10 is an array of ten characters.
2. Function To_String_10 returns constrained strings of String_10 type based on an input parameter of String type.
   • For strings that are more than 10 characters, omit everything after the 11th character.
   • For strings that are fewer than 10 characters, pad the string with ' ' characters until it is 10 characters.

Remarks:
1. Declaring String_10 as a subtype of String is the easiest way.
   • You may declare it as a new type as well. However, this requires some adaptations in the Main test procedure.

2. You can use Integer'Min to calculate the minimum of two integer values.

Listing 13: strings_10.ads

```ada
package Strings_10 is
  -- Complete the type and subprogram declarations:
  --
  -- subtype String_10 is ...;
  -- Using "type String_10 is..." is possible, too. However, it
  -- requires a custom Put_Line procedure that is called in Main:
  -- procedure Put_Line (S : String_10);
  -- function To_String_10 ...;
end Strings_10;
```

Listing 14: strings_10.adb

```ada
package body Strings_10 is
  -- Complete the subprogram declaration and implementation:
  --
  -- function To_String_10 ... is
end Strings_10;
```

Listing 15: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Strings_10; use Strings_10;

procedure Main is
  type Test_Case_Index is
    (String_10_Long_Chk, String_10_Short_Chk);

  procedure Check (TC : Test_Case_Index) is
    SL  : constant String := "And this is a long string just for testing...";
    SS  : constant String := "Hey!";
    S_10 : String_10;

  begin
    case TC is
      when String_10_Long_Chk =>
        S_10 := To_String_10 (SL);
        Put_Line (String (S_10));
      when String_10_Short_Chk =>
        S_10 := (others => ' ');
        S_10 := To_String_10 (SS);
        Put_Line (String (S_10));
    end case;
  end Check;
end Main;
```

(continues on next page)
begin
    if Argument_Count < 1 then
        Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
    end if;

    Check (Test_Case_Index'Value (Argument (1)));
end Main;

### 106.6 List of Names

**Goal:** create a system for a list of names and ages.

**Steps:**

1. Implement the Names_Ages package.
   1. Declare the People_Array array type.
   2. Complete the declaration of the People record type with the People_A element of People_Array type.
   3. Implement the Add procedure.
   4. Implement the Reset procedure.
   5. Implement the Get function.
   6. Implement the Update procedure.
   7. Implement the Display procedure.

**Requirements:**

1. Each person is represented by the Person type, which is a record containing the name and the age of that person.
2. People_Array is an unconstrained array of Person type with a positive range.
3. The Max_People constant is set to 10.
4. Record type People contains:
   1. The People_A element of People_Array type.
   2. This array must be constrained by the Max_People constant.
5. Procedure Add adds a person to the list.
   1. By default, the age of this person is set to zero in this procedure.
6. Procedure Reset resets the list.
7. Function Get retrieves the age of a person from the list.
8. Procedure Update updates the age of a person in the list.
9. Procedure Display shows the complete list using the following format:
   1. The first line must be LIST OF NAMES:. It is followed by the name and age of each person in the next lines.
   2. For each person on the list, the procedure must display the information in the following format:
Remarks:

1. In the implementation of procedure Add, you may use an index to indicate the last valid position in the array — see Last_Valid in the code below.

2. In the implementation of procedure Display, you should use the Trim function from the Ada.Strings.Fixed package to format the person's name — for example: Trim (P.Name, Right).

3. You may need the Integer'Min (A, B) and the Integer'Max (A, B) functions to get the minimum and maximum values in a comparison between two integer values A and B.

4. Fixed-length strings can be initialized with whitespaces using the others syntax. For example: S : String_10 := (others => ' ');

5. You may implement additional subprograms to deal with other types declared in the Names_Ages package below, such as the Name_Type and the Person type.
   1. For example, a function To_Name_Type to convert from String to Name_Type might be useful.
   2. Take a moment to reflect on which additional subprograms could be useful as well.

Listing 16: names_ages.ads

```ada
package Names_Ages is

  Max_People : constant Positive := 10;

  subtype Name_Type is String (1 .. 50);

  type Age_Type is new Natural;

  type Person is record
    Name : Name_Type;
    Age  : Age_Type;
  end record;

  -- Add type declaration for People_Array record:
  --
  --  type People_Array is ...;

  -- Replace type declaration for People record. You may use the
  -- following template:
  --
  --  type People is record
  --    People_A : People Array ...;
  --    Last_Valid : Natural;
  --  end record;

  type People is null record;

  procedure Reset (P : in out People);

  procedure Add (P : in out People;
                 Name : String);

  function Get (P : People;
                Name : String) return Age_Type;

end Names_Ages;
```

(continues on next page)
procedure Update (P : in out People;
    Name : String;
    Age : Age_Type);

procedure Display (P : People);

end Names_Ages;

Listing 17: names_ages.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Names_Ages is

    procedure Reset (P : in out People) is
    begin
        null;
    end Reset;

    procedure Add (P : in out People;
        Name : String) is
    begin
        null;
    end Add;

    function Get (P : People;
        Name : String) return Age_Type is
    begin
        return 0;
    end Get;

    procedure Update (P : in out People;
        Name : String;
        Age : Age_Type) is
    begin
        null;
    end Update;

    procedure Display (P : People) is
    begin
        null;
    end Display;

end Names_Ages;

Listing 18: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Names_Ages; use Names_Ages;

procedure Main is
    type Test_Case_Index is
        (Names_Ages_Chk,
         Get_Age_Chk);

    procedure Check (TC : Test_Case_Index) is
(continues on next page)
P : People;

begin
  case TC is
  when Names_Ages_Chk =>
    Reset (P);
    Add (P, "John");
    Add (P, "Patricia");
    Add (P, "Josh");
    Display (P);
    Update (P, "John", 18);
    Update (P, "Patricia", 35);
    Update (P, "Josh", 53);
    Display (P);
  when Get_Age_Chk =>
    Reset (P);
    Add (P, "Peter");
    Update (P, "Peter", 45);
    Put_Line ("Peter is 
      & Age_Type'Image (Get (P, "Peter"))
      & " years old.");
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
107.1 Aggregate Initialization

**Goal:** initialize records and arrays using aggregates.

**Steps:**
1. Implement the Aggregates package.
   1. Create the record type Rec.
   2. Create the array type Int_Arr.
   3. Implement the Init procedure that outputs a record of Rec type.
   4. Implement the Init_Some procedure.
   5. Implement the Init procedure that outputs an array of Int_Arr type.

**Requirements:**
1. Record type Rec has four components of Integer type. These are the components with the corresponding default values:
   - \( W = 10 \)
   - \( X = 11 \)
   - \( Y = 12 \)
   - \( Z = 13 \)
2. Array type Int_Arr has 20 elements of Integer type (with indices ranging from 1 to 20).
3. The first Init procedure outputs a record of Rec type where:
   1. \( X \) is initialized with 100,
   2. \( Y \) is initialized with 200, and
   3. the remaining elements use their default values.
4. Procedure Init_Some outputs an array of Int_Arr type where:
   1. the first five elements are initialized with the value 99, and
   2. the remaining elements are initialized with the value 100.
5. The second Init procedure outputs an array of Int_Arr type where:
   1. all elements are initialized with the value 5.
package Aggregates is
  -- type Rec is ...;
  -- type Int_Arr is ...;
  procedure Init;
  -- procedure Init_Some ...;
  -- procedure Init ...
end Aggregates;

package body Aggregates is
  procedure Init is null;
end Aggregates;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Aggregates; use Aggregates;

procedure Main is
  -- Remark: the following line is not relevant.
  F : array (1 .. 10) of Float := (others => 42.42)
  with Unreferenced;

  type Test_Case_Index is
    (Default_Rec_Chk,
     Init_Rec_Chk,
     Init_Some_Arr_Chk,
     Init_Arr_Chk);

  procedure Check (TC : Test_Case_Index) is
    A : Int_Arr;
    R : Rec;
    DR : constant Rec := (others => <>);
  begin
    case TC is
      when Default_Rec_Chk =>
        R := DR;
        Put_Line ("Record Default:");
        Put_Line ("W => " & Integer'Image (R.W));
        Put_Line ("X => " & Integer'Image (R.X));
        Put_Line ("Y => " & Integer'Image (R.Y));
        Put_Line ("Z => " & Integer'Image (R.Z));
      when Init_Rec_Chk =>
        Init (R);
        Put_Line ("Record Init:");
        Put_Line ("W => " & Integer'Image (R.W));
        Put_Line ("X => " & Integer'Image (R.X));
        Put_Line ("Y => " & Integer'Image (R.Y));
36  
37  when Init_Some_Arr_Chk =>  
38      Init_Some (A);  
39      Put_Line ("Array Init_Some:");  
40  for I in A'Range loop  
41      Put_Line (Integer'Image (I) & " 
42                  & Integer'Image (A (I)));  
43  end loop;  
44  when Init_Arr_Chk =>  
45      Init (A);  
46      Put_Line ("Array Init:");  
47  for I in A'Range loop  
48      Put_Line (Integer'Image (I) & " 
49                  & Integer'Image (A (I)));  
50  end loop;  
51  end case;  
52  end Check;  
53  
54  begin  
55  if Argument_Count < 1 then  
56      Put_Line ("ERROR: missing arguments! Exiting...");  
57      return;  
58  elsif Argument_Count > 1 then  
59      Put_Line ("Ignoring additional arguments...");  
60  end if;  
61  Check (Test_Case_Index'Value (Argument (1)));  
62  end Main;  
63

107.2 Versioning

**Goal:** implement a simple package for source-code versioning.

**Steps:**
1. Implement the Versioning package.  
   1. Declare the record type Version.  
   2. Implement the Convert function that returns a string.  
   3. Implement the Convert function that returns a floating-point number.

**Requirements:**
1. Record type Version has the following components of **Natural** type:  
   1. Major,  
   2. Minor, and  
2. The first Convert function returns a string containing the version number.  
3. The second Convert function returns a floating-point value.  
   1. For this floating-point value:  
      1. the number before the decimal point must correspond to the major number, and  
      2. the number after the decimal point must correspond to the minor number.
3. the maintenance number is ignored.
2. For example, version "1.3.5" is converted to the floating-point value 1.3.
3. An obvious limitation of this function is that it can only handle one-digit numbers for the minor component.
   - For example, we cannot convert version "1.10.0" to a reasonable value with the approach described above. The result of the call `Convert ((1, 10, 0))` is therefore unspecified.
   - For the scope of this exercise, only version numbers with one-digit components are checked.

Remarks:
1. We use overloading for the `Convert` functions.
2. For the function `Convert` that returns a string, you can make use of the `Image_Trim` function, as indicated in the source-code below — see package body of `Versioning`.

Listing 4: versioning.ads

```ada
package Versioning is
  -- type Version is record...
  -- function Convert ...
  -- function Convert
end Versioning;
```

Listing 5: versioning.adb

```ada
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Versioning is
  function Image_Trim (N : Natural) return String is
    S_N : constant String := Trim (Natural'Image (N), Left);
    begin
      return S_N;
    end Image_Trim;

    -- function Convert ...
    -- S_Major : constant String := Image_Trim (V.Major);
    -- S_Minor : constant String := Image_Trim (V.Minor);
    -- S_Maint : constant String := Image_Trim (V.Maintenance);
    -- begin
    --  end Convert;

    -- function Convert ...
    -- begin
    --  end Convert;

end Versioning;
```

Listing 6: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)
with Versioning; use Versioning;

procedure Main is
  type Test_Case_Index is
    (Ver_String_Chk,
     Ver_Float_Chk);

  procedure Check (TC : Test_Case_Index) is
    V : constant Version := (1, 3, 23);
  begin
    case TC is
      when Ver_String_Chk =>
        Put_Line (Convert (V));
      when Ver_Float_Chk =>
        Put_Line ('Float' Image (Convert (V)));
    end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

107.3 Simple todo list

**Goal:** implement a simple to-do list system.

**Steps:**

1. Implement the Todo_Lists package.
   1. Declare the Todo_Item type.
   2. Declare the Todo_List type.
   3. Implement the Add procedure.
   4. Implement the Display procedure.

**Requirements:**

1. Todo_Item type is used to store a to-do item.
   1. It should be implemented as an access type to strings.
2. Todo_Items type is an array of to-do items.
   1. It should be implemented as an unconstrained array with positive range.
3. Todo_List type is the container for all to-do items.
   1. This record type must have a discriminant for the maximum number of elements of the list.
   2. In order to store the to-do items, it must contain a component named Items of Todo_Items type.
   3. Don't forget to keep track of the last element added to the list!
• You should declare a Last component in the record.

4. Procedure Add adds items (of Todo_Item type) to the list (of Todo_List type).
   1. This requires allocating a string for the access type.
   2. An item can only be added to the list if the list isn’t full yet — see next point for details on error handling.

5. Since the number of items that can be stored on the list is limited, the list might eventually become full in a call to Add.
   1. You must write code in the implementation of the Add procedure that verifies this condition.
   2. If the procedure detects that the list is full, it must display the following message: "ERROR: list is full!".

6. Procedure Display is used to display all to-do items.
   1. The header (first line) must be TO-DO LIST.
   2. It must display one item per line.

Remarks:

1. We use access types and unconstrained arrays in the implementation of the Todo_Lists package.

Listing 7: todo_lists.ads

```
package Todo_Lists is

   -- Replace by actual type declaration
   type Todo_Item is null record;

   -- Replace by actual type declaration
   type Todo_Items is null record;

   -- Replace by actual type declaration
   type Todo_List is null record;

   procedure Add (Todos : in out Todo_List;
                  Item : String);

   procedure Display (Todos : Todo_List);

end Todo_Lists;
```

Listing 8: todo_lists.adb

```
with Ada.Text_IO; use Ada.Text_IO;

package body Todo_Lists is

   procedure Add (Todos : in out Todo_List;
                  Item : String) is
   begin
      Put_Line ("ERROR: list is full!");
   end Add;

   procedure Display (Todos : Todo_List) is
   begin
      null;
   end Display;
```

(continues on next page)
end Todo_Lists;

Listing 9: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is
type Test_Case_Index is (Todo_List_Chk);

procedure Check (TC : Test_Case_Index) is
  T : Todo_List (10);
begin
  case TC is
  when Todo_List_Chk =>
    Add (T, "Buy milk");
    Add (T, "Buy tea");
    Add (T, "Buy present");
    Add (T, "Buy tickets");
    Add (T, "Pay electricity bill");
    Add (T, "Schedule dentist appointment");
    Add (T, "Call sister");
    Add (T, "Revise spreadsheet");
    Add (T, "Edit entry page");
    Add (T, "Select new design");
    Add (T, "Create upgrade plan");
    Display (T);
  end case;
end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

107.4 Price list

Goal: implement a list containing prices

Steps:
1. Implement the Price_Lists package.
   1. Declare the Price_Type type.
   2. Declare the Price_List record.
3. Implement the Reset procedure.
4. Implement the Add procedure.
Learning Ada

5. Implement the Get function.
6. Implement the Display procedure.

Requirements:
1. Price_Type is a decimal fixed-point data type with a delta of two digits (e.g. 0.01) and twelve digits in total.
2. Price_List is a record type that contains the price list.
   1. This record type must have a discriminant for the maximum number of elements of the list.
3. Procedure Reset resets the list.
4. Procedure Add adds a price to the list.
   1. You should keep track of the last element added to the list.
5. Function Get retrieves a price from the list using an index.
   1. This function returns a record instance of Price_Result type.
   2. Price_Result is a variant record containing:
      1. the Boolean component Ok, and
      2. the component Price (of Price_Type).
   3. The returned value of Price_Result type is one of the following:
      1. If the index specified in a call to Get contains a valid (initialized) price, then
         • 0k is set to True, and
         • the Price component contains the price for that index.
      2. Otherwise:
         • 0k is set to False, and
         • the Price component is not available.
6. Procedure Display shows all prices from the list.
   1. The header (first line) must be PRICE LIST.
   2. The remaining lines contain one price per line.
   3. For example:
      • For the following code:

        ```ada
        procedure Test is
          L : Price_List (10);
        begin
          Reset (L);
          Add (L, 1.45);
          Add (L, 2.37);
          Display (L);
        end Test;
        
        The output is:
        
        PRICE LIST
        1.45
        2.37
        ```

Remarks:
1. To implement the package, you'll use the following features of the Ada language:
1. decimal fixed-point types;
2. records with discriminants;
3. dynamically-sized record types;
4. variant records.

2. For record type Price_List, you may use an unconstrained array as a component of the record and use the discriminant in the component declaration.

Listing 10: price_lists.ads

```ada
package Price_Lists is
  -- Replace by actual type declaration
  type Price_Type is new Float;

  -- Replace by actual type declaration
  type Price_List is null record;

  -- Replace by actual type declaration
  type Price_Result is null record;

  procedure Reset (Prices : in out Price_List);
  procedure Add (Prices : in out Price_List;
                 Item     : Price_Type);

  function Get (Prices : Price_List;
                Idx     : Positive) return Price_Result;

  procedure Display (Prices : Price_List);
end Price_Lists;
```

Listing 11: price_lists.adb

```ada
package body Price_Lists is

  procedure Reset (Prices : in out Price_List) is
  begin
    null;
  end Reset;

  procedure Add (Prices : in out Price_List;
                 Item     : Price_Type) is
  begin
    null;
  end Add;

  function Get (Prices : Price_List;
                Idx     : Positive) return Price_Result is
  begin
    null;
  end Get;

  procedure Display (Prices : Price_List) is
  begin
    null;
  end Display;
end Price_Lists;
```
Listing 12: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Price_Lists; use Price_Lists;

procedure Main is
  type Test_Case_Index is
    (Price_Type_Chk, Price_List_Chk, Price_List_Get_Chk);

  procedure Check (TC : Test_Case_Index) is
    L : Price_List (10);

  procedure Local_Init_List is
    begin
      Reset (L);
      Add (L, 1.45);
      Add (L, 2.37);
      Add (L, 3.21);
      Add (L, 4.14);
      Add (L, 5.22);
      Add (L, 6.69);
      Add (L, 7.77);
      Add (L, 8.14);
      Add (L, 9.99);
      Add (L, 10.01);
    end Local_Init_List;

  procedure Get_Display (Idx : Positive) is
    R : constant Price_Result := Get (L, Idx);
    begin
      Put_Line ("Attempt Get # " & Positive'Image (Idx));
      if R.Ok then
        Put_Line ("Element # " & Positive'Image (Idx)
                   & " => " & Price_Type'Image (R.Price));
      else
        declare
          begin
            Put_Line ("Element # " & Positive'Image (Idx)
                       & " => " & Price_Type'Image (R.Price));
        exception
          when others =>
            Put_Line ("Element not available (as expected)" );
        end;
      end if;
    end Get_Display;

    begin
      case TC is
        when Price_Type_Chk =>
          Put_Line ("The delta value of Price_Type is 
                     & Price_Type'Image (Price_Type'Delta) & ",");
          Put_Line ("The minimum value of Price_Type is 
                     & Price_Type'Image (Price_Type'First) & ",");
          Put_Line ("The maximum value of Price_Type is 
                     & Price_Type'Image (Price_Type'Last) & ",");
        when Price_List_Chk =>
          
          (continues on next page)
    ```
60      Local_Init_List;
61      Display (L);
62      when Price_List_Get_CHK =>
63          Local_Init_List;
64      Get_Display (5);
65      Get_Display (40);
66      end case;
67      end Check;
68  
69  begin
70      if Argument_Count < 1 then
71          Put_Line ("ERROR: missing arguments! Exiting...");
72          return;
73      elsif Argument_Count > 1 then
74          Put_Line ("Ignoring additional arguments...");
75      end if;
76      Check (Test_Case_Index'Value (Argument (1)));
77  end Main;
108.1 Directions

Goal: create a package that handles directions and geometric angles using a previous implementation.

Steps:
1. Fix the implementation of the Test_Directions procedure.

Requirements:
1. The implementation of the Test_Directions procedure must compile correctly.

Remarks:
1. This exercise is based on the Directions exercise from the Records (page 1835) labs.
   1. In this version, however, Ext_Angle is a private type.
2. In the implementation of the Test_Directions procedure below, the Ada developer tried to initialize All_Directions — an array of Ext_Angle type — with aggregates.
   1. Since we now have a private type, the compiler complains about this initialization.
3. To fix the implementation of the Test_Directions procedure, you should use the appropriate function from the Directions package.
4. The initialization of All_Directions in the code below contains a consistency error where the angle doesn't match the assessed direction.
   1. See if you can spot this error!
   2. This kind of errors can happen when record components that have correlated information are initialized individually without consistency checks — using private types helps to avoid the problem by requiring initialization routines that can enforce consistency.

Listing 1: directions.ads

```ada
package Directions is

    type Angle_Mod is mod 360;

    type Direction is
    (North,
     Northwest,
     West,
     Southwest,
     South,
     Southeast,
     East);
```

(continues on next page)
function To_Direction (N : Angle_Mod) return Direction;

type Ext_Angle is private;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;

procedure Display (N : Ext_Angle);

private

type Ext_Angle is record
  Angle_Elem   : Angle_Mod;
  Direction_Elem : Direction;
end record;

end Directions;

Listing 2: directions.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Directions is

  procedure Display (N : Ext_Angle) is
  begin
    Put_Line ("Angle: "
      & Angle_Mod'Image (N.Angle_Elem)
      & " => "
      & Direction'Image (N.Direction_Elem)
      & ".");
  end Display;

  function To_Direction (N : Angle_Mod) return Direction is
  begin
    case N is
      when 0 => return East;
      when 1 .. 89 => return Northwest;
      when 90 => return North;
      when 91 .. 179 => return Northwest;
      when 180 => return West;
      when 181 .. 269 => return Southwest;
      when 270 => return South;
      when 271 .. 359 => return Southeast;
    end case;
  end To_Direction;

  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
  begin
    return (Angle_Elem  => N,
        Direction_Elem => To_Direction (N));
  end To_Ext_Angle;

end Directions;

Listing 3: test_directions.adb

with Directions; use Directions;

procedure Test_Directions is
type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;

All_Directions : constant Ext_Angle_Array (1 .. 6) := ((0, East),
(45, Northwest),
(90, North),
(91, North),
(180, West),
(270, South));

begin
   for I in All_Directions'Range loop
      Display (All_Directions (I));
   end loop;
end Test_Directions;

Listing 4: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Test_Directions;

procedure Main is
   type Test_Case_Index is (Direction_Chk);

   procedure Check (TC : Test_Case_Index) is
   begin
      case TC is
         when Direction_Chk =>
            Test_Directions;
      end case;
   end Check;

begin
   if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
   elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
   end if;

   Check (Test_Case_Index'Value (Argument (1)));
end Main;

108.2 Limited Strings

Goal: work with limited private types.

Steps:
1. Implement the Limited_Strings package.
   1. Implement the Copy function.
   2. Implement the = operator.

Requirements:
1. For both Copy and =, the two parameters may refer to strings with different lengths. We'll limit the implementation to just take the minimum length:

   1. In case of copying the string "Hello World" to a string with 5 characters, the copied string is "Hello":

   ```
   S1 : constant Lim_String := Init ("Hello World");
   S2 : Lim_String := Init (5);
   begin
   Copy (From => S1, To => S2);
   Put_Line (S2); -- This displays "Hello".
   ```

   2. When comparing "Hello World" to "Hello", the = operator indicates that these strings are equivalent:

   ```
   S1 : constant Lim_String := Init ("Hello World");
   S2 : constant Lim_String := Init ("Hello");
   begin
   if S1 = S2 then
   -- True => This branch gets selected.
   ```

   2. When copying from a short string to a longer string, the remaining characters of the longer string must be initialized with underscores (_). For example:

   ```
   S1 : constant Lim_String := Init ("Hello");
   S2 : Lim_String := Init (10);
   begin
   Copy (From => S1, To => S2);
   Put_Line (S2); -- This displays "Hello____".
   ```

Remarks:

1. As we've discussed in the course:

   1. Variables of limited types have the following limitations:

      a. they cannot be assigned to;
      b. they don't have an equality operator (=).

   2. We can, however, define our own, custom subprograms to circumvent these limitations:

      a. In order to copy instances of a limited type, we can define a custom Copy procedure.
      b. In order to compare instances of a limited type, we can define an = operator.

2. You can use the Min_Last constant — which is already declared in the implementation of these subprograms — in the code you write.

3. Some details about the Limited_Strings package:

   1. The Lim_String type acts as a container for strings.

      a. In the private part, Lim_String is declared as an access type to a String.

   2. There are two versions of the Init function that initializes an object of Lim_String type:

      a. The first one takes another string.
      b. The second one receives the number of characters for a string container.

   3. Procedure Put_Line displays object of Lim_String type.

   4. The design and implementation of the Limited_Strings package is very simplistic.
1. A good design would have better handling of access types, for example.

Listing 5: limited_strings.ads

package Limited_Strings is

  type Lim_String is limited private;

  function Init ($ : String) return Lim_String;
  function Init (Max : Positive) return Lim_String;
  procedure Put_Line (LS : Lim_String);
  procedure Copy (From : Lim_String;
                  To : in out Lim_String);
  function "=" (Ref, Dut : Lim_String) return Boolean;

private

  type Lim_String is access String;

end Limited_Strings;

Listing 6: limited_strings.adb

with Ada.Text_IO;

package body Limited_Strings is

  function Init ($ : String) return Lim_String is
  begin
    LS := constant Lim_String := new String'($);
    return LS;
  end Init;

  function Init (Max : Positive) return Lim_String is
  begin
    LS := constant Lim_String := new String'(1 .. Max);
    LS.all := (others => '_');
    return LS;
  end Init;

  procedure Put_Line (LS : Lim_String) is
  begin
    Ada.Text_IO.Put_Line (LS.all);
  end Put_Line;

  function Get_Min_Last (A, B : Lim_String) return Positive is
  begin
    return Positive'Min (A'Last, B'Last);
  end Get_Min_Last;

  procedure Copy (From : Lim_String;
                  To : in out Lim_String) is
  begin
    To := constant Positive := Get_Min_Last (From, To);
    null;
  end;

end Limited_Strings;
function "=" (Ref, Dut : Lim_String) return Boolean is
  Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
begin
  -- Complete the implementation!
  return True;
end;

end Limited_Strings;

Listing 7: check_lim_string.adb

with Ada.Text_IO; use Ada.Text_IO;
with Limited_Strings; use Limited_Strings;

procedure Check_Lim_String is
  S : constant String := "----------";
  S1 : constant Lim_String := Init ("Hello World");
  S2 : constant Lim_String := Init (30);
  S3 : Lim_String := Init (5);
  S4 : Lim_String := Init (S & S & S);
begin
  Put ("S1 => ");
  Put_Line (S1);
  Put ("S2 => ");
  Put_Line (S2);
  if S1 = S2 then
    Put_LINE ("S1 is equal to S2.");
  else
    Put_LINE ("S1 isn't equal to S2.");
  end if;
  Copy (From => S1, To => S3);
  Put ("S3 => ");
  Put_Line (S3);
  if S1 = S3 then
    Put_LINE ("S1 is equal to S3.");
  else
    Put_LINE ("S1 isn't equal to S3.");
  end if;
  Copy (From => S1, To => S4);
  Put ("S4 => ");
  Put_LINE (S4);
  if S1 = S4 then
    Put_LINE ("S1 is equal to S4.");
  else
    Put_LINE ("S1 isn't equal to S4.");
  end if;
end Check_Lim_String;

Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

(continues on next page)
with Check_Lim_String;

procedure Main is
    type Test_Case_Index is
        (Lim_String_Chk);
    procedure Check (TC : Test_Case_Index) is
    begin
        case TC is
            when Lim_String_Chk =>
                Check_Lim_String;
        end case;
    end Check;

    begin
        if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
        end if;
        Check (Test_Case_Index'Value (Argument (1)));
    end Main;

108.3 Bonus exercise

In previous labs, we had many source-code snippets containing records that could be declared private. The source-code for the exercise above (Directions) is an example: we've modified the type declaration of Ext_Angle, so that the record is now private. Encapsulating the record components — by declaring record components in the private part — makes the code safer. Also, because many of the code snippets weren't making use of record components directly (but handling record types via the API instead), they continue to work fine after these modifications.

This exercise doesn't contain any source-code. In fact, the goal here is to modify previous labs, so that the record declarations are made private. You can look into those labs, modify the type declarations, and recompile the code. The corresponding test-cases must still pass.

If no other changes are needed apart from changes in the declaration, then that indicates we have used good programming techniques in the original code. On the other hand, if further changes are needed, then you should investigate why this is the case.

Also note that, in some cases, you can move support types into the private part of the specification without affecting its compilation. This is the case, for example, for the People_Array type of the List of Names lab mentioned below. You should, in fact, keep only relevant types and subprograms in the public part and move all support declarations to the private part of the specification whenever possible.

Below, you find the selected labs that you can work on, including changes that you should make. In case you don't have a working version of the source-code of previous labs, you can look into the corresponding solutions.
108.3.1 Colors

Chapter: Records (page 1835)

Steps:
1. Change declaration of RGB type to private.

Requirements:
1. Implementation must compile correctly and test cases must pass.

108.3.2 List of Names

Chapter: Arrays (page 1845)

Steps:
1. Change declaration of Person and People types to limited private.
2. Move type declaration of People_Array to private part.

Requirements:
1. Implementation must compile correctly and test cases must pass.

108.3.3 Price List

Chapter: More About Types (page 1863)

Steps:
1. Change declaration of Price_List type to limited private.

Requirements:
1. Implementation must compile correctly and test cases must pass.
109.1 Display Array

**Goal:** create a generic procedure that displays the elements of an array.

**Steps:**
1. Implement the generic procedure `Display_Array`.

**Requirements:**
1. Generic procedure `Display_Array` displays the elements of an array.
   1. It uses the following scheme:
      • First, it displays a header.
      • Then, it displays the elements of the array.
   2. When displaying the elements, it must:
      • use one line per element, and
      • include the corresponding index of the array.
   3. This is the expected format:

   ```
   <HEADER>
   <index #1>: <element #1>
   <index #2>: <element #2>
   ...
   ```

   4. For example:
      • For the following code:

      ```
      procedure Test is
      A : Int_Array (1 .. 2) := (1, 5);
      begin
      Display_Int_Array ("Elements of A", A);
      end Test;
      ```
      • The output is:

      ```
      Elements of A
      1: 1
      2: 5
      ```

2. These are the formal parameters of the procedure:
   1. a range type `T_Range` for the the array;
   2. a formal type `T_Element` for the elements of the array;
• This type must be declared in such a way that it can be mapped to any type in the instantiation — including record types.

3. an array type T_Array using the T_Range and T_Element types;
4. a function Image that converts a variable of T_Element type to a String.

Listing 1: display_array.ads

```ada
generic
procedure Display_Array (Header : String;
A : T_Array);
```

Listing 2: display_array.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_Array (Header : String;
A : T_Array) is
begin
null;
end Display_Array;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Array;

procedure Main is

    type Test_Case_Index is (Int_Array_Chk, Point_Array_Chk);

    procedure Test_Int_Array is
        type Int_Array is array (Positive range <>) of Integer;

        procedure Display_Int_Array is new
            Display_Array (T_Range => Positive,
                T_Element => Integer,
                T_Array => Int_Array,
                Image => Integer'Image);

        A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
        begin
            Display_Int_Array ("Integers", A);
        end Test_Int_Array;

    procedure Test_Point_Array is
        type Point is record
            X : Float;
            Y : Float;
        end record;

        type Point_Array is array (Natural range <>) of Point;

        function Image (P : Point) return String is
            begin
                return "(" & Float'Image (P.X) & ", " & Float'Image (P.Y) & ")";
            end Image;
```
109.2 Average of Array of Float

Goal: create a generic function that calculates the average of an array of floating-point elements.

Steps:
1. Declare and implement the generic function `Average`.

Requirements:
1. Generic function `Average` calculates the average of an array containing floating-point values of arbitrary precision.
2. Generic function `Average` must contain the following formal parameters:
   1. a range type `T_Range` for the array;
   2. a formal type `T_Element` that can be mapped to floating-point types of arbitrary precision;
   3. an array type `T_Array` using `T_Range` and `T_Element`;

Remarks:
1. You should use the `Float` type for the accumulator.
Listing 4: average.ads

generic
function Average (A : T_Array) return T_Element;

Listing 5: average.adb

function Average (A : T_Array) return T_Element is
begin
  return 0.0;
end Average;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Average;

procedure Main is
  type Test_Case_Index is (Float_Array_Chk, Digits_7_Float_Array_Chk);

  procedure Test_Float_Array is
    type Float_Array is array (Positive range <>) of Float;
    function Average_Float is new
      Average (T_Range => Positive, T_Element => Float, T_Array => Float_Array);

    A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
    begin
      Put_Line ("Average: " & Float'Image (Average_Float (A)));
    end Test_Float_Array;

  procedure Test_Digits_7_Float_Array is
    type Custom_Float is digits 7 range 0.0 .. 1.0;
    type Float_Array is array (Integer range <>) of Custom_Float;
    function Average_Float is new
      Average (T_Range => Integer, T_Element => Custom_Float, T_Array => Float_Array);

    A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
    begin
      Put_Line ("Average: " & Custom_Float'Image (Average_Float (A)));
    end Test_Digits_7_Float_Array;

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Float_Array_Chk =>
        Test_Float_Array;
      when Digits_7_Float_Array_Chk =>
        Test_Digits_7_Float_Array;
    end case;
  end Check;

(continues on next page)
(continued from previous page)

```ada
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
```

### 109.3 Average of Array of Any Type

**Goal:** create a generic function that calculates the average of an array of elements of any arbitrary type.

**Steps:**
1. Declare and implement the generic function `Average`.
2. Implement the test procedure `Test_Item`.
   1. Declare the `F_IO` package.
   2. Implement the `Get_Total` function for the `Item` type.
   3. Implement the `Get_Price` function for the `Item` type.
   4. Declare the `Average_Total` function.
   5. Declare the `Average_Price` function.

**Requirements:**
1. Generic function `Average` calculates the average of an array containing elements of any arbitrary type.
2. Generic function `Average` has the same formal parameters as in the previous exercise, except for:
   1. `T_Element`, which is now a formal type that can be mapped to any arbitrary type.
   2. `To_Float`, which is an *additional* formal parameter.
      - `To_Float` is a function that converts the arbitrary element of `T_Element` type to the `Float` type.
3. Procedure `Test_Item` is used to test the generic `Average` procedure for a record type (`Item`).
   1. Record type `Item` contains the `Quantity` and `Price` components.
4. The following functions have to be implemented to be used for the formal `To_Float` function parameter:
   1. For the `Decimal` type, the function is pretty straightforward: it simply returns the floating-point value converted from the decimal type.
   2. For the `Item` type, two functions must be created to convert to floating-point type:
      1. `Get_Total`, which returns the multiplication of the quantity and the price components of the `Item` type;
      2. `Get_Price`, which returns just the price.
5. The generic function Average must be instantiated as follows:

   1. For the Item type, you must:
      1. declare the Average_Total function (as an instance of Average) using the Get_Total for the To_Float parameter;
      2. declare the Average_Price function (as an instance of Average) using the Get_Price for the To_Float parameter.

6. You must use the Put procedure from Ada.Text_IO.Float_IO.

   1. The generic standard package Ada.Text_IO.Float_IO must be instantiated as F_IO in the test procedures.
   2. This is the specification of the Put procedure, as described in the appendix A.10.9 of the Ada Reference Manual:

   ```
   procedure Put(Item : in Num;
                  Fore : in Field := Default_Fore;
                  Aft  : in Field := Default_Aft;
                  Exp  : in Field := Default_Exp);
   ```

   3. This is the expected format when calling Put from Float_IO:

<table>
<thead>
<tr>
<th>Function</th>
<th>Fore</th>
<th>Aft</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test_Item</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Remarks:

1. In this exercise, you'll abstract the Average function from the previous exercises a step further.

   1. In this case, the function shall be able to calculate the average of any arbitrary type — including arrays containing elements of record types.
   2. Since record types can be composed by many components of different types, we need to provide a way to indicate which component (or components) of the record will be used when calculating the average of the array.
   3. This problem is solved by specifying a To_Float function as a formal parameter, which converts the arbitrary element of T_Element type to the Float type.
   4. In the implementation of the Average function, we use the To_Float function and calculate the average using a floating-point variable.

Listing 7: average.ads

```ada
generic
function Average (A : T_Array) return Float;
```

Listing 8: average.adb

```ada
function Average (A : T_Array) return Float is
begin
   null;
end Average;
```

Listing 9: test_item.ads

```ada
procedure Test_Item;
```
**Learning Ada**

**Listing 10: test_item.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Average;

procedure Test_Item is
  type Amount is delta 0.01 digits 12;
  type Item is record
    Quantity : Natural;
    Price : Amount;
  end record;

  type Item_Array is
    array (Positive range <>) of Item;

  A : constant Item_Array (1 .. 4)
    := ((Quantity => 5, Price => 10.00),
         (Quantity => 80, Price => 2.50),
         (Quantity => 40, Price => 5.00),
         (Quantity => 20, Price => 12.50));

begin
  Put ("Average per item & quantity: ");
  F_IO.Put (Average_Total (A));
  New_Line;
  Put ("Average price: ");
  F_IO.Put (Average_Price (A));
  New_Line;
end Test_Item;
```

**Listing 11: main.adb**

```ada
with Ada.Command_Line; use Ada.Command_Line;

with Ada.Text_IO; use Ada.Text_IO;

with Test_Item;

procedure Main is
  type Test_Case_Index is (Item_Array_Chk);

  procedure Check (TC : Test_Case_Index) is
    begin
      case TC is
        when Item_Array_Chk =>
          Test_Item;
      end case;
    end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1))); end Main;
```

**109.3. Average of Array of Any Type**
109.4 Generic list

Goal: create a system based on a generic list to add and displays elements.

Steps:
1. Declare and implement the generic package Gen_List.
   1. Implement the Init procedure.
   2. Implement the Add procedure.
   3. Implement the Display procedure.

Requirements:
1. Generic package Gen_List must have the following subprograms:
   1. Procedure Init initializes the list.
   2. Procedure Add adds an item to the list.
      1. This procedure must contain a Status output parameter that is set to False when the list was full — i.e. if the procedure failed while trying to add the item;
   3. Procedure Display displays the complete list.
      1. This includes the name of the list and its elements — using one line per element.
      2. This is the expected format:

```
<NAME>
<element #1>
<element #2>
...
```

2. Generic package Gen_List has these formal parameters:
   1. an arbitrary formal type Item;
   2. an unconstrained array type Items of Item element with positive range;
   3. the Name parameter containing the name of the list;
      • This must be a formal input object of String type.
      • It must be used in the Display procedure.
   4. an actual array List_Array to store the list;
      • This must be a formal in out object of Items type.
   5. the variable Last to store the index of the last element;
      • This must be a formal in out object of Natural type.
   6. a procedure Put for the Item type.
      • This procedure is used in the Display procedure to display individual elements of the list.

3. The test procedure Test_Int is used to test a list of elements of Integer type.
4. For both test procedures, you must:
   1. add missing type declarations;
   2. declare and implement a Put procedure for individual elements of the list;
   3. declare instances of the Gen_List package.
      • For the Test_Int procedure, declare the Int_List package.
Remarks:

1. In previous labs, you've been implementing lists for a variety of types.
   - The List of Names exercise from the Arrays (page 1845) labs is an example.
   - In this exercise, you have to abstract those implementations to create the generic Gen_List package.

Listing 12: gen_list.ads

generic
package Gen_List is

procedure Init;

procedure Add (I : Item;
    Status : out Boolean);

procedure Display;

end Gen_List;

Listing 13: gen_list.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Gen_List is

procedure Init is
begin
null;
end Init;

procedure Add (I : Item;
    Status : out Boolean) is
begin
null;
end Add;

procedure Display is
begin
null;
end Display;

end Gen_List;

Listing 14: test_int.ads

procedure Test_Int;

Listing 15: test_int.adb

with Ada.Text_IO; use Ada.Text_IO;

with Gen_List;

procedure Test_Int is

type Integer_Array is array (Positive range <>) of Integer;

A : Integer_Array (1 .. 3);

(continues on next page)
L : Natural;
Success : Boolean;

procedure Display_Add_Success (Success : Boolean) is
begin
  if Success then
    Put_Line ("Added item successfully!");
  else
    Put_Line ("Couldn't add item!");
  end if;
end Display_Add_Success;

begin
  Int_List.Init;

  Int_List.Add (2, Success);
  Display_Add_Success (Success);

  Int_List.Add (5, Success);
  Display_Add_Success (Success);

  Int_List.Add (7, Success);
  Display_Add_Success (Success);

  Int_List.Add (8, Success);
  Display_Add_Success (Success);

  Int_List.Display;
end Test_Int;

Listing 16: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Test_Int;

procedure Main is
type Test_Case_Index is (Int_chk);

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Int_check =>
      Test_Int;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'value (Argument (1)));
end Main;
110.1 Uninitialized Value

**Goal:** implement an enumeration to avoid the use of uninitialized values.

**Steps:**
1. Implement the Options package.
   1. Declare the Option enumeration type.
   2. Declare the Uninitialized_Value exception.
   3. Implement the Image function.

**Requirements:**
1. Enumeration Option contains:
   1. the Uninitialized value, and
   2. the actual options:
      • Option_1,
      • Option_2,
      • Option_3.
2. Function Image returns a string for the Option type.
   1. In case the argument to Image is Uninitialized, the function must raise the Uninitialized_Value exception.

**Remarks:**
1. In this exercise, we employ exceptions as a mechanism to avoid the use of uninitialized values for a certain type.

Listing 1: options.ads

```ada
package Options is
   -- Declare the Option enumeration type!
   type Option is null record;
   function Image (O : Option) return String;
end Options;
```

Listing 2: options.adb

```ada
package body Options is
   function Image (O : Option) return String is
```

(continues on next page)
begin
    return "";
end Image;

end Options;

Listing 3: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Options; use Options;

procedure Main is
  type Test_Case_Index is
    (Options_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check (O : Option) is
      begin
        Put_Line (Image (O));
        exception
          when E : Uninitialized_Value =>
            Put_Line (Exception_Message (E));
      end Check;
    begin
      case TC is
        when Options_Chk =>
          for O in Option loop
            Check (O);
          end loop;
        end case;
      end Check;
    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;

110.2 Numerical Exception

Goal: handle numerical exceptions in a test procedure.

Steps:

1. Add exception handling to the Check_Exception procedure.

Requirements:

1. The test procedure Num_Exception_Test from the Tests package below must be used in the implementation of Check_Exception.
2. The Check_Exception procedure must be extended to handle exceptions as follows:

1. If the exception raised by Num_Exception_Test is Constraint_Error, the procedure must display the message "Constraint_Error detected!" to the user.
2. Otherwise, it must display the message associated with the exception.

Remarks:
1. You can use the Exception_Message function to retrieve the message associated with an exception.

Listing 4: tests.ads

```ada
package Tests is

  type Test_ID is (Test_1, Test_2);

  Custom_Exception : exception;

  procedure Num_Exception_Test (ID : Test_ID);

end Tests;
```

Listing 5: tests.adb

```ada
package body Tests is

  pragma Warnings (Off, "variable ""C"" is assigned but never read");

  procedure Num_Exception_Test (ID : Test_ID) is
    A, B, C : Integer;
  begin
    case ID is
      when Test_1 =>
        A := Integer'Last;
        B := Integer'Last;
        C := A + B;
      when Test_2 =>
        raise Custom_Exception with "Custom_Exception raised!";
    end case;
  end Num_Exception_Test;

  pragma Warnings (On, "variable ""C"" is assigned but never read");

end Tests;
```

Listing 6: check_exception.adb

```ada
with Tests; use Tests;

procedure Check_Exception (ID : Test_ID) is
  begin
    Num_Exception_Test (ID);
  end Check_Exception;
```

Listing 7: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;

(continues on next page)```
with Check_Exception;

procedure Main is
  type Test_Case_Index is
    (Exception_1_Chk, Exception_2_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Handle_Exception (ID : Test_ID) is
      begin
        Check_Exception (ID);
        exception
          when Constraint_Error =>
            Put_Line ("Constraint_Error" 
                       & " (raised by Check_Exception) detected!");
          when E : others =>
            Put_Line (Exception_Name (E) 
                       & " (raised by Check_Exception) detected!");
        end Check_Handle_Exception;

      begin
        case TC is
          when Exception_1_Chk =>
            Check_Handle_Exception (Test_1);
          when Exception_2_Chk =>
            Check_Handle_Exception (Test_2);
          end case;
        end Check;

      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting...");
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;

        Check (Test_Case_Index'Value (Argument (1)));
      end Main;

110.3 Re-raising Exceptions

**Goal:** make use of exception re-raising in a test procedure.

**Steps:**
1. Declare new exception: Another_Exception.
2. Add exception re-raise to the Check_Exception procedure.

**Requirements:**
1. Exception Another_Exception must be declared in the Tests package.
2. Procedure Check_Exception must be extended to re-raise any exception. When an exception is detected, the procedure must:
   1. display a user message (as implemented in the previous exercise), and then
   2. Raise or re-raise exception depending on the exception that is being handled:
1. In case of Constraint_Error exception, re-raise the exception.
2. In all other cases, raise Another_Exception.

**Remarks:**
1. In this exercise, you should extend the implementation of the Check_Exception procedure from the previous exercise.
   1. Naturally, you can use the code for the Check_Exception procedure from the previous exercise as a starting point.

**Listing 8: tests.ads**

```ada
package Tests is

  type Test_ID is (Test_1, Test_2);

  Custom_Exception : exception;

  procedure Num_Exception_Test (ID : Test_ID);

end Tests;
```

**Listing 9: tests.adb**

```ada
package body Tests is

  pragma Warnings (Off, "variable "C" is assigned but never read");

  procedure Num_Exception_Test (ID : Test_ID) is
    A, B, C : Integer;
    begin
      case ID is
        when Test_1 =>
          A := Integer'Last;
          B := Integer'Last;
          C := A + B;
        when Test_2 =>
          raise Custom_Exception with "Custom_Exception raised!";
      end case;
    end Num_Exception_Test;
  pragma Warnings (On, "variable "C" is assigned but never read");

end Tests;
```

**Listing 10: check_exception.ads**

```ada
with Tests; use Tests;

procedure Check_Exception (ID : Test_ID);
```

**Listing 11: check_exception.adb**

```ada
procedure Check_Exception (ID : Test_ID) is
  begin
    Num_Exception_Test (ID);
  end Check_Exception;
```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;
with Check_Exception;

procedure Main is
  type Test_Case_Index is
    (Exception_1_Chk,
     Exception_2_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Handle_Exception (ID : Test_ID) is
      begin
        Check_Exception (ID);
        exception
        when Constraint_Error =>
          Put_Line("Constraint_Error" & " (raised by Check_Exception) detected!");
        when E : others =>
          Put_Line(Exception_Name (E) & " (raised by Check_Exception) detected!");
      end Check_Handle_Exception;
    begin
      case TC is
      when Exception_1_Chk =>
        Check_Handle_Exception (Test_1);
      when Exception_2_Chk =>
        Check_Handle_Exception (Test_2);
      end case;
    end Check;

begin
  if Argument_Count < 1 then
    Put_Line("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
111.1 Display Service

**Goal:** create a simple service that displays messages to the user.

**Steps:**
1. Implement the Display_Services package.
   1. Declare the task type Display_Service.
   2. Implement the Display entry for strings.
   3. Implement the Display entry for integers.

**Requirements:**
1. Task type Display_Service uses the Display entry to display messages to the user.
2. There are two versions of the Display entry:
   1. One that receives messages as a string parameter.
   2. One that receives messages as an Integer parameter.
3. When a message is received via a Display entry, it must be displayed immediately to the user.

Listing 1: display_services.ads
```
package Display_Services is
end Display_Services;
```

Listing 2: display_services.adb
```
package body Display_Services is
end Display_Services;
```

Listing 3: main.adb
```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Services; use Display_Services;

procedure Main is
  type Test_Case_Index is (Display_Service_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    -- Main body...
  end Check;
```

(continues on next page)
begin TC is
    when Display_Service_Chk =>
        Display.Display ("Hello");
        delay 0.5;
        Display.Display ("Hello again");
        delay 0.5;
        Display.Display (55);
        delay 0.5;
    end case;
end Check;
begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
end Main;

111.2 Event Manager

Goal: implement a simple event manager.

Steps:
1. Implement the Event_Managers package.
   1. Declare the task type Event_Manager.
   2. Implement the Start entry.
   3. Implement the Event entry.

Requirements:
1. The event manager has a similar behavior as an alarm
   1. The sole purpose of this event manager is to display the event ID at the correct time.
   2. After the event ID is displayed, the task must finish.
2. The event manager (Event_Manager type) must have two entries:
   1. Start, which starts the event manager with an event ID;
   2. Event, which delays the task until a certain time and then displays the event ID as a user message.
3. The format of the user message displayed by the event manager is Event #<event_id>.
   1. You should use Natural’Image to display the ID (as indicated in the body of the Event_Managers package below).

Remarks:
1. In the Start entry, you can use the Natural type for the ID.
2. In the Event entry, you should use the Time type from the Ada.Real_Time package for the time parameter.

3. Note that the test application below creates an array of event managers with different delays.

### Listing 4: event_managers.ads

```ada
package Event_Managers is
end Event_Managers;
```

### Listing 5: event_managers.adb

```ada
package body Event_Managers is
   -- Don't forget to display the event ID:
   -- Put_Line("Event #" & Natural'Image (Event_ID));
end Event_Managers;
```

### Listing 6: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Event_Managers; use Event_Managers;
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
type Test_Case_Index is (Event_Manager_Chk);

   procedure Check (TC : Test_Case_Index) is
   begin
      Ev_Mng : array (1 .. 5) of Event_Manager;
      begin
         case TC is
            when Event_Manager_Chk =>
               for I in Ev_Mng'Range loop
                  Ev_Mng (I).Start (I);
               end loop;
               Ev_Mng (1).Event (Clock + Seconds (5));
               Ev_Mng (2).Event (Clock + Seconds (3));
               Ev_Mng (3).Event (Clock + Seconds (1));
               Ev_Mng (4).Event (Clock + Seconds (2));
               Ev_Mng (5).Event (Clock + Seconds (4));
         end case;
      end Check;
      begin
         if Argument_Count < 1 then
            Put_Line("ERROR: missing arguments! Exiting...");
            return;
         elsif Argument_Count > 1 then
            Put_Line("Ignoring additional arguments...");
         end if;
         Check (Test_Case_Index'Value (Argument (1)));
      end Main;
```

111.2. Event Manager
111.3 Generic Protected Queue

**Goal:** create a queue container using a protected type.

**Steps:**
1. Implement the generic package Gen_Queues.
   1. Declare the protected type Queue.
   2. Implement the Empty function.
   3. Implement the Full function.
   4. Implement the Push entry.
   5. Implement the Pop entry.

**Requirements:**
1. These are the formal parameters for the generic package Gen_Queues:
   1. a formal modular type;
      - This modular type should be used by the Queue to declare an array that stores the elements of the queue.
      - The modulus of the modular type must correspond to the maximum number of elements of the queue.
   2. the data type of the elements of the queue.
      - Select a formal parameter that allows you to store elements of any data type in the queue.

2. These are the operations of the Queue type:
   1. Function Empty indicates whether the queue is empty.
   2. Function Full indicates whether the queue is full.
   3. Entry Push stores an element in the queue.
   4. Entry Pop removes an element from the queue and returns the element via output parameter.

**Remarks:**
1. In this exercise, we create a queue container by declaring and implementing a protected type (Queue) as part of a generic package (Gen_Queues).
2. As a bonus exercise, you can analyze the body of the Queue_Tests package and understand how the Queue type is used there.
   1. In particular, the procedure Concurrent_Test implements two tasks: T_Producer and T_Consumer. They make use of the queue concurrently.

Listing 7: gen_queues.ads
```ada
package Gen_Queues is
end Gen_Queues;
```

Listing 8: gen_queues.adb
```ada
package body Gen_Queues is
end Gen_Queues;
```
Listing 9: queue_tests.ads

```ada
package Queue_Tests is

   procedure Simple_Test;
   procedure Concurrent_Test;

end Queue_Tests;
```

Listing 10: queue_tests.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Gen_Queues;

package body Queue_Tests is

   Max : constant := 10;
   type Queue_Mod is mod Max;

   procedure Simple_Test is
      package Queues_Float is new Gen_Queues (Queue_Mod, Float);
      Q_F : Queues_Float.Queue;
      V : Float;
      begin
         V := 10.0;
         while not Q_F.Full loop
            Q_F.Push (V);
            V := V + 1.5;
         end loop;

         while not Q_F.Empty loop
            Q_F.Pop (V);
            Put_Line ("Value from queue: " & Float'Image (V));
         end loop;
      end Simple_Test;

   procedure Concurrent_Test is
      package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
      Q_I : Queues_Integer.Queue;
      task T_Producer;
      task T_Consumer;

      task body T_Producer is
         V : Integer := 100;
      begin
         for I in 1 .. 2 * Max loop
            Q_I.Push (V);
            V := V + 1;
         end loop;
      end T_Producer;

      task body T_Consumer is
         V : Integer;
      begin
         delay 1.5;
      end T_Consumer;
```

(continues on next page)
Listing 11: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Tests; use Queue_Tests;

procedure Main is
  type Test_Case_Index is (Simple_Queue_Chk,
                            Concurrent_Queue_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
    when Simple_Queue_Chk =>
      Simple_Test;
    when Concurrent_Queue_Chk =>
      Concurrent_Test;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
```
112.1 Price Range

**Goal:** use predicates to indicate the correct range of prices.

**Steps:**
1. Complete the Prices package.
   1. Rewrite the type declaration of Price.

**Requirements:**
1. Type Price must use a predicate instead of a range.

**Remarks:**
1. As discussed in the course, ranges are a form of contract.
   1. For example, the subtype Price below indicates that a value of this subtype must always be positive:

   ```
   subtype Price is Amount range 0.0 .. Amount'Last;
   ```

   2. Interestingly, you can replace ranges by predicates, which is the goal of this exercise.

Listing 1: prices.ads

```ada
package Prices is

   type Amount is delta 10.0 ** (-2) digits 12;

   subtype Price is Amount range 0.0 .. Amount'Last;

end Prices;
```

Listing 2: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Prices; use Prices;

procedure Main is

   type Test_Case_Index is
      (Price_Range_Chk);
```

(continues on next page)
procedure Check (TC : Test_Case_Index) is

procedure Check_Range (A : Amount) is
  P : constant Price := A;
begin
  Put_Line ("Price: " & Price'Image (P));
  end Check_Range;

begin
  case TC is
  when Price_Range_Chk =>
    Check_Range (-2.0);
  end case;
exception
  when Constraint_Error =>
    Put_Line ("Constraint_Error detected (NOT as expected)."/
  when Assert_Failure =>
    Put_Line ("Assert_Failure detected (as expected)."/nend Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting..."/
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments..."
  end if;

  Check (Test_Case_Index'Value (Argument (1)));
end Main/

112.2 Pythagorean Theorem: Predicate

Goal: use the Pythagorean theorem as a predicate.

Steps:
1. Complete the Triangles package.
   1. Add a predicate to the Right_Triangle type.

Requirements:
1. The Right_Triangle type must use the Pythagorean theorem as a predicate to ensure that its components are consistent.

Remarks:
1. As you probably remember, the Pythagoras' theorem\(^{383}\) states that the square of the hypotenuse of a right triangle is equal to the sum of the squares of the other two sides.

Listing 3: triangles.ads

package Triangles is

  subtype Length is Integer;

type Right_Triangle is record
    H  : Length := 0;

\(^{383}\) https://en.wikipedia.org/wiki/Pythagorean_theorem
-- Hypotenuse
C1, C2 : Length := 0;
-- Catheti / legs
end record;

function Init (H, C1, C2 : Length) return Right_Triangle is
  ((H, C1, C2));
end Triangles;

Listing 4: triangles-io.ads
package Triangles.IO is
  function Image (T : Right_Triangle) return String;
end Triangles.IO;

Listing 5: triangles-io.adb
package body Triangles.IO is
  function Image (T : Right_Triangle) return String is
    "(" & Length'Image (T.H)
    & ", " & Length'Image (T.C1)
    & ", " & Length'Image (T.C2)
    & ")";
end Triangles.IO;

Listing 6: main.adb
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is
  type Test_Case_Index is
    (Triangle_8_6_Pass_Chk, 
     Triangle_8_6_Fail_Chk, 
     Triangle_10_24_Pass_Chk, 
     Triangle_10_24_Fail_Chk, 
     Triangle_18_24_Pass_Chk, 
     Triangle_18_24_Fail_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
      exception
        when Constraint_Error =>
          Put_Line ("Constraint_Error detected (NOT as expected)."odable.
        when Assert_Failure =>
          Put_Line ("Assert_Failure detected (NOT as expected)."
);
112.3 Pythagorean Theorem: Precondition

Goal: use the Pythagorean theorem as a precondition.

Steps:
1. Complete the Triangles package.
   1. Add a precondition to the Init function.

Requirements:
1. The Init function must use the Pythagorean theorem as a precondition to ensure that the input values are consistent.

Remarks:
1. In this exercise, you'll work again with the Right_Triangle type.
   1. This time, your job is to use a precondition instead of a predicate.
   2. The precondition is applied to the Init function, not to the Right_Triangle type.

Listing 7: triangles.ads

```ada
package Triangles is
  subtype Length is Integer;
  type Right_Triangle is record
    H : Length := 0;  -- Hypotenuse
    C1, C2 : Length := 0;  -- Catheti / legs
  end record;
end Triangles;
```
function Init (H, C1, C2 : Length) return Right_Triangle is
    ((H, C1, C2));
end Triangles;

Listing 8: triangles-io.ads

package Triangles.IO is
    function Image (T : Right_Triangle) return String;
end Triangles.IO;

Listing 9: triangles-io.adb

package body Triangles.IO is
    function Image (T : Right_Triangle) return String is
        ("(" & Length'Image (T.H) & ", " & Length'Image (T.C1) & ", " & Length'Image (T.C2) & ")");
end Triangles.IO;

Listing 10: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

    type Test_Case_Index is
        (Triangle_8_6_Pass_Chk,
         Triangle_8_6_Fail_Chk,
         Triangle_10_24_Pass_Chk,
         Triangle_10_24_Fail_Chk,
         Triangle_18_24_Pass_Chk,
         Triangle_18_24_Fail_Chk);

    procedure Check (TC : Test_Case_Index) is
        procedure Check_Triangle (H, C1, C2 : Length) is
            T : Right_Triangle;
        begin
            T := Init (H, C1, C2);
            Put_Line (Image (T));
        exception
            when Constraint_Error =>
                Put_Line ("Constraint_Error detected (NOT as expected).");
            when Assert_Failure =>
                Put_Line ("Assert_Failure detected (as expected).");
        end Check_Triangle;
    begin
        case TC is
            (continues on next page)
Learning Ada

(continued from previous page)

```ada
when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'Value (Argument (1)));
end Main;
```

112.4 Pythagorean Theorem: Postcondition

**Goal:** use the Pythagorean theorem as a postcondition.

**Steps:**

1. Complete the Triangles package.
   1. Add a postcondition to the Init function.

**Requirements:**

1. The Init function must use the Pythagorean theorem as a postcondition to ensure that the returned object is consistent.

**Remarks:**

1. In this exercise, you'll work again with the Triangles package.
   1. This time, your job is to apply a postcondition instead of a precondition to the Init function.

Listing 11: triangles.ads

```ada
package Triangles is

  subtype Length is Integer;

  type Right_Triangle is record
    H : Length := 0;
    -- Hypotenuse
    C1, C2 : Length := 0;
    -- Catheti / legs
  end record;

  function Init (H, C1, C2 : Length) return Right_Triangle is
    ((H, C1, C2));
end Triangles;
```

1910 Chapter 112. Design by contracts
Listing 12: triangles-io.ads

```ada
package Triangles.IO is

  function Image (T : Right_Triangle) return String;

end Triangles.IO;
```

Listing 13: triangles-io.adb

```ada
package body Triangles.IO is

  function Image (T : Right_Triangle) return String is
    ("(" & Length'Image (T.H) & ", " & Length'Image (T.C1) & ", " & Length'Image (T.C2) & ")");

end Triangles.IO;
```

Listing 14: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;

with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

  type Test_Case_Index is
    (Triangle_8_6_Pass_Chk, Triangle_8_6_Fail_Chk, Triangle_10_24_Pass_Chk, Triangle_10_24_Fail_Chk, Triangle_18_24_Pass_Chk, Triangle_18_24_Fail_Chk);

  procedure Check (TC : Test_Case_Index) is

    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
      exception
        when Constraint_Error =>
          Put_Line ("Constraint_Error detected (NOT as expected)."/MPL);
        when Assert_Failure =>
          Put_Line ("Assert_Failure detected (as expected)."/MPL);
      end Check_Triangle;
      begin
        case TC is
          when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
          when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
          when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
          when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
          when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
          when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
          (continues on next page)
      end case;
    end Check_Triangle;
  end Check;
```

112.4. Pythagorean Theorem: Postcondition
end case;
end Check;

begin
if Argument_Count < 1 then
  Put_Line ("ERROR: missing arguments! Exiting...");
  return;
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
-- Hypotenuse
C1, C2 : Length := 0;
-- Catheti / legs

end record;

function Init (H, C1, C2 : Length) return Right_Triangle is
  ((H, C1, C2));
end Triangles;

Listing 16: triangles-io.ads

package Triangles.IO is
  function Image (T : Right_Triangle) return String;
end Triangles.IO;

Listing 17: triangles-io.adb

package body Triangles.IO is

  function Image (T : Right_Triangle) return String is
    "(" & Length’Image (T.H)
    & "", " & Length’Image (T.C1)
    & ", " & Length’Image (T.C2)
    & ")";
end Triangles.IO;

Listing 18: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

  type Test_Case_Index is
    (Triangle 8 6 Pass_Chk,
     Triangle 8 6 Fail_Chk,
     Triangle 10 24 Pass_Chk,
     Triangle 10 24 Fail_Chk,
     Triangle 18 24 Pass_Chk,
     Triangle 18 24 Fail_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
    begin
      T := Init (H, C1, C2);
      Put_Line (Image (T));
    exception
      when Constraint_Error =>
        Put_Line ("Constraint_Error detected (NOT as expected). ");
      when Assert_Failure =>

(continues on next page)
112.6 Primary Color

**Goal:** extend a package for HTML colors so that it can handle primary colors.

**Steps:**

1. Complete the Color_Types package.
   
   1. Declare the HTML_RGB_Color subtype.
   2. Implement the To_Int_Color function.

**Requirements:**

1. The HTML_Color type is an enumeration that contains a list of HTML colors.
2. The To_RGB_Lookup_Table array implements a lookup-table to convert the colors into a hexadecimal value using RGB color components (i.e. Red, Green and Blue).
3. Function To_Int_Color extracts one of the RGB components of an HTML color and returns its hexadecimal value.
   
   1. The function has two parameters:
      
      - First parameter is the HTML color (HTML_Color type).
      - Second parameter indicates which RGB component is to be extracted from the HTML color (HTML_RGB_Color subtype).
   2. For example, if we call To_Int_Color (Salmon, Red), the function returns #FA,
      
      - This is the hexadecimal value of the red component of the Salmon color.
      - You can find further remarks below about this color as an example.
   4. The HTML_RGB_Color subtype is limited to the primary RGB colors components (i.e. Red, Green and Blue).
      
      1. This subtype is used to select the RGB component in calls to To_Int_Color.
2. You must use a predicate in the type declaration.

Remarks:

1. In this exercise, we reuse the code of the Colors: Lookup-Table exercise from the Arrays (page 1845) labs.

2. These are the hexadecimal values of the colors that we used in the original exercise:

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salmon</td>
<td>#FA8072</td>
</tr>
<tr>
<td>Firebrick</td>
<td>#B22222</td>
</tr>
<tr>
<td>Red</td>
<td>#FF0000</td>
</tr>
<tr>
<td>Darkred</td>
<td>#8B0000</td>
</tr>
<tr>
<td>Lime</td>
<td>#00FF00</td>
</tr>
<tr>
<td>Forestgreen</td>
<td>#228B22</td>
</tr>
<tr>
<td>Green</td>
<td>#008000</td>
</tr>
<tr>
<td>Darkgreen</td>
<td>#006400</td>
</tr>
<tr>
<td>Blue</td>
<td>#0000FF</td>
</tr>
<tr>
<td>Mediumblue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>Darkblue</td>
<td>#00008B</td>
</tr>
</tbody>
</table>

3. You can extract the hexadecimal value of each primary color by splitting the values from the table above into three hexadecimal values with two digits each.
   - For example, the hexadecimal value of Salmon is #FA8072, where:
     - the first part of this hexadecimal value (#FA) corresponds to the red component,
     - the second part (#80) corresponds to the green component, and
     - the last part (#72) corresponds to the blue component.

Listing 19: color_types.ads

```ada
package Color_Types is
  type HTML_Color is
    (Salmon, Firebrick, Red, Darkred, Lime, Forestgreen, Green, Darkgreen, Blue, Mediumblue, Darkblue);

  subtype Int_Color is Integer range 0 .. 255;

  function Image (I : Int_Color) return String;

  type RGB is record
    Red : Int_Color;
    Green : Int_Color;
    Blue : Int_Color;
  end record;

  function To_RGB (C : HTML_Color) return RGB;
```

(continues on next page)
function Image (C : RGB) return String;

type HTML_Color_RGB_Array is array (HTML_Color) of RGB;

To_RGB_Lookup_Table : constant HTML_Color_RGB_Array := (Salmon => (16#FA#, 16#80#, 16#72#), Firebrick => (16#B2#, 16#22#, 16#22#), Red => (16#FF#, 16#00#, 16#00#), Darkred => (16#80#, 16#00#, 16#00#), Lime => (16#00#, 16#FF#, 16#00#), Forestgreen => (16#22#, 16#8B#, 16#22#), Green => (16#00#, 16#80#, 16#00#), Darkgreen => (16#00#, 16#64#, 16#00#), Blue => (16#00#, 16#00#, 16#FF#), Mediumblue => (16#00#, 16#00#, 16#CD#), Darkblue => (16#00#, 16#00#, 16#8B#));

subtype HTML_RGB_Color is HTML_Color;

function To_Int_Color (C : HTML_Color; S : HTML_RGB_Color) return Int_Color is
begin
   -- Implement function!
   return 0;
end To_Int_Color;

function Image (I : Int_Color) return String is subtype Str_Range is Integer range 1 .. 10;
S : String (Str_Range);
begin
   Ada.Integer_Text_IO.Put (To => S, Item => I, Base => 16);
   return S;
end Image;

function Image (C : RGB) return String is
begin
   return "(Red => " & Image (C.Red) & ", Green => " & Image (C.Green) & ", Blue => " & Image (C.Blue) & ")";
end Image;

(continues on next page)
end Color_Types;

Listing 21: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is
    (HTML_Color_Red_Chk, HTML_Color_Green_Chk, HTML_Color_Blue_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    procedure Check_HTML_Colors (S : HTML_RGB_Color) is
    begin
      Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
      for I in HTML_Color'Range loop
        Put_Line (HTML_Color'Image (I) & " => " & Image (To_Int_Color (I, S)) & ".");
      end loop;
    end Check_HTML_Colors;

    begin
      case TC is
        when HTML_Color_Red_Chk =>
          Check_HTML_Colors (Red);
        when HTML_Color_Green_Chk =>
          Check_HTML_Colors (Green);
        when HTML_Color_Blue_Chk =>
          Check_HTML_Colors (Blue);
      end case;
      end Check;

    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;
113.1 Simple type extension

Goal: work with type extensions using record types containing numeric components.

Steps:
1. Implement the Type_Extensions package.
   1. Declare the record type T_Float.
   2. Declare the record type T_Mixed
   3. Implement the Init function for the T_Float type with a floating-point input parameter.
   4. Implement the Init function for the T_Float type with an integer input parameter.
   5. Implement the Image function for the T_Float type.
   6. Implement the Init function for the T_Mixed type with a floating-point input parameter.
   7. Implement the Init function for the T_Mixed type with an integer input parameter.
   8. Implement the Image function for the T_Mixed type.

Requirements:
1. Record type T_Float contains the following component:
   1. F, a floating-point type.
2. Record type T_Mixed is derived from the T_Float type.
   1. T_Mixed extends T_Float with the following component:
      1. I, an integer component.
   2. Both components must be numerically synchronized:
      • For example, if the floating-point component contains the value 2.0, the value of the integer component must be 2.
      • In order to simplify the implementation, you can simply use Integer (F) to convert a floating-point variable F to integer.
3. Function Init returns an object of the corresponding type (T_Float or T_Mixed).
   1. For each type, two versions of Init must be declared:
      1. one with a floating-point input parameter,
      2. another with an integer input parameter.
   2. The parameter to Init is used to initialize the record components.
4. Function Image returns a string for the components of the record type.

1. In case of the Image function for the T_Float type, the string must have the format 
   "\{ F => <float value> \}".
   - For example, the call Image (T_Float'(Init (8.0))) should return the string 
     "\{ F => 8.00000E+00 \}".

2. In case of the Image function for the T_Mixed type, the string must have the format 
   "\{ F => <float value>, I => <integer value> \}".
   - For example, the call Image (T_Mixed'(Init (8.0))) should return the string 
     "\{ F => 8.00000E+00, I => 8 \}".

Listing 1: type_extensions.ads

```ada
package Type_Extensions is

   -- Create declaration of T_Float type!
   type T_Float is null record;

   -- function Init ...

   -- function Image ...

   -- Create declaration of T_Mixed type!
   type T_Mixed is null record;

end Type_Extensions;
```

Listing 2: type_extensions.adb

```ada
package body Type_Extensions is

end Type_Extensions;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Type_Extensions; use Type_Extensions;

procedure Main is

   type Test_Case_Index is
      (Type_Extension_Chk);

   procedure Check (TC : Test_Case_Index) is
      F1, F2 : T_Float;
      M1, M2 : T_Mixed;
   begin
      case TC is
         when Type_Extension_Chk =>
            F1 := Init (2.0);
            F2 := Init (3);
            M1 := Init (4.0);
            M2 := Init (5);

            if M2 in T_Float'Class then
               Put_Line ("T_Mixed is in T_Float'Class as expected");
            end if;
      end case;
   end Check;

   Check (Type_Extension_Chk);
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
Put_Line ("F1: " & Image (F1));
Put_Line ("F2: " & Image (F2));
Put_Line ("M1: " & Image (M1));
Put_Line ("M2: " & Image (M2));
end case;
end Check;
begin
if Argument_Count < 1 then
  Put_Line ("ERROR: missing arguments! Exiting...");
  return;
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;
```

113.2 Online Store

**Goal:** create an online store for the members of an association.

**Steps:**

1. Implement the Online_Store package.
   1. Declare the Member type.
   2. Declare the Full_Member type.
   3. Implement the Get_Status function for the Member type.
   4. Implement the Get_Price function for the Member type.
   5. Implement the Get_Status function for the Full_Member type.
   6. Implement the Get_Price function for the Full_Member type.

2. Implement the Online_Store.Tests child package.
   1. Implement the Simple_Test procedure.

**Requirements:**

1. Package Online_Store implements an online store application for the members of an association.
   1. In this association, members can have one of the following status:
      - associate member, or
      - full member.

2. Function Get_Price returns the correct price of an item.
   1. Associate members must pay the full price when they buy items from the online store.
   2. Full members can get a discount.
      1. The discount rate can be different for each full member — depending on factors that are irrelevant for this exercise.

3. Package Online_Store has following types:
   1. Percentage type, which represents a percentage ranging from 0.0 to 1.0.
2. Member type for associate members containing following components:
   - Start, which indicates the starting year of the membership.
     - This information is common for both associate and full members.
     - You can use the Year_Number type from the standard Ada.Calendar package for this component.

3. Full_Member type for full members.
   1. This type must extend the Member type above.
   2. It contains the following additional component:
      - Discount, which indicates the discount rate that the full member gets in the online store.
        - This component must be of Percentage type.

4. For the Member and Full_Member types, you must implement the following functions:
   1. Get_Status, which returns a string with the membership status.
      - The string must be "Associate Member" or "Full Member", respectively.
   2. Get_Price, which returns the adapted price of an item — indicating the actual due amount.
      - For example, for a full member with a 10% discount rate, the actual due amount of an item with a price of 100.00 is 90.00.
      - Associated members don't get a discount, so they always pay the full price.

5. Procedure Simple_Test (from the Online_Store.Tests package) is used for testing.
   1. Based on a list of members that bought on the online store and the corresponding full price of the item, Simple_Test must display information about each member and the actual due amount after discounts.

   2. Information about the members must be displayed in the following format:

   ```plaintext
   Member # <number>
   Status: <status>
   Since: <year>
   Due Amount: <value>
   --------
   ```

   3. For this exercise, Simple_Test must use the following list:

<table>
<thead>
<tr>
<th>#</th>
<th>Membership status</th>
<th>Start (year)</th>
<th>Discount</th>
<th>Full Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Associate</td>
<td>2010</td>
<td>N/A</td>
<td>250.00</td>
</tr>
<tr>
<td>2</td>
<td>Full</td>
<td>1998</td>
<td>10.0 %</td>
<td>160.00</td>
</tr>
<tr>
<td>3</td>
<td>Full</td>
<td>1987</td>
<td>20.0 %</td>
<td>400.00</td>
</tr>
<tr>
<td>4</td>
<td>Associate</td>
<td>2013</td>
<td>N/A</td>
<td>110.00</td>
</tr>
</tbody>
</table>

   4. In order to pass the tests, the information displayed by a call to Simple_Test must conform to the format described above.
      - You can find another example in the remarks below.

**Remarks:**
1. In previous labs, we could have implemented a simplified version of the system described above by simply using an enumeration type to specify the membership status. For example:
Learning Ada

type Member_Status is (Associate_Member, Full_Member);

1. In this case, the Get_Price function would then evaluate the membership status and adapt the item price — assuming a fixed discount rate for all full members. This could be the corresponding function declaration:

```ada
with Ada.Calendar; use Ada.Calendar;
package Online_Store is

  type Amount is delta 10.0**(-2) digits 10;

  subtype Percentage is Amount range 0.0 .. 1.0;

  type Member is null record;

  function Get_Status (M : Member) return String;

  function Get_Price (M : Member; P : Amount) return Amount;

  -- Create declaration of Member type!
  -- You can use Year_Number from Ada.Calendar for the membership starting year.
`
-- Create declaration of Full_Member type!
-- Use the Percentage type for storing the membership discount.

```ada
type Full_Member is null record;

function Get_Status (M : Full_Member) return String;
function Get_Price (M : Full_Member; P : Amount) return Amount;
end Online_Store;
```

Listing 5: online_store.adb

```ada
package body Online_Store is

function Get_Status (M : Member) return String is 
(""");

function Get_Status (M : Full_Member) return String is 
(""");

function Get_Price (M : Member; P : Amount) return Amount is (0.0);

function Get_Price (M : Full_Member; P : Amount) return Amount is 
(0.0);
end Online_Store;
```

Listing 6: online_store-tests.ads

```ada
package Online_Store.Tests is

procedure Simple_Test;
end Online_Store.Tests;
```

Listing 7: online_store-tests.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Online_Store.Tests is

procedure Simple_Test is
    begin
        null;
    end Simple_Test;
end Online_Store.Tests;
```

Listing 8: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Online_Store; use Online_Store;
```

(continues on next page)
with Online_Store.Tests; use Online_Store Tests;

procedure Main is

  type Test_Case_Index is
    (Type_Chk,
     Unit_Test_Chk);

  procedure Check (TC : Test_Case_Index) is

    function Result_Image (Result : Boolean) return String is
      (if Result then "OK" else "not OK");

    begin
      case TC is
        when Type_Chk =>
          declare
            AM : constant Member := (Start => 2002);
            FM : constant Full_Member := (Start => 1990,
                                          Discount => 0.2);

          begin
            Put_Line ("Testing Status of Associate Member Type => "
                      & Result_Image (AM.Get_Status = "Associate Member"));
            Put_Line ("Testing Status of Full Member Type => "
                      & Result_Image (FM.Get_Status = "Full Member"));
            Put_Line ("Testing Discount of Associate Member Price => "
                      & Result_Image (AM.Get_Price (100.0) = 100.0));
            Put_Line ("Testing Discount of Full Member Price => "
                      & Result_Image (FM.Get_Price (100.0) = 80.0));
          end;

        when Unit_Test_Chk =>
          Simple_Test;
      end case;
    end Check;

    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;

      Check (Test_Case_Index'Value (Argument (1)));
    end Main;
114.1 Simple todo list

Goal: implement a simple to-do list system using vectors.

Steps:

1. Implement the Todo_Lists package.
   1. Declare the Todo_Item type.
   2. Declare the Todo_List type.
   3. Implement the Add procedure.
   4. Implement the Display procedure.
2. Todo_Item type is used to store to-do items.
   1. It should be implemented as an access type to strings.
3. Todo_List type is the container for all to-do items.
   1. It should be implemented as a vector.
4. Procedure Add adds items (of Todo_Item type) to the list (of Todo_List type).
   1. This requires allocating a string for the access type.
5. Procedure Display is used to display all to-do items.
   1. It must display one item per line.

Remarks:

1. This exercise is based on the Simple todo list exercise from the More About Types (page 1863).
   1. Your goal is to rewrite that exercise using vectors instead of arrays.
   2. You may reuse the code you've already implemented as a starting point.

Listing 1: todo_lists.ads

```ada
package Todo_Lists is

  type Todo_Item is access String;

  type Todo_List is null record;

  procedure Add (Todos : in out Todo_List;
                 Item   : String);

  procedure Display (Todos : Todo_List);
```

(continues on next page)
Listing 2: todo_lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Todo_Lists is

   procedure Add (Todos : in out Todo_List; Item : String) is
     begin
       null;
     end Add;

   procedure Display (Todos : Todo_List) is
     begin
       Put_Line ("TO-DO LIST");
     end Display;

end Todo_Lists;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is

   type Test_Case_Index is (Todo_List_Chk);

   procedure Check (TC : Test_Case_Index) is
     T : Todo_List;
     begin
       case TC is
         when Todo_List_Chk =>
           Add (T, "Buy milk");
           Add (T, "Buy tea");
           Add (T, "Buy present");
           Add (T, "Buy tickets");
           Add (T, "Pay electricity bill");
           Add (T, "Schedule dentist appointment");
           Add (T, "Call sister");
           Add (T, "Revise spreadsheet");
           Add (T, "Edit entry page");
           Add (T, "Select new design");
           Add (T, "Create upgrade plan");
           Display (T);
         end case;
       end Check;

     begin
       if Argument_Count < 1 then
         Put_Line ("ERROR: missing arguments! Exiting...");
         return;
       elsif Argument_Count > 1 then
         Put_Line ("Ignoring additional arguments...");
       end if;
```
114.2 List of unique integers

**Goal:** create function that removes duplicates from and orders a collection of elements.

**Steps:**
1. Implement package Ops.
   1. Declare the Int_Array type.
   2. Declare the Integer_Sets type.
   3. Implement the Get_Unique function that returns a set.
   4. Implement the Get_Unique function that returns an array of integer values.

**Requirements:**
1. The Int_Array type is an unconstrained array of positive range.
2. The Integer_Sets package is an instantiation of the Ordered_Sets package for the Integer type.
3. The Get_Unique function must remove duplicates from an input array of integer values and order the elements.
   1. For example:
      • if the input array contains \((7, 7, 1)\)
      • the function must return \((1, 7)\).
2. You must implement this function by using sets from the Ordered_Sets package.
3. Get_Unique must be implemented in two versions:
   • one version that returns a set — Set type from the Ordered_Sets package.
   • one version that returns an array of integer values — Int_Array type.

**Remarks:**
1. Sets — as the one found in the generic Ordered_Sets package — are useful for quickly and easily creating an algorithm that removes duplicates from a list of elements.

Listing 4: ops.ads

```ada
with Ada.Containers.Ordered_Sets;

package Ops is

  -- type Int_Array is ...
  -- package Integer_Sets is ...

  subtype Int_Set is Integer_Sets.Set;

  function Get_Unique (A : Int_Array) return Int_Set;

  function Get_Unique (A : Int_Array) return Int_Array;
```

(continues on next page)
end Ops;

Listing 5: ops.adb

package body Ops is
  function Get_Unique (A : Int_Array) return Int_Set is
    begin
      null;
    end Get_Unique;

  function Get_Unique (A : Int_Array) return Int_Array is
    begin
      null;
    end Get_Unique;
end Ops;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ops; use Ops;

procedure Main is
  type Test_Case_Index is
    (Get_Unique_Set_Chk,
     Get_Unique_Array_Chk);

  procedure Check (TC : Test_Case_Index; A : Int_Array) is

    procedure Display_Use_Set (A : Int_Array) is
      S : constant Int_Set := Get_Unique (A);
      begin
        for E of S loop
          Put_Line (Integer'Image (E));
        end loop;
      end Display_Use_Set;

    procedure Display_Use_Array (A : Int_Array) is
      AU : constant Int_Array := Get_Unique (A);
      begin
        for E of AU loop
          Put_Line (Integer'Image (E));
        end loop;
      end Display_Use_Array;

    begin
      case TC is
        when Get_Unique_Set_Chk => Display_Use_Set (A);
        when Get_Unique_Array_Chk => Display_Use_Array (A);
      end case;
    end Check;

    begin
      if Argument_Count < 3 then
        Put_Line ("ERROR: missing arguments! Exiting...");
      end if;
end Main;
return;

else
  declare
    A : Int_Array (1 .. Argument_Count - 1);
  begin
    for I in A'Range loop
      A (I) := Integer'Value (Argument (I + 1));
    end loop;
    Check (Test_Case_Index'Value (Argument (1)), A);
  end;
end if;
end Main;
115.1 Holocene calendar

Goal: create a function that returns the year in the Holocene calendar.

Steps:
1. Implement the To_Holocene_Year function.

Requirements:
1. The To_Holocene_Year function extracts the year from a time object (Time type) and returns the corresponding year for the Holocene calendar.
   1. For positive (AD) years, the Holocene year is calculated by adding 10,000 to the year number.

Remarks:
1. In this exercise, we don't deal with BC years.
2. Note that the year component of the Time type from the Ada.Calendar package is limited to years starting with 1901.

Listing 1: to_holocene_year.adb

```ada
with Ada.Calendar; use Ada.Calendar;

function To_Holocene_Year (T : Time) return Integer is
begin
  return 0;
end To_Holocene_Year;
```

Listing 2: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_I0; use Ada.Text_I0;
with Ada.Calendar; use Ada.Calendar;

with To_Holocene_Year;

procedure Main is
  type Test_Case_Index is (Holocene_CHK);

  procedure Display_Holocene_Year (Y : Year_Number) is
  begin
    HY : Integer;
    begin
```

---

384 https://en.wikipedia.org/wiki/Holocene_calendar
14 \text{HY} := \text{To_Holocene-Year} \text{ (Time Of (Y, 1, 1))};
15 \text{Put_Line} \text{ ("Year (Gregorian): " & Year\text{\_Number}'Image (Y));}
16 \text{Put_Line} \text{ ("Year (Holocene): " & Integer'Image (HY));}
17 \text{end Display_Holocene_Year;}
18 \text{procedure} \text{Check} \text{ (TC : Test\_Case\_Index) is}
19 \begin{case} \text{TC is}
20 \text{when Holocene\_Chk =>}
21 \quad \text{Display_Holocene_Year (2012)};
22 \quad \text{Display_Holocene_Year (2020)};
23 \end{case}
24 \text{end Check;}
25 \begin{if} \text{Argument\_Count < 1 then}
26 \text{Put(Line} \text{ ("ERROR: missing arguments! Exiting..."));}
27 \text{return;}\end{if}
28 \begin{elsif} \text{Argument\_Count > 1 then}
29 \text{Put(Line} \text{ ("Ignoring additional arguments..."));}
30 \text{end if;}
31 \text{Check (Test\_Case\_Index'Value (Argument (1))})
32 \text{end Main;}\end{if}

115.2 List of events

\textbf{Goal:} create a system to manage a list of events.

\textbf{Steps:}

1. Implement the Events package.
   1. Declare the Event\_Item type.
   2. Declare the Event\_Items type.
2. Implement the Events.Lists package.
   1. Declare the Event\_List type.
   2. Implement the Add procedure.
   3. Implement the Display procedure.

\textbf{Requirements:}

1. The Event\_Item type (from the Events package) contains the \textit{description of an event}.
   1. This description shall be stored in an access-to-string type.
2. The Event\_Items type stores a list of events.
   1. This will be used later to represent multiple events for a specific date.
   2. You shall use a vector for this type.
3. The Events.Lists package contains the subprograms that are used in the test application.
4. The Event\_List type (from the Events.Lists package) maps a list of events to a specific date.
   1. You must use the Event\_Items type for the list of events.
2. You shall use the Time type from the Ada.Calendar package for the dates.

3. Since we expect the events to be ordered by the date, you shall use ordered maps for the Event_List type.

5. Procedure Add adds an event into the list of events for a specific date.

6. Procedure Display must display all events for each date (ordered by date) using the following format:

   <event_date #1>
   <description of item #1a>
   <description of item #1b>
   <event_date #2>
   <description of item #2a>
   <description of item #2b>

1. You should use the auxiliary Date_Image function — available in the body of the Events.Lists package — to display the date in the YYYY-MM-DD format.

**Remarks:**

1. Let's briefly illustrate the expected output of this system.

   1. Consider the following example:

   ```ada
   with Ada.Calendar;
   with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
   with Events.Lists; use Events.Lists;
   procedure Test is
     EL : Event_List;
   begin
     EL.Add (Time_Of (2019, 4, 16), "Item #2");
     EL.Add (Time_Of (2019, 4, 15), "Item #1");
     EL.Add (Time_Of (2019, 4, 16), "Item #3");
     EL.Display;
   end Test;
   
   2. The expected output of the Test procedure must be:

   EVENTS LIST
   - 2019-04-15
     - Item #1
   - 2019-04-16
     - Item #2
     - Item #3
   
   Listing 3: events.ads
   ```

   ```ada
   package Events is
   
   type Event_Item is null record;
   
   type Event_Items is null record;
   
   end Events;
   ```
Listing 4: events-lists.ads

```ada
with Ada.Calendar; use Ada.Calendar;

package Events.Lists is

    type Event_List is tagged private;

    procedure Add (Events : in out Event_List;
                   Event_Time :    Time;
                   Event      :    String);

    procedure Display (Events : Event_List);

private

    type Event_List is tagged null record;

end Events.Lists;
```

Listing 5: events-lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

package body Events.Lists is

    procedure Add (Events : in out Event_List;
                   Event_Time :    Time;
                   Event      :    String) is
    begin
        null;
    end Add;

    function Date_Image (T : Time) return String is
        Date_Img : constant String := Image (T);
    begin
        return Date_Img (1 .. 10);
    end;

    procedure Display (Events : Event_List) is
        T : Time;
    begin
        Put_Line ("EVENTS LIST");
        -- You should use Date_Image (T) here!
    end Display;

end Events.Lists;
```

Listing 6: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Events.Lists; use Events.Lists;

procedure Main is
    type Test_Case_Index is
        (Event_List_Chk);
    (continues on next page)
```
procedure Check (TC : Test_Case_Index) is
   EL : Event_List;
begin
   case TC is
      when Event_List_Chk =>
         EL.Add (Time_Of (2018, 2, 16), "Final check");
         EL.Add (Time_Of (2018, 2, 16), "Release");
         EL.Add (Time_Of (2018, 12, 3), "Brother's birthday");
         EL.Add (Time_Of (2018, 1, 1), "New Year's Day");
         EL.Display;
   end case;
   end Check;
begin
   if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
   elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
   end if;
   Check (Test_Case_Index'Value (Argument (1)));
end Main;
116.1 Concatenation

**Goal:** implement functions to concatenate an array of unbounded strings.

**Steps:**
1. Implement the `Str_Concat` package.
   1. Implement the `Concat` function for `Unbounded_String`.
   2. Implement the `Concat` function for `String`.

**Requirements:**
1. The first `Concat` function receives an unconstrained array of unbounded strings and returns the concatenation of those strings as an unbounded string.
   1. The second `Concat` function has the same parameters, but returns a standard string (`String` type).
2. Both `Concat` functions have the following parameters:
   1. An unconstrained array of `Unbounded_String` strings (Unbounded_Strings type).
   2. `Trim_Str`, a Boolean parameter indicating whether each unbounded string must be trimmed.
   3. `Add_Whitespace`, a Boolean parameter indicating whether a whitespace shall be added between each unbounded string and the next one.
      1. No whitespace shall be added after the last string of the array.

**Remarks:**
1. You can use the `Trim` function from the Ada.Strings.Unbounded package.

Listing 1: `str_concat.ads`

```ada

package Str_Concat is

   type Unbounded_Strings is array (Positive range <>) of Unbounded_String;

   function Concat (USA : Unbounded_Strings;
                     Trim_Str : Boolean;
                     Add_Whitespace : Boolean) return Unbounded_String;

   function Concat (USA : Unbounded_Strings;
                     Trim_Str : Boolean;
                     Add_Whitespace : Boolean) return String;
```

(continues on next page)
with Ada.Strings; use Ada.Strings;

package body Str_Concat is

function Concat (USA : Unbounded_Strings;
                 Trim_Str : Boolean;
                 Add_Whitespace : Boolean) return Unbounded_String is
begin
  return "";
end Concat;

function Concat (USA : Unbounded_Strings;
                 Trim_Str : Boolean;
                 Add_Whitespace : Boolean) return String is
begin
  return "";
end Concat;

end Str_Concat;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Str_Concat; use Str_Concat;

procedure Main is
  type Test_Case_Index is
    (Unbounded_Concat_No_Trim_No_WS_Chk,
     Unbounded_Concat_Trim_No_WS_Chk,
     String_Concat_Trim_WS_Chk,
     Concat_Single_Element);

  procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Unbounded_Concat_No_Trim_No_WS_Chk =>
      declare
        S : constant Unbounded_Strings := (
          To_Unbounded_String ("Hello"),
          To_Unbounded_String (" World"),
          To_Unbounded_String ("!"));
      begin
        Put_Line (To_String (Concat (S, False, False)));
      end;
    when Unbounded_Concat_Trim_No_WS_Chk =>
      declare
        S : constant Unbounded_Strings := (
          To_Unbounded_String (" This "),
          To_Unbounded_String (" is "),
          To_Unbounded_String (" a "),
          To_Unbounded_String (" _check "));
      begin
        Check (String_Concat_Trim_WS_Chk);
      end;
    when String_Concat_Trim_WS_Chk =>
      declare
        S : constant String := (" This "," is "," a "," _check ");
      begin
        Check (Concat_Single_Element);
      end;
    when Concat_Single_Element =>
      declare
        S : constant Unbounded_Strings := (To_Unbounded_String (" Single "));
      begin
        Put_Line (To_String (Concat (S, True, True)));
      end;
  end case;
end Check;

end Main;
Put_Line (To_String (Concat (S, True, False)));
end;

when String_Concat_Trim_WS_Chk =>
declare
  S : constant Unbounded_Strings := (
    To_Unbounded_String (" This "),
    To_Unbounded_String (" is a "),
    To_Unbounded_String (" test. "));
begin
  Put_Line (Concat (S, True, True));
end;
when Concat_Single_Element =>
declare
  S : constant Unbounded_Strings := (1 => To_Unbounded_String (" Hi "));
begin
  Put_Line (Concat (S, True, True));
end;
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

116.2 List of events

Goal: create a system to manage a list of events.

Steps:
1. Implement the Events package.
   1. Declare the Event_Item subtype.
2. Implement the Events.Lists package.
   1. Adapt the Add procedure.
   2. Adapt the Display procedure.

Requirements:
1. The Event_Item type (from the Events package) contains the description of an event.
   1. This description is declared as a subtype of unbounded string.
2. Procedure Add adds an event into the list of events for a specific date.
   1. The declaration of E needs to be adapted to use unbounded strings.
3. Procedure Display must display all events for each date (ordered by date) using the following format:
   1. The arguments to Put_Line need to be adapted to use unbounded strings.

Remarks:
1. We use the lab on the list of events from the previous chapter (*Standard library: Dates & Times* (page 1933)) as a starting point.

Listing 4: events.ads

```ada
with Ada.Containers.Vectors;

package Events is

-- subtype Event_Item is

package Event_Item_Containers is new
  Ada.Containers.Vectors
  (Index_Type     => Positive,
   Element_Type   => Event_Item);

subtype Event_Items is Event_Item_Containers.Vector;

end Events;
```

Listing 5: events-lists.ads

```ada
with Ada.Calendar; use Ada.Calendar;
with Ada.Containers.Ordered_Maps;

package Events.Lists is

type Event_List is tagged private;

procedure Add (Events : in out Event_List;
                Event_Time :     Time;
                Event      :     String);

procedure Display (Events : Event_List);

private

package Event_Time_Item_Containers is new
  Ada.Containers.Ordered_Maps
  (Key_Type     => Time,
   Element_Type => Event_Items,
   "="          => Event_Item_Containers."=");

type Event_List is new Event_Time_Item_Containers.Map with null record;

end Events.Lists;
```

Listing 6: events-lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

package body Events.Lists is

procedure Add (Events : in out Event_List;
               Event_Time :     Time;
               Event      :     String) is
  use Event_Item_Containers;
  E : constant Event_Item := new String'(Event);
begin
  if not Events.Contains (Event_Time) then
    Events.Include (Event_Time, Empty_Vector);
  end if;
  Events.Include (Event_Time, E);
end Add;
```

(continues on next page)
end if;
Events (Event_Time).Append (E);
end Add;

function Date_Image (T: Time) return String is
Date_Img : constant String := Image (T);
beg
return Date_Img (1 .. 10);
end;

procedure Display (Events : Event_List) is
use Event_Time_Item_Containers;
T : Time;
beg
Put_Line ("EVENTS LIST");
for C in Events.Iterate loop
T := Key (C);
Put_Line ("- " & Date_Image (T));
for I of Events (C) loop
Put_Line (" - " & I.all);
end loop;
end loop;
end Display;
end Events.Lists;

Listing 7: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Events;
with Events.Lists; use Events.Lists;

procedure Main is
type Test_Case_Index is
(Unbounded_String_Chk,
 Event_List_Chk);

procedure Check (TC : Test_Case_Index) is
EL : Event_List;
beg
case TC is
when Unbounded_String_Chk =>
declare
S : constant Events.Event_Item := To_Unbounded_String ("Checked");
beg
Put_Line (To_String (S));
end;
when Event_List_Chk =>
EL.Add (Time_Of (2018, 2, 16),
"Final check");
EL.Add (Time_Of (2018, 2, 16),
"Release");
EL.Add (Time_Of (2018, 12, 3),
"Brother's birthday");
EL.Add (Time_Of (2018, 1, 1),
"New Year's Day");
EL.Display;
end case;
end Check;

begin
if Argument_Count < 1 then
  Put_Line("ERROR: missing arguments! Exiting...");
  return;
elsif Argument_Count > 1 then
  Put_Line("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;
117.1 Decibel Factor

**Goal:** implement functions to convert from Decibel values to factors and vice-versa.

**Steps:**
1. Implement the Decibels package.
   1. Implement the To_Decibel function.
   2. Implement the To_Factor function.

**Requirements:**
1. The subtypes Decibel and Factor are based on a floating-point type.
2. Function To_Decibel converts a multiplication factor (or ratio) to decibels.
   • For the implementation, use $20 \times \log_{10}(F)$, where $F$ is the factor/ratio.
3. Function To_Factor converts a value in decibels to a multiplication factor (or ratio).
   • For the implementation, use $10^{D/20}$, where $D$ is the value in Decibel.

**Remarks:**
1. The Decibel\(^{385}\) is used to express the ratio of two values on a logarithmic scale.
   1. For example, an increase of 6 dB corresponds roughly to a multiplication by two (or an increase by 100 % of the original value).
2. You can find the functions that you'll need for the calculation in the Ada.Numerics. Elementary_Functions package.

Listing 1: decibels.ads
```
package Decibels is

  subtype Decibel is Float;
  subtype Factor is Float;

  function To_Decibel (F : Factor) return Decibel;
  function To_Factor (D : Decibel) return Factor;

end Decibels;
```

\(^{385}\) https://en.wikipedia.org/wiki/Decibel
package body Decibels is

  function To_Decibel (F : Factor) return Decibel is
  begin
    return 0.0;
  end To_Decibel;

  function To_Factor (D : Decibel) return Factor is
  begin
    return 0.0;
  end To_Factor;

end Decibels;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Decibels; use Decibels;

procedure Main is
  type Test_Case_Index is
    (Db_Chk,
     Factor_Chk);
  procedure Check (TC : Test_Case_Index; V : Float) is
    package F_IO is new Ada.Text_IO.Float_IO (Factor);
    package D_IO is new Ada.Text_IO.Float_IO (Decibel);
    procedure Put_Decibel_Cnvt (D : Decibel) is
      F : constant Factor := To_Factor (D);
    begin
      D_IO.Put (D, 0, 2, 0);
      Put (" dB => Factor of ");
      F_IO.Put (F, 0, 2, 0);
      New_Line;
    end;
    procedure Put_Factor_Cnvt (F : Factor) is
      D : constant Decibel := To_Decibel (F);
    begin
      Put ("Factor of ");
      F_IO.Put (F, 0, 2, 0);
      Put (" => ");
      D_IO.Put (D, 0, 2, 0);
      Put_Line (" dB");
    end;
    begin
      case TC is
        when Db_Chk =>
          Put_Decibel_Cnvt (Decibel (V));
        when Factor_Chk =>
          Put_Factor_Cnvt (Factor (V));
      end case;
    end Check;
  begin
    (continues on next page)
if Argument_Count < 2 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
end Main;

117.2 Root-Mean-Square

Goal: implement a function to calculate the root-mean-square of a sequence of values.

Steps:
  1. Implement the Signals package.
     1. Implement the Rms function.

Requirements:
  1. Subtype Sig_Value is based on a floating-point type.
  2. Type Signal is an unconstrained array of Sig_Value elements.
  3. Function Rms calculates the RMS of a sequence of values stored in an array of type Signal.
     1. See the remarks below for a description of the RMS calculation.

Remarks:
  1. The root-mean-square \(^{386}\) (RMS) value is an important information associated with sequences of values.
     1. It's used, for example, as a measurement for signal processing.
     2. It is calculated by:
        1. Creating a sequence \(S\) with the square of each value of an input sequence \(S_{in}\).
        2. Calculating the mean value \(M\) of the sequence \(S\).
        3. Calculating the square-root \(R\) of \(M\).
     3. You can optimize the algorithm above by combining steps #1 and #2 into a single step.

Listing 4: signals.ads

```ada
package Signals is

    subtype Sig_Value is Float;

    type Signal is array (Natural range <>) of Sig_Value;

    function Rms (S : Signal) return Sig_Value;

end Signals;
```

\(^{386}\) https://en.wikipedia.org/wiki/Root_mean_square
Listing 5: signals.adb

```ada

package body Signals is

function Rms (S : Signal) return Sig_Value is
begin
    return 0.0;
end;

end Signals;
```

Listing 6: signals-std.ads

```ada
package Signals.Std is

Sample_Rate : Float := 8000.0;

function Generate_Sine (N : Positive; Freq : Float) return Signal;
function Generate_Square (N : Positive) return Signal;
function Generate_Triangular (N : Positive) return Signal;

end Signals.Std;
```

Listing 7: signals-std.adb

```ada
with Ada.Numerics; use Ada.Numerics;

package body Signals.Std is

function Generate_Sine (N : Positive; Freq : Float) return Signal is
    S : Signal (0 .. N - 1);
    begin
        for I in S'First .. S'Last loop
            S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
        end loop;
        return S;
    end;

function Generate_Square (N : Positive) return Signal is
    S : constant Signal (0 .. N - 1) := (others => 1.0);
    return S;
end;

function Generate_Triangular (N : Positive) return Signal is
    S : Signal (0 .. N - 1);
    S_Half : constant Natural := S'Last / 2;
    begin
        for I in S'First .. S_Half loop
            S (I) := 1.0 * (Float (I) / Float (S_Half));
        end loop;
        for I in S_Half .. S'Last loop
            S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
        end loop;
    end;
```

(continues on next page)
Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Signals; use Signals;
with Signals.Std; use Signals.Std;

procedure Main is
    type Test_Case_Index is
        (Sine_Signal_CHK,
         Square_Signal_CHK,
         Triangular_Signal_CHK);

    procedure Check (TC : Test_Case_Index) is
        package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
        N : constant Positive := 1024;
        S_Si : constant Signal := Generate_Sine (N, 440.0);
        S_Sq : constant Signal := Generate_Square (N);
        S_Tr : constant Signal := Generate_Triangular (N + 1);
    begin
        case TC is
            when Sine_Signal_CHK =>
                Put ("RMS of Sine Signal: ");
                Sig_IO.Put (Rms (S_Si), 0, 2, 0);
                New_Line;
            when Square_Signal_CHK =>
                Put ("RMS of Square Signal: ");
                Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
                New_Line;
            when Triangular_Signal_CHK =>
                Put ("RMS of Triangular Signal: ");
                Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
                New_Line;
        end case;
    end Check;

begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exitting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
    end if;

    Check (Test_Case_Index'Value (Argument (1)));
end Main;
117.3 Rotation

**Goal:** use complex numbers to calculate the positions of an object in a circle after rotation.

**Steps:**
1. Implement the Rotation package.
   1. Implement the Rotation function.

**Requirements:**
1. Type Complex_Points is an unconstrained array of complex values.
2. Function Rotation returns a list of positions (represented by the Complex_Points type) when dividing a circle in \(N\) equal slices.
   1. See the remarks below for a more detailed explanation.
3. Subtype Angle is based on a floating-point type.
4. Type Angles is an unconstrained array of angles.
5. Function To_Angles returns a list of angles based on an input list of positions.

**Remarks:**
1. Complex numbers are particularly useful in computer graphics to simplify the calculation of rotations.
   1. For example, let’s assume you’ve drawn an object on your screen on position (1.0, 0.0).
   2. Now, you want to move this object in a circular path — i.e. make it rotate around position (0.0, 0.0) on your screen.
     - You could use sine and cosine functions to calculate each position of the path.
     - However, you could also calculate the positions using complex numbers.
2. In this exercise, you’ll use complex numbers to calculate the positions of an object that starts on zero degrees — on position (1.0, 0.0) — and rotates around (0.0, 0.0) for \(N\) slices of a circle.
   1. For example, if we divide the circle in four slices, the object’s path will consist of following points / positions:

```
Point #1: ( 1.0,  0.0)
Point #2: ( 0.0,  1.0)
Point #3: (-1.0,  0.0)
Point #4: ( 0.0, -1.0)
Point #5: ( 1.0,  0.0)
```
   1. As expected, point #5 is equal to the starting point (point #1), since the object rotates around (0.0, 0.0) and returns to the starting point.
   2. We can also describe this path in terms of angles. The following list presents the angles for the path on a four-sliced circle:

```
Point #1: 0.00 degrees
Point #2: 90.00 degrees
Point #3: 180.00 degrees
Point #4: -90.00 degrees (= 270 degrees)
Point #5: 0.00 degrees
```
1. To rotate a complex number simply multiply it by a unit vector whose arg is the radian angle to be rotated: $Z = e^{i\theta}$

Listing 9: rotation.ads

```ada
with Ada.Numerics.Complex_Types;
use Ada.Numerics.Complex_Types;

package Rotation is
   type Complex_Points is array (Positive range <>) of Complex;
   function Rotation (N : Positive) return Complex_Points;
end Rotation;
```

Listing 10: rotation.adb

```ada
with Ada.Numerics; use Ada.Numerics;

package body Rotation is
   function Rotation (N : Positive) return Complex_Points is
      C : Complex_Points (1 .. 1) := (others => (0.0, 0.0));
      begin
         return C;
      end;
end Rotation;
```

Listing 11: angles.ads

```ada
with Rotation; use Rotation;

package Angles is
   subtype Angle is Float;
   type Angles is array (Positive range <>) of Angle;
   function To_Angles (C : Complex_Points) return Angles;
end Angles;
```

Listing 12: angles.adb

```ada
with Ada.Numerics; use Ada.Numerics;

package body Angles is
   function To_Angles (C : Complex_Points) return Angles is
      begin
         return A : Angles (C'Range) do
            for I in A'Range loop
               A (I) := Argument (C (I)) / Pi * 180.0;
            end loop;
         end return;
      end To_Angles;
end Angles;
```
Listing 13: rotation-tests.ads

```ada
package Rotation.Tests is
    procedure Test_Rotation (N : Positive);
    procedure Test_Angles (N : Positive);
end Rotation.Tests;
```

Listing 14: rotation-tests.adb

```ada
with Ada.Text_IO;        use Ada.Text_IO;
with Ada.Text_IO.Complex_IO; use Ada.Text_IO.Complex_IO (Complex_Types);
with Ada.Numerics;       use Ada.Numerics;
with Angles;            use Angles;

package body Rotation.Tests is
    package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
    package F_IO is new Ada.Text_IO.Float_IO (Float);

-- Adapt value due to floating-point inaccuracies

function Adapt (C : Complex) return Complex is
    function Check_Zero (F : Float) return Float is
        (if F <= 0.0 and F >= -0.01 then 0.0 else F);
    begin
        return C_Out : Complex := C do
            C_Out.Re := Check_Zero (C_Out.Re);
            C_Out.Im := Check_Zero (C_Out.Im);
        end return;
    end Adapt;

function Adapt (A : Angle) return Angle is
    (if A <= -179.99 and A >= -180.01 then 180.0 else A);

procedure Test_Rotation (N : Positive) is
    C : constant Complex_Points := Rotation (N);
begin
    Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
    for V of C loop
        Put ("Point: ");
        C_IO.Put (Adapt (V), 0, 1, 0);
        New_Line;
    end loop;
end Test_Rotation;

procedure Test_Angles (N : Positive) is
    C : constant Complex_Points := Rotation (N);
    A : constant Angles.Angles := To_Angles (C);
begin
    Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
    for V of A loop
        Put ("Angle: ");
        F_IO.Put (Adapt (V), 0, 2, 0);
        Put_Line (" degrees");
    end loop;
end Test_Angles;
```

(continues on next page)
Listing 15: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Rotation.Tests; use Rotation.Tests;

procedure Main is

    type Test_Case_Index is
    (Rotation_Chk, Angles_Chk);

    procedure Check (TC : Test_Case_Index; N : Positive) is
        begin
            case TC is
                when Rotation_Chk =>
                    Test_Rotation (N);
                when Angles_Chk =>
                    Test_Angles (N);
            end case;
        end Check;

    begin
        if Argument_Count < 2 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 2 then
            Put_Line ("Ignoring additional arguments...");
        end if;

        Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
    end Main;
```

117.3. Rotation
118.1 Imperative Language

118.1.1 Hello World

Listing 1: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
  Put_Line ("Hello World!");
end Main;
```

118.1.2 Greetings

Listing 2: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  procedure Greet (Name : String) is
  begin
    Put_Line ("Hello " & Name & "!");
  end Greet;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Greet (Argument (1));
end Main;
```
### 118.1.3 Positive Or Negative

Listing 3: classify_number.ads

```ada
procedure Classify_Number (X : Integer);
```

Listing 4: classify_number.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Classify_Number (X : Integer) is
begin
  if X > 0 then
    Put_Line ("Positive");
  elsif X < 0 then
    Put_Line ("Negative");
  else
    Put_Line ("Zero");
  end if;
end Classify_Number;
```

Listing 5: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Classify_Number;

procedure Main is
  A : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  A := Integer'Value (Argument (1));
  Classify_Number (A);
end Main;
```

### 118.1.4 Numbers

Listing 6: display_numbers.ads

```ada
procedure Display_Numbers (A, B : Integer);
```

Listing 7: display_numbers.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_Numbers (A, B : Integer) is
  X, Y : Integer;
begin
  if A <= B then
    X := A;
  end if;
end Display_Numbers;
```
Y := B;
else
  X := B;
  Y := A;
end if;
for I in X .. Y loop
  Put_Line (Integer'Image (I));
end loop;
end Display_Numbers;

Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Numbers;
procedure Main is
  A, B : Integer;
begnin
  if Argument_Count < 2 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  A := Integer'Value (Argument (1));
  B := Integer'Value (Argument (2));
  Display_Numbers (A, B);
end Main;

118.2 Subprograms

118.2.1 Subtract Procedure

Listing 9: subtract.ads

procedure Subtract (A, B : Integer;
                     Result : out Integer);

Listing 10: subtract.adb

procedure Subtract (A, B : Integer;
                     Result : out Integer) is
  begin
    Result := A - B;
  end Subtract;

Listing 11: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

(continues on next page)
with Subtract;

procedure Main is
    type Test_Case_Index is
        (Sub_10_1_Chk,
         Sub_10_100_Chk,
         Sub_0_5_Chk,
         Sub_0_Minus_5_Chk);

    procedure Check (TC : Test_Case_Index) is
        Result : Integer;
    begin
        case TC is
            when Sub_10_1_Chk =>
                Subtract (10, 1, Result);
                Put_Line ("Result: " & Integer'Image (Result));
            when Sub_10_100_Chk =>
                Subtract (10, 100, Result);
                Put_Line ("Result: " & Integer'Image (Result));
            when Sub_0_5_Chk =>
                Subtract (0, 5, Result);
                Put_Line ("Result: " & Integer'Image (Result));
            when Sub_0_Minus_5_Chk =>
                Subtract (0, -5, Result);
                Put_Line ("Result: " & Integer'Image (Result));
        end case;
    end Check;

    begin
        if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
        end if;
        Check (Test_Case_Index'Value (Argument (1)));
    end Main;

118.2.2 Subtract Function

Listing 12: subtract.ads

function Subtract (A, B : Integer) return Integer;

Listing 13: subtract.adb

function Subtract (A, B : Integer) return Integer is
begin
    return A - B;
end Subtract;

Listing 14: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Subtract; (continues on next page)
procedure Main is
  type Test_Case_Index is
    (Sub_10_1_Chk,
     Sub_10_100_Chk,
     Sub_0_5_Chk,
     Sub_0_Minus_5_Chk);
  procedure Check (TC : Test_Case_Index) is
    Result : Integer;
  begin
    case TC is
      when Sub_10_1_Chk =>
        Result := Subtract (10, 1);
        Put_Line ("Result: " & Integer'Image (Result));
      when Sub_10_100_Chk =>
        Result := Subtract (10, 100);
        Put_Line ("Result: " & Integer'Image (Result));
      when Sub_0_5_Chk =>
        Result := Subtract (0, 5);
        Put_Line ("Result: " & Integer'Image (Result));
      when Sub_0_Minus_5_Chk =>
        Result := Subtract (0, -5);
        Put_Line ("Result: " & Integer'Image (Result));
    end case;
  end Check;
  begin
    if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
    return;
    elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
  end Main;

118.2.3 Equality function

Listing 15: is_equal.ads

function Is_Equal (A, B : Integer) return Boolean;

Listing 16: is_equal.adb

function Is_Equal (A, B : Integer) return Boolean is
  begin
    return A = B;
  end Is_Equal;

Listing 17: main.adb

with Ada.Command_Line;     use Ada.Command_Line;
with Ada.Text_IO;           use Ada.Text_IO;
with Is_Equal;

(continues on next page)
procedure Main is
  type Test_Case_Index is
  (Equal_CHK, Inequal_CHK);
  procedure Check (TC : Test_Case_Index) is
    procedure Display_Equal (A, B : Integer; Equal : Boolean) is
      begin
        Put (Integer'Image (A));
        if Equal then
          Put (" is equal to ");
        else
          Put (" isn't equal to ");
        end if;
        Put_Line (Integer'Image (B) & ".");
      end Display_Equal;
      Result : Boolean;
      begin
        case TC is
          when Equal_CHK =>
            for I in 0 .. 10 loop
              Result := Is_Equal (I, I);
              Display_Equal (I, I, Result);
            end loop;
          when Inequal_CHK =>
            for I in 0 .. 10 loop
              Result := Is_Equal (I, I - 1);
              Display_Equal (I, I - 1, Result);
            end loop;
        end case;
      end Check;
      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting...");
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;
        Check (Test_Case_Index'Value (Argument (1)));
      end Main;
end Main;

118.2.4 States

Listing 18: display_state.ads

procedure Display_State (State : Integer);

Listing 19: display_state.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Display_State (State : Integer) is
  begin
    Display_State (State : Integer) is
    (continues on next page)
case State is
    when 0 =>
        Put_Line ("Off");
    when 1 =>
        Put_Line ("On: Simple Processing");
    when 2 =>
        Put_Line ("On: Advanced Processing");
    when others =>
        null;
end case;
end Display_State;

Listing 20: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_State;

procedure Main is
    State : Integer;
begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
        end if;
    State := Integer'Value (Argument (1));
    Display_State (State);
end Main;

118.2.5 States #2

Listing 21: get_state.ads

function Get_State (State : Integer) return String;

Listing 22: get_state.adb

function Get_State (State : Integer) return String is
begin
    return (case State is
    when 0 => "Off",
    when 1 => "On: Simple Processing",
    when 2 => "On: Advanced Processing",
    when others => "");
end Get_State;

Listing 23: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Get_State;
**118.2.6 States #3**

Listing 24: is_on.ads

```ada
function Is_On (State : Integer) return Boolean;
```

Listing 25: is_on.adb

```ada
with Ada.Text_IO;
use Ada.Text_IO;

function Is_On (State : Integer) return Boolean is
begin
    return not (State = 0);
end Is_On;
```

Listing 26: display_on_off.ads

```ada
procedure Display_On_Off (State : Integer);
```

Listing 27: display_on_off.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_On_Off (State : Integer) is
begin
    Put_Line (if Is_On (State) then "On" else "Off");
end Display_On_Off;
```

Listing 28: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_On_Off; use Is_On;

procedure Main is
State : Integer;
begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
        end if;
    State := Integer'Value (Argument (1));
    Put_Line (Get_State (State));
end Main;
```

(continues on next page)
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments..." RESPONSE);
end if;

State := Integer'Value (Argument (1));
Display_On_Off (State);
Put_Line (Boolean'Image (Is_On (State)));

end Main;

118.2.7 States #4

Listing 29: set_next.ads
procedure Set_Next (State : in out Integer);

Listing 30: set_next.adb
procedure Set_Next (State : in out Integer) is
begin
  State := (if State < 2 then State + 1 else 0);
end Set_Next;

Listing 31: main.adb
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Set_Next;

procedure Main is
  State : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting..." RESPONSE);
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments..." RESPONSE);
  end if;
  State := Integer'Value (Argument (1));
  Set_Next (State);
  Put_Line (Integer'Image (State));
end Main;

118.2. Subprograms
118.3 Modular Programming

118.3.1 Months

Listing 32: months.ads

```ada
package Months is

    Jan : constant String := "January";
    Feb : constant String := "February";
    Mar : constant String := "March";
    Apr : constant String := "April";
    May : constant String := "May";
    Jun : constant String := "June";
    Jul : constant String := "July";
    Aug : constant String := "August";
    Sep : constant String := "September";
    Oct : constant String := "October";
    Nov : constant String := "November";
    Dec : constant String := "December";

    procedure Display_Months;

end Months;
```

Listing 33: months.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Months is

    procedure Display_Months is
    begin
        Put_Line ("Months:");
        Put_Line (". " & Jan);
        Put_Line (". " & Feb);
        Put_Line (". " & Mar);
        Put_Line (". " & Apr);
        Put_Line (". " & May);
        Put_Line (". " & Jun);
        Put_Line (". " & Jul);
        Put_Line (". " & Aug);
        Put_Line (". " & Sep);
        Put_Line (". " & Oct);
        Put_Line (". " & Nov);
        Put_Line (". " & Dec);
    end Display_Months;

end Months;
```

Listing 34: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Months; use Months;

procedure Main is

    type Test_Case_Index is
```

(continues on next page)
Learning Ada
(continued from previous page)

```ada
(Months_Chk);

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Months_Chk =>
      Display_Months;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
```

### 118.3.2 Operations

**Listing 35: operations.ads**

```ada
package Operations is
  function Add (A, B : Integer) return Integer;
  function Subtract (A, B : Integer) return Integer;
  function Multiply (A, B : Integer) return Integer;
  function Divide (A, B : Integer) return Integer;
end Operations;
```

**Listing 36: operations.adb**

```ada
package body Operations is
  function Add (A, B : Integer) return Integer is
  begin
    return A + B;
  end Add;

  function Subtract (A, B : Integer) return Integer is
  begin
    return A - B;
  end Subtract;

  function Multiply (A, B : Integer) return Integer is
  begin
    return A * B;
  end Multiply;

  function Divide (A, B : Integer) return Integer is
  begin
```

(continues on next page)
Learning Ada

Listing 37: operations-test.ads

```ada
package Operations.Test is
    procedure Display (A, B : Integer);
end Operations.Test;
```

Listing 38: operations-test.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Operations.Test is
    procedure Display (A, B : Integer) is
        A_Str : constant String := Integer'Image (A);
        B_Str : constant String := Integer'Image (B);
        begin
            Put_Line ("Operations: ");
            Put_Line (A_Str & " + " & B_Str & " = 
                        & Integer'Image (Add (A, B)) & ",");
            Put_Line (A_Str & " - " & B_Str & " = 
                        & Integer'Image (Subtract (A, B)) & ",");
            Put_Line (A_Str & " * " & B_Str & " = 
                        & Integer'Image (Multiply (A, B)) & ",");
            Put_Line (A_Str & " / " & B_Str & " = 
                        & Integer'Image (Divide (A, B)) & ",");
        end Display;
end Operations.Test;
```

Listing 39: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Operations; use Operations.Test;

procedure Main is

    type Test_Case_Index is
        (Operations_CHK, Operations_Display_CHK);

    procedure Check (TC : Test_Case_Index) is
        begin
            case TC is
                when Operations_CHK =>
                    Put_Line ("Add (100, 2) = " 
                                & Integer'Image (Operations.Add (100, 2)));
```

(continues on next page)
(continued from previous page)

Learning Ada

Put_Line ("Subtract (100, 2) = 
& Integer'Image (Operations.Subtract (100, 2)));

Put_Line ("Multiply (100, 2) = 
& Integer'Image (Operations.Multiply (100, 2)));

Put_Line ("Divide (100, 2) = 
& Integer'Image (Operations.Divide (100, 2)));

when Operations_Display_Chk =>
  Display (10, 5);
  Display (1, 2);
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.4 Strongly typed language

118.4.1 Colors

Listing 40: color_types.ads

package Color_Types is

  type HTML_Color is
  (Salmon,
   Firebrick,
   Red,
   Darkred,
   Lime,
   Forestgreen,
   Green,
   Darkgreen,
   Blue,
   Mediumblue,
   Darkblue);

  function To_Integer (C : HTML_Color) return Integer;

  type Basic_HTML_Color is
  (Red,
   Green,
   Blue);

  function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;

end Color_Types;
Learning Ada

Listing 41: color_types.adb
1

package body Color_Types is

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

function To_Integer (C
begin
case C is
when Salmon
when Firebrick
when Red
when Darkred
when Lime
when Forestgreen
when Green
when Darkgreen
when Blue
when Mediumblue
when Darkblue
end case;

: HTML_Color) return Integer is

=>
=>
=>
=>
=>
=>
=>
=>
=>
=>
=>

return
return
return
return
return
return
return
return
return
return
return

16#FA8072#;
16#B22222#;
16#FF0000#;
16#8B0000#;
16#00FF00#;
16#228B22#;
16#008000#;
16#006400#;
16#0000FF#;
16#0000CD#;
16#00008B#;

18
19

end To_Integer;

20
21
22
23
24
25
26
27
28

function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
begin
case C is
when Red
=> return Red;
when Green => return Green;
when Blue => return Blue;
end case;
end To_HTML_Color;

29
30

end Color_Types;

Listing 42: main.adb
1
2
3

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO;
use Ada.Text_IO;
with Ada.Integer_Text_IO;

4
5

with Color_Types;

use Color_Types;

6
7
8
9
10
11

procedure Main is
type Test_Case_Index is
(HTML_Color_Range,
HTML_Color_To_Integer,
Basic_HTML_Color_To_HTML_Color);

12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

procedure Check (TC : Test_Case_Index) is
begin
case TC is
when HTML_Color_Range =>
for I in HTML_Color'Range loop
Put_Line (HTML_Color'Image (I));
end loop;
when HTML_Color_To_Integer =>
for I in HTML_Color'Range loop
Ada.Integer_Text_IO.Put (Item => To_Integer (I),
Width => 1,
Base => 16);
New_Line;
end loop;
(continues on next page)

1968

Chapter 118. Solutions


27   when Basic_HTML_Color_To_HTML_Color =>
28     for I in Basic_HTML_Color'Range loop
29       Put_Line (HTML_Color'Image (To_HTML_Color (I)));
30     end loop;
31   end case;
32   end loop
33 end case;
34 end Check;
35 begin
36   if Argument_Count < 1 then
37     Put_Line ("ERROR: missing arguments! Exiting...");
38     return;
39   elsif Argument_Count > 1 then
40     Put_Line ("Ignoring additional arguments...");
41   end if;
42   Check (Test_Case_Index'Value (Argument (1)));
43 end Main;

118.4.2 Integers

Listing 43: int_types.ads

package Int_Types is
3   type I_100 is range 0 .. 100;
4   type U_100 is mod 101;
5   function To_I_100 (V : U_100) return I_100;
6   function To_U_100 (V : I_100) return U_100;
7   type D_50 is new I_100 range 10 .. 50;
8   subtype S_50 is I_100 range 10 .. 50;
9   function To_D_50 (V : I_100) return D_50;
10  function To_S_50 (V : I_100) return S_50;
11  function To_I_100 (V : D_50) return I_100;
12 end Int_Types;

Listing 44: int_types.adb

package body Int_Types is
3   function To_I_100 (V : U_100) return I_100 is
4     begin
5       return I_100 (V);
6     end To_I_100;
7
8   function To_U_100 (V : I_100) return U_100 is
9     begin
10       return U_100 (V);
11     end To_U_100;
12 end Int_Types;
function To_D_50 (V : I_100) return D_50 is
    Min : constant I_100 := I_100 (D_50'First);
    Max : constant I_100 := I_100 (D_50'Last);
begin
    if V > Max then
        return D_50'Last;
    elsif V < Min then
        return D_50'First;
    else
        return D_50 (V);
    end if;
end To_D_50;

function To_S_50 (V : I_100) return S_50 is
begin
    if V > S_50'Last then
        return S_50'Last;
    elsif V < S_50'First then
        return S_50'First;
    else
        return V;
    end if;
end To_S_50;

function To_I_100 (V : D_50) return I_100 is
begin
    return I_100 (V);
end To_I_100;
end Int_Types;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Int_Types; use Int_Types;

procedure Main is
    package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
    package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
    package D_50_IO is new Ada.Text_IO.Integer_IO (D_50);

    use I_100_IO;
    use U_100_IO;
    use D_50_IO;

    type Test_Case_Index is
        (I_100_Range,
         U_100_Range,
         U_100_Wraparound,
         U_100_To_I_100,
         I_100_To_I_100,
         D_50_Range,
         S_50_Range,
         I_100_To_D_50,
         I_100_To_S_50,
         D_50_To_I_100,
         S_50_To_I_100);

    procedure Check (TC : Test_Case_Index) is
begin
  I_100_IO.Default_Width := 1;
  U_100_IO.Default_Width := 1;
  D_50_IO.Default_Width := 1;

  case TC is
    when I_100_Range =>
      Put (I_100'First);
      New_Line;
      Put (I_100'Last);
      New_Line;
    when U_100_Range =>
      Put (U_100'First);
      New_Line;
      Put (U_100'Last);
      New_Line;
    when U_100_Wraparound =>
      Put (U_100'First - 1);
      New_Line;
      Put (U_100'Last + 1);
      New_Line;
    when U_100_To_I_100 =>
      for I in U_100'Range loop
        I_100_IO.Put (To_I_100 (I));
        New_Line;
      end loop;
    when I_100_To_U_100 =>
      for I in I_100'Range loop
        Put (To_U_100 (I));
        New_Line;
      end loop;
    when D_50_Range =>
      Put (D_50'First);
      New_Line;
      Put (D_50'Last);
      New_Line;
    when S_50_Range =>
      Put (S_50'First);
      New_Line;
      Put (S_50'Last);
      New_Line;
    when I_100_To_D_50 =>
      for I in I_100'Range loop
        Put (To_D_50 (I));
        New_Line;
      end loop;
    when I_100_To_S_50 =>
      for I in I_100'Range loop
        Put (To_S_50 (I));
        New_Line;
      end loop;
    when D_50_To_I_100 =>
      for I in D_50'Range loop
        Put (To_I_100 (I));
        New_Line;
      end loop;
    when S_50_To_I_100 =>
      for I in S_50'Range loop
        Put (I);
        New_Line;
      end loop;
  end case;
(continues on next page)
end case;
end Check;

begin
if Argument_Count < 1 then
   Put_Line ("ERROR: missing arguments! Exiting...");
   return;
elsif Argument_Count > 1 then
   Put_Line ("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.4.3 Temperatures

Listing 46: temperature_types.ads

package Temperature_Types is
   type Celsius is digits 6 range -273.15 .. 5504.85;
   type Int_Celsius is range -273 .. 5505;
   function To_Celsius (T : Int_Celsius) return Celsius;
   function To_Int_Celsius (T : Celsius) return Int_Celsius;
   type Kelvin is digits 6 range 0.0 .. 5778.00;
   function To_Celsius (T : Kelvin) return Celsius;
   function To_Kelvin (T : Celsius) return Kelvin;
end Temperature_Types;

Listing 47: temperature_types.adb

package body Temperature_Types is
   function To_Celsius (T : Int_Celsius) return Celsius is
      Min : constant Float := Float (Celsius'First);
      Max : constant Float := Float (Celsius'Last);
      F : constant Float := Float (T);
      begin
         if F > Max then
            return Celsius (Max);
         elsif F < Min then
            return Celsius (Min);
         else
            return Celsius (F);
         end if;
      end To_Celsius;
      function To_Int_Celsius (T : Celsius) return Int_Celsius is
      begin
         return Int_Celsius (T);
      end
end To_Int_Celsius;

function To_Celsius (T : Kelvin) return Celsius is
  F : constant Float := Float (T);
begin
  return Celsius (F - 273.15);
end To_Celsius;

function To_Kelvin (T : Celsius) return Kelvin is
  F : constant Float := Float (T);
begin
  return Kelvin (F + 273.15);
end To_Kelvin;
end Temperature_Types;

Listing 48: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Temperature_Types; use Temperature_Types;

procedure Main is
  package Celsius_IO is new Ada.Text_IO.Float_IO (Celsius);
  package Kelvin_IO is new Ada.Text_IO.Float_IO (Kelvin);
  package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);
use Celsius_IO;
use Kelvin_IO;
use Int_Celsius_IO;

type Test_Case_Index is
  (Celsius_Range, Celsius_To_Int_Celsius, Int_Celsius_To_Celsius, Celsius_To_Kelvin);

procedure Check (TC : Test_Case_Index) is
begin
  Celsius IO.Default_Fore := 1;
  Kelvin_IO.Default_Fore := 1;
  Int_Celsius_IO.Default_Width := 1;
  case TC is
  when Celsius_Range =>
    Put (Celsius'First);
    New_Line;
    Put (Celsius'Last);
    New_Line;
  when Celsius_To_Int_Celsius =>
    Put (To_Int_Celsius (Celsius'First));
    New_Line;
    Put (To_Int_Celsius (0.0));
    New_Line;
    Put (To_Int_Celsius (Celsius'Last));
    New_Line;
  when Int_Celsius_To_Celsius =>
    Put (To_Celsius (Int_Celsius'First));
    New_Line;
  when others =>
    null;
  end case;
end Check;

118.4. Strongly typed language 1973
Learning Ada

(continued from previous page)

44  Put (To_Celsius (0));
45  New_Line;
46  Put (To_Celsius (Int_Celsius'Last));
47  New_Line;
48  when Kelvin_To_Celsius =>
49  Put (To_Celsius (Kelvin'First));
50  New_Line;
51  Put (To_Celsius (0));
52  New_Line;
53  Put (To_Celsius (Kelvin'Last));
54  New_Line;
55  when Celsius_To_Kelvin =>
56  Put (To_Kelvin (Celsius'First));
57  New_Line;
58  Put (To_Kelvin (Celsius'Last));
59  New_Line;
60  end case;
61 end Check;
62
63 begin
64  if Argument_Count < 1 then
65    Put_Line ("ERROR: missing arguments! Exiting...");
66    return;
67  elsif Argument_Count > 1 then
68    Put_Line ("Ignoring additional arguments...");
69  end if;
70  Check (Test_Case_Index'Value (Argument (1)));
71 end Main;

118.5 Records

118.5.1 Directions

Listing 49: directions.ads

package Directions is

  type Angle_Mod is mod 360;

  type Direction is
    (North,
     Northeast,
     East,
     Southeast,
     South,
     Southwest,
     West,
     Northwest);

  function To_Direction (N: Angle_Mod) return Direction;

  type Ext_Angle is record
    Angle_Elem : Angle_Mod;
    Direction_Elem : Direction;
  end record;

(continues on next page)
function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;

procedure Display (N : Ext_Angle);

end Directions;

Listing 50: directions.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Directions is

procedure Display (N : Ext_Angle) is
  begin
    Put_Line ("Angle: ",
              & Angle_Mod'Image (N.Angle_Elem)
              & " => ",
              & Direction'Image (N.Direction_Elem)
              & ".");
  end Display;

function To_Direction (N : Angle_Mod) return Direction is
  begin
    case N is
      when 0 => return North;
      when 1 .. 89 => return Northeast;
      when 90 => return East;
      when 91 .. 179 => return Southeast;
      when 180 => return South;
      when 181 .. 269 => return Southwest;
      when 270 => return West;
      when 271 .. 359 => return Northwest;
    end case;
  end To_Direction;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
  begin
    return (Angle_Elem => N,
            Direction_Elem => To_Direction (N));
  end To_Ext_Angle;

end Directions;

Listing 51: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Directions; use Directions;

procedure Main is
  type Test_Case_Index is
  (Direction_Chk);

  procedure Check (TC : Test_Case_Index) is
    begin
      case TC is
        when Direction_Chk =>
          Display (To_Ext_Angle (0));
          Display (To_Ext_Angle (30));
Learning Ada

(continued from previous page)

    Display (To_Ext_Angle (45));
    Display (To_Ext_Angle (90));
    Display (To_Ext_Angle (91));
    Display (To_Ext_Angle (120));
    Display (To_Ext_Angle (180));
    Display (To_Ext_Angle (250));
    Display (To_Ext_Angle (270));
    end case;
end Check;

begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.5.2 Colors

Listing 52: color_types.ads

package Color_Types is

    type HTML_Color is
        (Salmon,
         Firebrick,
         Red,
         Darkred,
         Lime,
         Forestgreen,
         Green,
         Darkgreen,
         Blue,
         Mediumblue,
         Darkblue);

    function To_Integer (C : HTML_Color) return Integer;

    type Basic_HTML_Color is
        (Red,
         Green,
         Blue);

    function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;

    subtype Int_Color is Integer range 0 .. 255;

    type RGB is record
        Red : Int_Color;
        Green : Int_Color;
        Blue : Int_Color;
    end record;

    function To_RGB (C : HTML_Color) return RGB;

(continues on next page)
function Image (C : RGB) return String;
end Color_Types;

Listing 53: color_types.adb

with Ada.Integer_Text_IO;
package body Color_Types is
  function To_Integer (C : HTML_Color) return Integer is
    begin
      case C is
        when Salmon => return 16#FA8072#;
        when Firebrick => return 16#B22222#;
        when Red => return 16#FF0000#;
        when Darkred => return 16#8B0000#;
        when Lime => return 16#00FF00#;
        when Forestgreen => return 16#228B22#;
        when Green => return 16#008000#;
        when Darkgreen => return 16#006400#;
        when Blue => return 16#0000FF#;
        when Mediumblue => return 16#0000CD#;
        when Darkblue => return 16#00008B#;
      end case;
      end To_Integer;

  function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
    begin
      case C is
        when Red => return Red;
        when Green => return Green;
        when Blue => return Blue;
      end case;
      end To_HTML_Color;

  function To_RGB (C : HTML_Color) return RGB is
    begin
      case C is
        when Salmon => return (16#FA#, 16#80#, 16#72#);
        when Firebrick => return (16#B2#, 16#22#, 16#22#);
        when Red => return (16#FF#, 16#00#, 16#00#);
        when Darkred => return (16#8B#, 16#00#, 16#00#);
        when Lime => return (16#00#, 16#FF#, 16#00#);
        when Forestgreen => return (16#22#, 16#8B#, 16#22#);
        when Green => return (16#00#, 16#80#, 16#00#);
        when Darkgreen => return (16#00#, 16#64#, 16#00#);
        when Blue => return (16#00#, 16#00#, 16#FF#);
        when Mediumblue => return (16#00#, 16#00#, 16#CD#);
        when Darkblue => return (16#00#, 16#00#, 16#8B#);
      end case;
      end To_RGB;

  function Image (C : RGB) return String is
    subtype Str_Range is Integer range 1 .. 10;
    SR : String (Str_Range);
    SG : String (Str_Range);
    SB : String (Str_Range);
    begin
      (continues on next page)
Ada.Integer_Text_IO.Put (To => SR,
                     Item => C.Red,
                     Base => 16);
Ada.Integer_Text_IO.Put (To => SG,
                     Item => C.Green,
                     Base => 16);
Ada.Integer_Text_IO.Put (To => SB,
                     Item => C.Blue,
                     Base => 16);
return ("(Red => " & SR
       & ", Green => " & SG
       & ", Blue => " & SB
       & ")");
end Image;
end Color_Types;

Listing 54: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is
    (HTML_Color_To_RGB);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when HTML_Color_To_RGB =>
        for I in HTML_Color'Range loop
          Put_Line (HTML_Color'Image (I) & " => "
                     & Image (To_RGB (I)) & ");
        end loop;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
### 118.5.3 Inventory

#### Listing 55: inventory_pkg.ads

```ada
package Inventory_Pkg is

  type Item_Name is
    (Ballpoint_Pen, Oil-Based_Pen_Marker, Feather_Quill_Pen);

  function To_String (I : Item_Name) return String;

  type Item is record
    Name : Item_Name;
    Quantity : Natural;
    Price : Float;
  end record;

  function Init (Name : Item_Name;
                Quantity : Natural;
                Price : Float) return Item;

  procedure Add (Assets : in out Float;
                 I : Item);

end Inventory_Pkg;
```

#### Listing 56: inventory_pkg.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Inventory_Pkg is

  function To_String (I : Item_Name) return String is
  begin
    case I is
      when Ballpoint_Pen => return "Ballpoint Pen";
      when Oil-Based_Pen_Marker => return "Oil-based Pen Marker";
      when Feather_Quill_Pen => return "Feather Quill Pen";
    end case;
  end To_String;

  function Init (Name : Item_Name;
                 Quantity : Natural;
                 Price : Float) return Item is
  begin
    Put_Line ("Item: " & To_String (Name) & ".");
    return (Name => Name,
            Quantity => Quantity,
            Price => Price);
  end Init;

  procedure Add (Assets : in out Float;
                 I : Item) is
  begin
    Assets := Assets + Float (I.Quantity) * I.Price;
  end Add;

end Inventory_Pkg;
```
with Ada.Command_Line; use Ada.Command_Line;

with Ada.Text_IO; use Ada.Text_IO;

with Inventory_Pkg; use Inventory_Pkg;

procedure Main is
   -- Remark: the following line is not relevant.
   F : array (1 .. 10) of Float := (others => 42.42);

type Test_Case_Index is
   (Inventory_Chk);

procedure Display (Assets : Float) is
   package F_IO is new Ada.Text_IO.Float_IO (Float);
   use F_IO;
   begin
      Put ("Assets: ", Assets, 1, 2, 0);
      New_Line;
   end Display;

procedure Check (TC : Test_Case_Index) is
   I : Item;
   Assets : Float := 0.0;

   -- Please ignore the following three lines!
   pragma Warnings (Off, "default initialization");
   for Assets'Address use F'Address;
   pragma Warnings (On, "default initialization");
   begin
      case TC is
         when Inventory_Chk =>
            I := Init (Ballpoint_Pen, 185, 0.15);
            Add (Assets, I);
            Display (Assets);

            I := Init (Oil_Based_Pen_Marker, 100, 9.0);
            Add (Assets, I);
            Display (Assets);

            I := Init (Feather_Quill_Pen, 2, 40.0);
            Add (Assets, I);
            Display (Assets);
      end case;
      end Check;
   begin
      if Argument_Count < 1 then
         Put_Line ("ERROR: missing arguments! Exiting...");
         return;
      elsif Argument_Count > 1 then
         Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
   end Main;
118.6 Arrays

118.6.1 Constrained Array

Listing 58: constrained_arrays.ads

```ada
package Constrained_Arrays is

  type My_Index is range 1 .. 10;
  type My_Array is array (My_Index) of Integer;

  function Init return My_Array;
  procedure Double (A : in out My_Array);
  function First_Elem (A : My_Array) return Integer;
  function Last_Elem (A : My_Array) return Integer;
  function Length (A : My_Array) return Integer;

  A : My_Array := (1, 2, others => 42);

end Constrained_Arrays;
```

Listing 59: constrained_arrays.adb

```ada
package body Constrained_Arrays is

  function Init return My_Array is
    A : My_Array;
    begin
      for I in A'Range loop
        A (I) := Integer (I);
      end loop;
      return A;
    end Init;

  procedure Double (A : in out My_Array) is
    begin
      for I in A'Range loop
        A (I) := A (I) * 2;
      end loop;
    end Double;

  function First_Elem (A : My_Array) return Integer is
    begin
      return A (A'First);
    end First_Elem;

  function Last_Elem (A : My_Array) return Integer is
    begin
      return A (A'Last);
    end Last_Elem;

  function Length (A : My_Array) return Integer is
    begin
      return A'Length;
    end Length;

end Constrained_Arrays;
```

(continues on next page)
Listing 60: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Constrained_Arrays; use Constrained_Arrays;

procedure Main is
  type Test_Case_Index is
    (Range_Chk,
     Array_Range_Check,
     A_Obj_Chk,
     Init_Chk,
     Double_Chk,
     First_Elem_Chk,
     Last_Elem_Chk,
     Length_Chk);

  procedure Check (TC : Test_Case_Index) is
    AA : My_Array;

    procedure Display (A : My_Array) is
      begin
        for I in A'Range loop
          Put_Line (Integer'Image (A (I)));
        end loop;
      end Display;

    procedure Local_Init (A : in out My_Array) is
      begin
        A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
      end Local_Init;

      begin
        case TC is
          when Range_Chk =>
            for I in My_Index loop
              Put_Line (My_Index'Image (I));
            end loop;
          when Array_Range_Chk =>
            for I in My_Array'Range loop
              Put_Line (My_Index'Image (I));
            end loop;
          when A_Obj_Chk =>
            Display (A);
          when Init_Chk =>
            AA := Init;
            Display (AA);
          when Double_Chk =>
            Local_Init (AA);
            Display (AA);
          when First_Elem_Chk =>
            Local_Init (AA);
            Put_Line (Integer'Image (First_Elem (AA)));
          when Last_Elem_Chk =>
            Local_Init (AA);
            Put_Line (Integer'Image (Last_Elem (AA)));
```
when Length_Chk =>
  Put_Line (Integer'Image (Length (AA)));
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.6.2 Colors: Lookup-Table

Listing 61: color_types.ads

package Color_Types is
  type HTML_Color is
    (Salmon, Firebrick, Red, Darkred, Lime, Forestgreen, Green, Darkgreen, Blue, Mediumblue, Darkblue);

  subtype Int_Color is Integer range 0 .. 255;

  type RGB is record
    Red : Int_Color;
    Green : Int_Color;
    Blue : Int_Color;
  end record;

  function To_RGB (C : HTML_Color) return RGB;
  function Image (C : RGB) return String;

  type HTML_Color_RGB is array (HTML_Color) of RGB;

  To_RGB.Lookup_Table := (Salmon => (16#FA#, 16#80#, 16#72#),
                          Firebrick => (16#B2#, 16#22#, 16#22#),
                          Red => (16#FF#, 16#00#, 16#00#),
                          Darkred => (16#8B#, 16#00#, 16#00#),
                          Lime => (16#00#, 16#FF#, 16#00#),
                          Forestgreen => (16#22#, 16#8B#, 16#22#),
                          Green => (16#00#, 16#80#, 16#00#),
                          Darkgreen => (16#00#, 16#64#, 16#00#),
                          Blue => (16#00#, 16#00#, 16#FF#),

  (continues on next page)
Listing 62: color_types.adb

```ada
with Ada.Integer_Text_IO;
package body Color_Types is

function To_RGB (C : HTML_Color) return RGB is
begin
return To_RGB_Lookup_Table (C);
end To_RGB;

function Image (C : RGB) return String is
subtype Str_Range is Integer range 1 .. 10;
SR : String (Str_Range);
SG : String (Str_Range);
SB : String (Str_Range);
begin
Ada.Integer_Text_IO.Put (To => SR,
Item => C.Red,
Base => 16);
Ada.Integer_Text_IO.Put (To => SG,
Item => C.Green,
Base => 16);
Ada.Integer_Text_IO.Put (To => SB,
Item => C.Blue,
Base => 16);
return "(Red => " & SR
& ", Green => " & SG
& ", Blue => " & SB
& ")";
end Image;
end Color_Types;
```

Listing 63: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
	type Test_Case_Index is
	(Color_Table_Chk,
HTML_Color_To_Integer_Chk);

type Color_Table_Type is
	(Hex, HTML_Color);

type HTML_Color is
	(Accessor => HTML_Color_Type);

procedure Check (TC : Test_Case_Index) is
begin
case TC is
when Color_Table_Chk =>
Put_Line ("Size of HTML_Color_RGB: 
" & Integer'Image (HTML_Color_RGB'Length));
when HTML_Color_To_Integer_Chk =>
for I in HTML_Color'Range loop
```

(continues on next page)
Learning Ada

(continued from previous page)

```ada
Put_Line (HTML_Color'Image (I) & " => " & Image (To_RGB (I)) & ".");
end loop;
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
```

### 118.6.3 Unconstrained Array

#### Listing 64: unconstrained_arrays.ads

```ada
package Unconstrained_Arrays is

  type My_Array is array (Positive range <>) of Integer;

  procedure Init (A : in out My_Array);
  function Init (I, L : Positive) return My_Array;
  procedure Double (A : in out My_Array);
  function Diff_Prev_Elem (A : My_Array) return My_Array;

end Unconstrained_Arrays;
```

#### Listing 65: unconstrained_arrays.adb

```ada
package body Unconstrained_Arrays is

  procedure Init (A : in out My_Array) is
    Y : Natural := A'Last;
  begin
    for I in A'Range loop
      A (I) := Y;
      Y := Y - 1;
    end loop;
  end Init;

  function Init (I, L : Positive) return My_Array is
    A : My_Array (I .. I + L - 1);
  begin
    Init (A);
    return A;
  end Init;

  procedure Double (A : in out My_Array) is
  begin
    for I in A'Range loop
```

(continues on next page)
A (I) := A (I) * 2;
end loop;
end Double;

function Diff_Prev_Elem (A : My_Array) return My_Array is
   A_Out : My_Array (A'Range);
begin
   A_Out (A'First) := 0;
   for I in A'First + 1 .. A'Last loop
      A_Out (I) := A (I) - A (I - 1);
   end loop;
   return A_Out;
end Diff_Prev_Elem;
end Unconstrained_Arrays;

Listing 66: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Unconstrained_Arrays; use Unconstrained_Arrays;

procedure Main is
   type Test_Case_Index is
      (Init_Chk,
       Init_Proc_Chk,
       Double_Chk,
       Diff_Prev_Chk,
       Diff_Prev_Single_Chk);

   procedure Check (TC : Test_Case_Index) is
      AA : My_Array (1 .. 5);
      AB : My_Array (5 .. 9);
      begin
         case TC is
            when Init_Chk =>
               AA := Init (AA'First, AA'Length);
               AB := Init (AB'First, AB'Length);
               Display (AA);
               Display (AB);
            when Init_Proc_Chk =>
               Init (AA);
               Init (AB);
               Display (AA);
               Display (AB);
            when Double_Chk =>
               Local_Init (A : in out My_Array) is
                  begin
                     A := (1, 2, 5, 10, -10);
                  end Local_Init;
         end case;
      end Check;

   procedure Display (A : My_Array) is
      begin
         for I in A'Range loop
            Put_Line (Integer'Image (A (I)));
         end loop;
      end Display;

   procedure Local_Init (A : in out My_Array) is
      begin
         A := (1, 2, 5, 10, -10);
      end Local_Init;

begin
      case TC is
         when Init_Chk =>
            AA := Init (AA'First, AA'Length);
            AB := Init (AB'First, AB'Length);
            Display (AA);
            Display (AB);
         when Init_Proc_Chk =>
            Init (AA);
            Init (AB);
            Display (AA);
            Display (AB);
         when Double_Chk =>

         end case;
      end;

(continues on next page)
Local_Init (AB);
Double (AB);
Display (AB);

when Diff_Prev_Chk =>
Local_Init (AB);
AB := Diff_Prev_Elem (AB);
Display (AB);

when Diff_Prev_Single_Chk =>

begin
A1 := Diff_Prev_Elem (A1);
Display (A1);
end;
edn case;
end Check;

begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting...");
return;
elsif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

### 118.6.4 Product info

Listing 67: product_info_pkg.ads

```ada
package Product_Info_Pkg is

  subtype Quantity is Natural;
  subtype Currency is Float;

  type Product_Info is record
    Units : Quantity;
    Price : Currency;
  end record;

  type ProductInfos is array (Positive range <>) of Product_Info;
  type Currency_Array is array (Positive range <>) of Currency;

  procedure Total (P : ProductInfos;
                  Tot : out Currency_Array);

  function Total (P : ProductInfos) return Currency_Array;

  function Total (P : ProductInfos) return Currency;

end Product_Info_Pkg;
```

### 118.6. Arrays

1987
package body Product_Info_Pkg is

   -- Get total for single product
   function Total (P : Product_Info) return Currency is
      (Currency (P.Units) * P.Price);
   procedure Total (P : Product_Infos;
      Tot : out Currency_Array) is
      begin
         for I in P'Range loop
            Tot (I) := Total (P (I));
         end loop;
      end Total;

   function Total (P : Product_Infos) return Currency_Array is
      Tot : Currency_Array (P'Range);
      begin
         Total (P, Tot);
         return Tot;
   end Total;

   function Total (P : Product_Infos) return Currency is
      Tot : Currency := 0.0;
      begin
         for I in P'Range loop
            Tot := Tot + Total (P (I));
         end loop;
         return Tot;
   end Total;

end Product_Info_Pkg;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Product_Info_Pkg; use Product_Info_Pkg;

procedure Main is
   package Currency_IO is new Ada.Text_IO.Float_IO (Currency);
   type Test_Case_Index is
      (Total_Func_Chk, Total_Proc_Chk, Total_Value_Chk);
   procedure Check (TC : Test_Case_Index) is
      subtype Test_Range is Positive range 1 .. 5;
      P : Product_Infos (Test_Range);
      Tots : Currency_Array (Test_Range);
      Tot : Currency;

      procedure Display (Tots : Currency_Array) is
         begin
            for I in Tots'Range loop
               (continues on next page)
24  Currency_IO.Put (Tots (I));
25  New_Line;
26  end loop;
27  end Display;

28  procedure Local_Init (P : in out ProductInfos) is
29  begin
30      P := ((1, 0.5),
31             (2, 10.0),
32             (5, 40.0),
33             (10, 10.0),
34             (10, 20.0));
35  end Local_Init;

36  begin
37      Currency_IO.Default_Fore := 1;
38      Currency_IO.Default_Aft := 2;
39      Currency_IO.Default_Exp := 0;
40      case TC is
41          when Total.Func_Chk =>
42              Local_Init (P);
43              Tots := Total (P);
44              Display (Tots);
45          when Total.Proc_Chk =>
46              Local_Init (P);
47              Total (P, Tots);
48              Display (Tots);
49          when Total.Value.Chk =>
50              Local_Init (P);
51              Tot := Total (P);
52              Currency_IO.Put (Tot);
53              New_Line;
54      end case;
55  end case;
56  end Check;
57
58  begin
59      if Argument_Count < 1 then
60          Put_Line ("ERROR: missing arguments! Exiting...");
61          return;
62      elsif Argument_Count > 1 then
63          Put_Line ("Ignoring additional arguments...");
64      end if;
65
66      Check (Test_Case_Index'Value (Argument (1)));
67  end Main;

118.6.5 String_10

Listing 70: strings_10.ads

package Strings_10 is

3    subtype String_10 is String (1 .. 10);
4
5    -- Using "type String_10 is..." is possible, too.
6
7    function To_String_10 (S : String) return String_10;
8
(continues on next page)
end Strings_10;

Listing 71: strings_10.adb

package body Strings_10 is

function To_String_10 (S : String) return String_10 is
  S_Out : String_10;
begin
  for I in String_10'First .. Integer'Min (String_10'Last, S'Last) loop
    S_Out (I) := S (I);
  end loop;

  for I in Integer'Min (String_10'Last + 1, S'Last + 1) .. String_10'Last loop
    S_Out (I) := ' ';
  end loop;

  return S_Out;
end To_String_10;
end Strings_10;

Listing 72: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Strings_10; use Strings_10;

procedure Main is

  type Test_Case_Index is
    (String_10_Long_Chk,
     String_10_Short_Chk);

  procedure Check (TC : Test_Case_Index) is
    SL : constant String := "And this is a long string just for testing...";
    SS : constant String := "Hey!";
    S_10 : String_10;

begin
  case TC is
    when String_10_Long_Chk =>
      S_10 := To_String_10 (SL);
      Put_Line (String (S_10));
    when String_10_Short_Chk =>
      S_10 := (others => ' ');
      S_10 := To_String_10 (SS);
      Put_Line (String (S_10));
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'Value (Argument (1)));
end Main;
118.6.6 List of Names

Listing 73: names_ages.ads

package Names_Ages is

Max_People : constant Positive := 10;

subtype Name_Type is String (1 .. 50);

type Age_Type is new Natural;

type Person is record
    Name : Name_Type;
    Age  : Age_Type;
end record;

type People_Array is array (Positive range <>) of Person;

type People is record
    People_A : People_Array (1 .. Max_People);
    Last_Valid : Natural;
end record;

procedure Reset (P : in out People);

procedure Add (P   : in out People;
               Name : String);

function Get (P : People;
              Name : String) return Age_Type;

procedure Update (P : in out People;
                  Name : String;
                  Age  : Age_Type);

procedure Display (P : People);
end Names_Ages;

Listing 74: names_ages.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Names_Ages is

function To_Name_Type (S : String) return Name_Type is
    S_Out : Name_Type := (others => ' ');
    begin
        for I in 1 .. Integer'Min (S'Last, Name_Type'Last) loop
            S_Out (I) := S (I);
        end loop;
        return S_Out;
    end To_Name_Type;

procedure Init (P   : in out Person;
               Name : String) is
        begin
        end Init;
(continues on next page)
P.Name := To_Name_Type (Name);
P.Age := 0;
end Init;

function Match (P : Person; Name : String) return Boolean is
begin
  return P.Name = To_Name_Type (Name);
end Match;

function Get (P : Person) return Age_Type is
begin
  return P.Age;
end Get;

procedure Update (P : in out Person; Age : Age_Type) is
begin
  P.Age := Age;
end Update;

procedure Display (P : Person) is
begin
  Put_Line ("NAME: " & Trim (P.Name, Right));
  Put_Line ("AGE: " & Age_Type'Image (P.Age));
end Display;

procedure Reset (P : in out People) is
begin
  P.Last_Valid := 0;
end Reset;

procedure Add (P : in out People; Name : String) is
begin
  P.Last_Valid := P.Last_Valid + 1;
  Init (P.People_A (P.Last_Valid), Name);
end Add;

function Get (P : People; Name : String) return Age_Type is
begin
  for I in P.People_A'First .. P.Last_Valid loop
    if Match (P.People_A (I), Name) then
      return Get (P.People_A (I));
    end if;
  end loop;
  return 0;
end Get;

procedure Update (P : in out People; Name : String; Age : Age_Type) is
begin
  for I in P.People_A'First .. P.Last_Valid loop
    if Match (P.People_A (I), Name) then
      Update (P.People_A (I), Age);
    end if;
  end loop;
end Update;
procedure Display (P : People) is
begin
    Put_Line ("LIST OF NAMES:");
    for I in P.People_A'First .. P.Last_Valid loop
        Display (P.People_A (I));
    end loop;
end Display;
end Names_Ages;

Listing 75: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Names_Ages; use Names_Ages;

procedure Main is
    type Test_Case_Index is
        (Names_Ages_Chk,
         Get_Age_Chk);

    procedure Check (TC : Test_Case_Index) is
        P : People;
    begin
        case TC is
            when Names_Ages_Chk =>
                Reset (P);
                Add (P, "John");
                Add (P, "Patricia");
                Add (P, "Josh");
                Display (P);
                Update (P, "John", 18);
                Update (P, "Patricia", 35);
                Update (P, "Josh", 53);
                Display (P);
            when Get_Age_Chk =>
                Reset (P);
                Add (P, "Peter");
                Update (P, "Peter", 45);
                Put_Line ("Peter is " & Age_Type'Image (Get (P, "Peter")) & " years old.");
        end case;
    end Check;
begin
    if Argument_Count < 1 then
        Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elseif Argument_Count > 1 then
        Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
end Main;
118.7 More About Types

118.7.1 Aggregate Initialization

Listing 76: aggregates.ads

```ada
package Aggregates is

   type Rec is record
      W : Integer := 10;
      X : Integer := 11;
      Y : Integer := 12;
      Z : Integer := 13;
   end record;

   type Int_Arr is array (1 .. 20) of Integer;

   procedure Init (R : out Rec);
   procedure Init_Some (A : out Int_Arr);
   procedure Init (A : out Int_Arr);
end Aggregates;
```

Listing 77: aggregates.adb

```ada
package body Aggregates is

   procedure Init (R : out Rec) is
      begin
         R := (X => 100,
               Y => 200,
               others => <>);
      end Init;

   procedure Init_Some (A : out Int_Arr) is
      begin
         A := (1..5 => 99,
               others => 100);
      end Init_Some;

   procedure Init (A : out Int_Arr) is
      begin
         A := (others => 5);
      end Init;
end Aggregates;
```

Listing 78: main.adb

```ada
with Ada.Command_Line;   use Ada.Command_Line;
with Ada.Text_IO;       use Ada.Text_IO;
with Aggregates;        use Aggregates;

procedure Main is
   -- Remark: the following line is not relevant.
   F : array (1 .. 10) of Float := (others => 42.42)
      with Unreferenced;
```

(continues on next page)
type Test_Case_Index is
  (Default_Rec_Chk,
   Init_Rec_Chk,
   Init_Some_Arr_Chk,
   Init_Arr_Chk);

procedure Check (TC : Test_Case_Index) is
  A : Int_Arr;
  R : Rec;
  DR : constant Rec := (others => <>);
begin
  case TC is
    when Default_Rec_Chk =>
      R := DR;
      Put_Line ("Record Default:");
      Put_Line ("W => ", Integer'Image (R.W));
      Put_Line ("X => ", Integer'Image (R.X));
      Put_Line ("Y => ", Integer'Image (R.Y));
      Put_Line ("Z => ", Integer'Image (R.Z));
    when Init_Rec_Chk =>
      Init (R);
      Put_Line ("Record Init:");
      Put_Line ("W => ", Integer'Image (R.W));
      Put_Line ("X => ", Integer'Image (R.X));
      Put_Line ("Y => ", Integer'Image (R.Y));
      Put_Line ("Z => ", Integer'Image (R.Z));
    when Init_Some_Arr_Chk =>
      Init_Some (A);
      for I in A'Range loop
        Put_Line ("Array Init_Some:", Integer'Image (I) & "
                  & Integer'Image (A (I)));
      end loop;
    when Init_Arr_Chk =>
      Init (A);
      for I in A'Range loop
        Put_Line ("Array Init:", Integer'Image (I) & "
                  & Integer'Image (A (I)));
      end loop;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
118.7.2 Versioning

Listing 79: versioning.ads

```ada
package Versioning is

    type Version is record
        Major : Natural;
        Minor : Natural;
        Maintenance : Natural;
    end record;

    function Convert (V : Version) return String;

    function Convert (V : Version) return Float;

end Versioning;
```

Listing 80: versioning.adb

```ada
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Versioning is

    function Image_Trim (N : Natural) return String is
        S_N : constant String := Trim (Natural'Image (N), Left);
        begin
            return S_N;
        end Image_Trim;

    function Convert (V : Version) return String is
        S_Major : constant String := Image_Trim (V.Major);
        S_Minor : constant String := Image_Trim (V.Minor);
        S_Maint : constant String := Image_Trim (V.Maintenance);
        begin
            return (S_Major & "." & S_Minor & "." & S_Maint);
        end Convert;

    function Convert (V : Version) return Float is
        begin
            return Float (V.Major) + (Float (V.Minor) / 10.0);
        end Convert;

end Versioning;
```

Listing 81: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Versioning; use Versioning;

procedure Main is

    type Test_Case_Index is
        (Ver_String_Chk,
         Ver_Float_Chk);

    procedure Check (TC : Test_Case_Index) is
        V : constant Version := (1, 3, 23);
        begin
            (continues on next page)
```
Learning Ada

118.7.3 Simple todo list

Listing 82: todo_lists.ads

```ada
package Todo_Lists is

    type Todo_Item is access String;

    type Todo_Items is array (Positive range <>) of Todo_Item;

    type Todo_List (Max.Len : Natural) is record
        Items : Todo_Items (1 .. Max.Len);
        Last : Natural := 0;
    end record;

    procedure Add (Todos : in out Todo_List;
                    Item       : String);

    procedure Display (Todos : Todo_List);

end Todo_Lists;
```

Listing 83: todo_lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Todo_Lists is

    procedure Add (Todos : in out Todo_List;
                   Item       : String) is

    begin
        if Todos.Last < Todos.Items.Last then
            Todos.Last := Todos.Last + 1;
            Todos.Items (Todos.Last) := new String' (Item);
        else
            Put_Line ("ERROR: list is full!");
        end if;
    end Add;
```

(continues on next page)
procedure Display (Todos : Todo_List) is
begin
  Put_Line ("TO-DO LIST");
  for I in Todos.Items'First .. Todos.Last loop
    Put_Line (Todos.Items (I).all);
  end loop;
end Display;
end Todo_Lists;

Listing 84: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is
  type Test_Case_Index is
    (Todo_List_Chk);

  procedure Check (TC : Test_Case_Index) is
    T : Todo_List (10);
  begin
    case TC is
      when Todo_List_Chk =>
        Add (T, "Buy milk");
        Add (T, "Buy tea");
        Add (T, "Buy present");
        Add (T, "Buy tickets");
        Add (T, "Pay electricity bill");
        Add (T, "Schedule dentist appointment");
        Add (T, "Call sister");
        Add (T, "Revise spreadsheet");
        Add (T, "Edit entry page");
        Add (T, "Select new design");
        Add (T, "Create upgrade plan");
        Display (T);
      end case;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
  return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'Value (Argument (1)));
end Main;
118.7.4 Price list

Listing 85: price_lists.ads

```ada
package Price_Lists is

  type Price_Type is delta 0.01 digits 12;

  type Price_List_Array is array (Positive range <>) of Price_Type;

  type Price_List (Max : Positive) is record
    List : Price_List_Array (1 .. Max);
    Last : Natural := 0;
  end record;

  type Price_Result (Ok : Boolean) is record
    case Ok is
      when False =>
        null;
      when True =>
        Price : Price_Type;
    end case;
  end record;

  procedure Reset (Prices : in out Price_List);

  procedure Add (Prices : in out Price_List;
    Item : Price_Type);

  function Get (Prices : Price_List;
    Idx : Positive) return Price_Result;

  procedure Display (Prices : Price_List);

end Price_Lists;
```

Listing 86: price_lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Price_Lists is

  procedure Reset (Prices : in out Price_List) is
  begin
    Prices.Last := 0;
  end Reset;

  procedure Add (Prices : in out Price_List;
    Item : Price_Type) is
  begin
    if Prices.Last < Prices.List'Last then
      Prices.Last := Prices.Last + 1;
      Prices.List (Prices.Last) := Item;
    else
      Put_Line ("ERROR: list is full!");
    end if;
  end Add;

  function Get (Prices : Price_List;
    Idx : Positive) return Price_Result is
  begin
    (continues on next page)
  ```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Price_Lists; use Price_Lists;

procedure Main is
  type Test_Case_Index is
    (Price_Type_Chk, Price_List_Chk, Price_List_Get_Chk);

  procedure Check (TC : Test_Case_Index) is
    L : Price_List (10);

  procedure Local_Init_List is
    begin
      Reset (L);
      Add (L, 1.45);
      Add (L, 2.37);
      Add (L, 3.21);
      Add (L, 4.14);
      Add (L, 5.22);
      Add (L, 6.69);
      Add (L, 7.77);
      Add (L, 8.14);
      Add (L, 9.99);
      Add (L, 10.01);
    end Local_Init_List;

  procedure Get_Display (Idx : Positive) is
    R : constant Price_Result := Get (L, Idx);
    begin
      Put_Line ("Attempt Get # " & Positive'Image (Idx));
      if R.Ok then
        Put_Line ("Element # " & Positive'Image (Idx) & ", Price => " & Price_Type'Image (R.Price));
      else
        declare
          begin
            Put_Line ("Element # " & Positive'Image (Idx) & ", Price => False");
        end begin;
    end Get_Display;

begin
  with Ada.Command_Line; use Ada.Command_Line;
  with Ada.Text_IO; use Ada.Text_IO;
  with Price_Lists; use Price_Lists;

  procedure Display (Prices : Price_List) is
    begin
      Put_Line ("PRICE LIST");
      for I in Prices.List'First .. Prices.Last loop
        Put_Line (Price_Type'Image (Prices.List (I)));
      end loop;
    end Display;

  end Price_Lists;

Listing 87: main.adb

if (Idx >= Prices.List'First and then
  Idx <= Prices.Last) then
  return Price_Result'(Ok => True,
    Price => Prices.List (Idx));
else
  return Price_Result'(Ok => False);
end if;

end Get;

procedure Display (Prices : Price_List) is
begin
  Put_Line ("PRICE LIST");
  for I in Prices.List'First .. Prices.Last loop
    Put_Line (Price_Type'Image (Prices.List (I)));
  end loop;
end Display;

end Price_Lists;

Listing 87: main.adb

if (Idx >= Prices.List'First and then
  Idx <= Prices.Last) then
  return Price_Result'(Ok => True,
    Price => Prices.List (Idx));
else
  return Price_Result'(Ok => False);
end if;

end Get;

procedure Display (Prices : Price_List) is
begin
  Put_Line ("PRICE LIST");
  for I in Prices.List'First .. Prices.Last loop
    Put_Line (Price_Type'Image (Prices.List (I)));
  end loop;
end Display;

end Price_Lists;
exception
when others =>
  Put_Line ("Element not available (as expected)");
end if;
end Get_Display;

begin
  case TC is
    when Price_Type_Chk =>
      Put_Line ("The delta value of Price_Type is 
        & Price_Type'Image (Price_Type'Delta) & ");
      Put_Line ("The minimum value of Price_Type is 
        & Price_Type'Image (Price_Type'First) & ");
      Put_Line ("The maximum value of Price_Type is 
        & Price_Type'Image (Price_Type'Last) & ");
    when Price_List_Chk =>
      Local_Init_List;
      Display (L);
      when Price_List_Get_Chk =>
        Local_Init_List;
        Get_Display (5);
        Get_Display (40);
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.8 Privacy

118.8.1 Directions

Listing 88: directions.ads

package Directions is
  type Angle_Mod is mod 360;
  type Direction is
    (North, Northwest, West, Southwest, South, Southeast, East);
function To_Direction (N : Angle_Mod) return Direction;

type Ext_Angle is private;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;

procedure Display (N : Ext_Angle);

Listing 89: directions.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Directions is

procedure Display (N : Ext_Angle) is
begin
   Put_Line ("Angle: "
   & Angle_Mod'Image (N.Angle_Elem)
   & " => "
   & Direction'Image (N.Direction_Elem)
   & ".");
end Display;

function To_Direction (N : Angle_Mod) return Direction is
begin
    case N is
    when 0 => return East;
    when 1 .. 89 => return Northwest;
    when 90 => return North;
    when 91 .. 179 => return Northwest;
    when 180 => return West;
    when 181 .. 269 => return Southwest;
    when 270 => return South;
    when 271 .. 359 => return Southeast;
    end case;
end To_Direction;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
begin
    return (Angle_Elem => N,
        Direction_Elem => To_Direction (N));
end To_Ext_Angle;
end Directions;

Listing 90: test_directions.adb

with Directions; use Directions;

procedure Test_Directions is (continues on next page)
type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;

AllDirections : constant Ext_Angle_Array (1 .. 6) := (To Ext_Angle (0),
To Ext_Angle (45),
To Ext_Angle (90),
To Ext_Angle (91),
To Ext_Angle (180),
To Ext_Angle (270));

begin
for I in AllDirections'Range loop
Display (AllDirections (I));
end loop;
end TestDirections;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Test_Directions;

procedure Main is

  type Test_Case_Index is
    (Direction_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
    when Direction_Chk =>
      Test_Directions;
    end case;
  end Check;

begin
if Argument_Count < 1 then
  Put_Line ("ERROR: missing arguments! Exiting...");
  return;
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.8.2 Limited Strings

package Limited_Strings is
  type Lim_String is limited private;
  function Init (S : String) return Lim_String;
  function Init (Max : Positive) return Lim_String;
end Limited_Strings;
procedure Put_Line (LS : Lim_String);
procedure Copy (From : Lim_String;
            To : in out Lim_String);
function "=" (Ref, Dut : Lim_String) return Boolean;

private

type Lim_String is access String;

end Limited_Strings;

Listing 93: limited_strings.adb

with Ada.Text_IO;

package body Limited_Strings is

function Init (S : String) return Lim_String is
    LS : constant Lim_String := new String' (S);
begin
    return LS;
end Init;

function Init (Max : Positive) return Lim_String is
    LS : constant Lim_String := new String (1 .. Max);
begin
    LS.all := (others => '_');
    return LS;
end Init;

procedure Put_Line (LS : Lim_String) is
begin
    Ada.Text_IO.Put_Line (LS.all);
end Put_Line;

function Get_Min_Last (A, B : Lim_String) return Positive is
begin
    return Positive' Min (A'Last, B'Last);
end Get_Min_Last;

procedure Copy (From : Lim_String;
            To : in out Lim_String) is
    Min_Last : constant Positive := Get_Min_Last (From, To);
begin
    To (To' First .. Min_Last) := From (To'First .. Min_Last);
    To (Min_Last + 1 .. To'Last) := (others => '_');
end;

function "=" (Ref, Dut : Lim_String) return Boolean is
begin
    for I in Dut'First .. Min_Last loop
        if Dut (I) /= Ref (I) then
            return False;
        end if;
    end loop;
end;
return True;
end;
end Limited_Strings;

Listing 94: check_lim_string.adb

with Ada.Text_IO; use Ada.Text_IO;
with Limited_Strings; use Limited_Strings;

procedure Check_Lim_String is
  S : constant String := "----------";
  S1 : constant Lim_String := Init ("Hello World");
  S2 : constant Lim_String := Init (30);
  S3 : Lim_String := Init (5);
  S4 : Lim_String := Init (S & S & S);
begin
  Put ("S1 => ");
  Put_Line (S1);
  Put ("S2 => ");
  Put_Line (S2);
  if S1 = S2 then
    Put_Line ("S1 is equal to S2.");
  else
    Put_Line ("S1 isn't equal to S2.");
  end if;
  Copy (From => S1, To => S3);
  Put ("S3 => ");
  Put_Line (S3);
  if S1 = S3 then
    Put_Line ("S1 is equal to S3.");
  else
    Put_Line ("S1 isn't equal to S3.");
  end if;
  Copy (From => S1, To => S4);
  Put ("S4 => ");
  Put_Line (S4);
  if S1 = S4 then
    Put_Line ("S1 is equal to S4.");
  else
    Put_Line ("S1 isn't equal to S4.");
  end if;
end Check_Lim_String;

Listing 95: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Check_Lim_String;

procedure Main is
  type Test_Case_Index is
    (Lim_String_Chk);
(continues on next page)
procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Lim_String_Chk =>
      Check_Lim_String;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.9 Generics

118.9.1 Display Array

Listing 96: display_array.ads

generic
  type T_Range is range <>;
  type T_Element is private;
  type T_Array is array (T_Range range <>) of T_Element;
  with function Image (E : T_Element) return String;
procedure Display_Array (Header : String;
                        A : T_Array);

Listing 97: display_array.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Display_Array (Header : String;
                         A : T_Array) is
begin
  Put_Line (Header);
  for I in A'Range loop
    Put_Line (T_Range'Image (I) & " : " & Image (A (I)));
  end loop;
end Display_Array;

Listing 98: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Display_Array;

procedure Main is
  type Test_Case_Index is (Int_Array_Chk,
procedure Test_Int_Array is
  type Int_Array is array (Positive range <>) of Integer;
  procedure Display_Int_Array is new Display_Array (T_Range => Positive,
                                                   T_Element => Integer,
                                                   T_Array => Int_Array,
                                                   Image => Integer'Image);

  A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
begin
  Display_Int_Array ("Integers", A);
end Test_Int_Array;

procedure Test_Point_Array is
  type Point is record
    X : Float;
    Y : Float;
  end record;
  type Point_Array is array (Natural range <>) of Point;
  function Image (P : Point) return String is
begin
  return "(" & Float'Image (P.X) & ", " & Float'Image (P.Y) & ")";
end Image;
  procedure Display_Point_Array is new Display_Array (T_Range => Natural,
                                                      T_Element => Point,
                                                      T_Array => Point_Array,
                                                      Image => Image);

  A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
                                       (5.0, 2.0), (-0.5, 2.0));
begin
  Display_Point_Array ("Points", A);
end Test_Point_Array;

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Int_Array_Chk =>
      Test_Int_Array;
    when Point_Array_Chk =>
      Test_Point_Array;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'Value (Argument (1)));
118.9.2 Average of Array of Float

Listing 99: average.ads

```ada
generic
  type T_Range is range <>;
  type T_Element is digits <>;
  type T_Array is array (T_Range range <>) of T_Element;
function Average (A : T_Array) return T_Element;
```

Listing 100: average.adb

```ada
function Average (A : T_Array) return T_Element is
  Acc : Float := 0.0;
begin
  for I in A'Range loop
    Acc := Acc + Float (A (I));
  end loop;
  return T_Element (Acc / Float (A'Length));
end Average;
```

Listing 101: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Average;

procedure Main is
  type Test_Case_Index is (Float_Array_Chk, Digits_7_Float_Array_Chk);

  procedure Test_Float_Array is
    type Float_Array is array (Positive range <>) of Float;

    function Average_Float is new
      Average (T_Range => Positive,
               T_Element => Float,
               T_Array => Float_Array);

    A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
  begin
    Put_Line ("Average: " & Float'Image (Average_Float (A)));
  end Test_Float_Array;

  procedure Test_Digits_7_Float_Array is
    type Custom_Float is digits 7 range 0.0 .. 1.0;

    type Float_Array is array (Integer range <>) of Custom_Float;

    function Average_Float is new
      Average (T_Range => Integer,
               T_Element => Custom_Float,
               T_Array => Float_Array);
```
A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
begin
  Put_Line ("Average: " & Custom_Float'Image (Average_Float (A)));
end Test_Digits_7_Float_Array;

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
  when Float_Array_Chk =>
    Test_Float_Array;
  when Digits_7_Float_Array_Chk =>
    Test_Digits_7_Float_Array;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.9.3 Average of Array of Any Type

Listing 102: average.ads

<table>
<thead>
<tr>
<th>generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>type T_Range is range &lt;&gt;;</td>
</tr>
<tr>
<td>type T_Element is private;</td>
</tr>
<tr>
<td>type T_Array is array (T_Range range &lt;&gt;) of T_Element;</td>
</tr>
<tr>
<td>with function To_Float (E : T_Element) return Float is &lt;&gt;;</td>
</tr>
<tr>
<td>function Average (A : T_Array) return Float is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>Acc := 0.0;</td>
</tr>
<tr>
<td>end loop;</td>
</tr>
<tr>
<td>return Acc / Float (A'Length);</td>
</tr>
<tr>
<td>end Average;</td>
</tr>
</tbody>
</table>

Listing 103: average.adb

function Average (A : T_Array) return Float is
  Acc : Float := 0.0;
begin
  for I in A'Range loop
    Acc := Acc + To_Float (A (I));
  end loop;
  return Acc / Float (A'Length);
end Average;

Listing 104: test_item.ads

procedure Test_Item;
Listing 105: test_item.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Average;

procedure Test_Item is
  package F_IO is new Ada.Text_IO.Float_IO (Float);

type Amount is delta 0.01 digits 12;

type Item is record
  Quantity : Natural;
  Price : Amount;
end record;

type Item_Array is
  array (Positive range <>) of Item;

function Get_Total (I : Item) return Float is
  (Float (I.Quantity) * Float (I.Price));

function Get_Price (I : Item) return Float is
  (Float (I.Price));

function Average_Total is new Average (T_Range => Positive,
  T_Element => Item,
  T_Array => Item_Array,
  To_Float => Get_Total);

function Average_Price is new Average (T_Range => Positive,
  T_Element => Item,
  T_Array => Item_Array,
  To_Float => Get_Price);

A : constant Item_Array (1 .. 4)
 := ((Quantity => 5, Price => 10.00),
  (Quantity => 80, Price => 2.50),
  (Quantity => 40, Price => 5.00),
  (Quantity => 20, Price => 12.50));

begin
  Put ("Average per item & quantity: ");
  F_IO.Put (Average_Total (A), 3, 2, 0);
  New_Line;

  Put ("Average price: ");
  F_IO.Put (Average_Price (A), 3, 2, 0);
  New_Line;
end Test_Item;
```

Listing 106: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Average;

procedure Main is
```

(continues on next page)
type Test_Case_Index is (Item_Array_CHK);

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Item_Array_CHK =>
      Test_Item;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.9.4 Generic list

Listing 107: gen_list.ads

generic
  type Item is private;
type Items is array (Positive range <>) of Item;
Name : String;
List_Array : in out Items;
Last : in out Natural;
with procedure Put (I : Item) is <>;
package Gen_List is
  procedure Init;

  procedure Add (I : Item;
    Status : out Boolean);

  procedure Display;
end Gen_List;

Listing 108: gen_list.adb

with Ada.Text_IO; use Ada.Text_IO;
package body Gen_List is
  procedure Init is
    begin
      Last := List_Array'First - 1;
    end Init;

  procedure Add (I : Item;
    Status : out Boolean) is
    begin
      Status := Last < List_Array'Last;
    end Add;
end Gen_List;
if Status then
  Last := Last + 1;
  List_Array (Last) := I;
end if;
end Add;

procedure Display is
begin
  Put_Line (Name);
  for I in List_Array'First .. Last loop
    Put (List_Array (I));
    New_Line;
  end loop;
end Display;
end Gen_List;

Listing 109: test_int.ads

procedure Test_Int;

Listing 110: test_int.adb

with Ada.Text_IO; use Ada.Text_IO;
with Gen_List;

procedure Test_Int is
  procedure Put (I : Integer) is
    begin
      Ada.Text_IO.Put (Integer'Image (I));
    end Put;

type Integer_Array is array (Positive range <>) of Integer;

A : Integer_Array (1 .. 3);
L : Natural;

package Int_List is new
  Gen_List (Item => Integer,
            Items => Integer_Array,
            Name => "List of integers",
            List_Array => A,
            Last => L);

Success : Boolean;

procedure Display_Add_Success (Success : Boolean) is
begin
  if Success then
    Put_Line ("Added item successfully!");
  else
    Put_Line ("Couldn't add item!");
  end if;
end Display_Add_Success;

begin
  (continues on next page)
Int_List.Init;
Int_List.Add (2, Success);
Display_Add_Success (Success);
Int_List.Add (5, Success);
Display_Add_Success (Success);
Int_List.Add (7, Success);
Display_Add_Success (Success);
Int_List.Add (8, Success);
Display_Add_Success (Success);
Int_List.Display;
end Test_Int;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Test_Int;

procedure Main is
  type Test_Case_Index is (Int_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Int_Chk =>
        Test_Int;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.10 Exceptions

118.10.1 Uninitialized Value

package Options is
  type Option is (Uninitialized,
                 Option_1,
                 Option_2,
                 (continues on next page))
Learning Ada

Option_3);

Uninitialized_Value : exception;

function Image (O : Option) return String;

end Options;

Listing 113: options.adb

package body Options is

function Image (O : Option) return String is
begin
  case O is
  when Uninitialized =>
    raise Uninitialized_Value with "Uninitialized value detected!";
  when others =>
    return Option'Image (O);
  end case;
end Image;
end Options;

Listing 114: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Options; use Options;

procedure Main is
  type Test_Case_Index is
    (Options.chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check (O : Option) is
      begin
        Put_Line (Image (O));
        exception
          when E : Uninitialized_Value =>
            Put_Line (Exception_Message (E));
          end Check;
      begin
        case TC is
          when Options.chk =>
            for O in Option loop
              Check (O);
            end loop;
        end case;
      end Check;
    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        (continues on next page)
Learning Ada

(continued from previous page)

35 | Put_Line ("Ignoring additional arguments..."));
36 |   end if;
37 |   Check (Test_Case_Index'Value (Argument (1)));
38 | end Main;

118.10.2 Numerical Exception

Listing 115: tests.ads

package Tests is

  type Test_ID is (Test_1, Test_2);

  Custom_Exception : exception;

  procedure Num_Exception_Test (ID : Test_ID);

end Tests;

Listing 116: tests.adb

package body Tests is

  pragma Warnings (Off, "variable ""C"" is assigned but never read");

  procedure Num_Exception_Test (ID : Test_ID) is
    A, B, C : Integer;
    begin
      case ID is
        when Test_1 =>
          A := Integer'Last;
          B := Integer'Last;
          C := A + B;
        when Test_2 =>
          raise Custom_Exception with "Custom_Exception raised!";
      end case;
    end Num_Exception_Test;

  pragma Warnings (On, "variable ""C"" is assigned but never read");

end Tests;

Listing 117: check_exception.adb

with Tests;       use Tests;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Check_Exception (ID : Test_ID) is
  begin
    Num_Exception_Test (ID);
  exception
    when Constraint_Error =>
      Put_Line ("Constraint_Error detected!");
    when E : others =>
      Put_Line (Exception_Message (E));

(continues on next page)
end Check_Exception;

Listing 118: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;
with Check_Exception;

procedure Main is
  type Test_Case_Index is
    (Exception_1_Chk, Exception_2_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_Handle_Exception (ID : Test_ID) is
      begin
        Check_Exception (ID);
        exception
          when Constraint_Error =>
            Put_Line ("Constraint_Error" & " (raised by Check_Exception) detected!");
          when E : others =>
            Put_Line (Exception_Name (E) & " (raised by Check_Exception) detected!");
      end Check_Handle_Exception;

      begin
        case TC is
          when Exception_1_Chk =>
            Check_Handle_Exception (Test_1);
          when Exception_2_Chk =>
            Check_Handle_Exception (Test_2);
        end case;
      end Check;

      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting...");
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;
        Check (Test_Case_Index'Value (Argument (1)));
      end Main;
118.10.3 Re-raising Exceptions

Listing 119: tests.ads

```ada
package Tests is

  type Test_ID is (Test_1, Test_2);
  
  Custom_Exception, Another_Exception : exception;

  procedure Num_Exception_Test (ID : Test_ID);

end Tests;
```

Listing 120: tests.adb

```ada
package body Tests is

  pragma Warnings (Off, "variable ""C"” is assigned but never read");

  procedure Num_Exception_Test (ID : Test_ID) is
    A, B, C : Integer;
  begin
    case ID is
      when Test_1 =>
        A := Integer’Last;
        B := Integer’Last;
        C := A + B;
      when Test_2 =>
        raise Custom_Exception with "Custom_Exception raised!";
    end case;
  end Num_Exception_Test;

  pragma Warnings (On, "variable ""C"” is assigned but never read");

end Tests;
```

Listing 121: check_exception.ads

```ada
with Tests; use Tests;

procedure Check_Exception (ID : Test_ID);
```

Listing 122: check_exception.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Check_Exception (ID : Test_ID) is
  begin
    Num_Exception_Test (ID);
  exception
    when Constraint_Error =>
      Put_Line ("Constraint_Error detected!");
      raise;
    when E : others =>
      Put_Line (Exception_Message (E));
      raise Another_Exception;
  end Check_Exception;
```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;
with Check_Exception;

procedure Main is
  type Test_Case_Index is
    (Exception_1_Chk, Exception_2_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_Handle_Exception (ID : Test_ID) is
      begin
        Check_Exception (ID);
        exception
          when Constraint_Error =>
            Put_Line ("Constraint_Error" & " (raised by Check_Exception) detected!");
          when E : others =>
            Put_Line (Exception_Name (E) & " (raised by Check_Exception) detected!");
      end Check_Handle_Exception;
    begin
      case TC is
        when Exception_1_Chk =>
          Check_Handle_Exception (Test_1);
        when Exception_2_Chk =>
          Check_Handle_Exception (Test_2);
      end case;
    end Check;
  begin
    if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
    elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
      end if;
    Check (Test_Case_Index'Value (Argument (1)));
  end Main;

118.11 Tasking

118.11.1 Display Service

package Display_Services is
  task type Display_Service is
(end of listing)
Listing 125: display_services.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Display_Services is

  task body Display_Service is
  begin
    loop
      select
        accept Display (S : String) do
          Put_Line (S);
          end Display;
        or
         accept Display (I : Integer) do
          Put_Line (Integer'Image (I));
          end Display;
        or
        terminate;
      end select;
    end loop;
  end Display_Service;
end Display_Services;
```

Listing 126: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Services; use Display_Services;

procedure Main is
  type Test_Case_Index is (Display_Service_Chk);
  procedure Check (TC : Test_Case_Index) is
    Display : Display_Service;
  begin
    case TC is
      when Display_Service_Chk =>
        Display.Display ("Hello");
        delay 0.5;
        Display.Display ("Hello again");
        delay 0.5;
        Display.Display (55);
        delay 0.5;
    end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting..."四周);
    return;
  elsif Argument_Count > 1 then
    (continues on next page)
```
Learning Ada

(continued from previous page)

```ada
28   Put_Line ("Ignoring additional arguments...");
29   end if;
30   Check (Test_Case_Index'Value (Argument (1)));
31   end Main;
```

118.11.2 Event Manager

Listing 127: event_managers.ads

```ada
with Ada.Real_Time; use Ada.Real_Time;

package Event_Managers is

  task type Event_Manager is
  entry Start (ID : Natural);
  entry Event (T : Time);
  end Event_Manager;

end Event_Managers;
```

Listing 128: event_managers.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Event_Managers is

  task body Event_Manager is
  Event_ID : Natural := 0;
  Event_Delay : Time;
  begin
    accept Start (ID : Natural) do
      Event_ID := ID;
    end Start;

    accept Event (T : Time) do
      Event_Delay := T;
    end Event;

    delay until Event_Delay;

    Put_Line ("Event #" & Natural'Image (Event_ID));
  end Event_Manager;

end Event_Managers;
```

Listing 129: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Event_Managers; use Event_Managers;
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
  type Test_Case_Index is (Event_Manager_Chk);

  procedure Check (TC : Test_Case_Index) is
```
Ev_Mng : array (1 .. 5) of Event_Manager;
begin
  case TC is
    when Event_Manager_Chk =>
      for I in Ev_Mng'Range loop
        Ev_Mng (I).Start (I);
      end loop;
      Ev_Mng (1).Event (Clock + Seconds (5));
      Ev_Mng (2).Event (Clock + Seconds (3));
      Ev_Mng (3).Event (Clock + Seconds (1));
      Ev_Mng (4).Event (Clock + Seconds (2));
      Ev_Mng (5).Event (Clock + Seconds (4));
  end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
  return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.11.3 Generic Protected Queue

Listing 130: gen_queues.ads

generic
  type Queue_Index is mod <;
  type T is private;
package Gen_Queues is
  type Queue_Array is array (Queue_Index) of T;
  protected type Queue is
    function Empty return Boolean;
    function Full return Boolean;
    entry Push (V : T);
    entry Pop (V : out T);
  private
    N : Natural := 0;
    Idx : Queue_Index := Queue_Array'First;
    A : Queue_Array;
  end Queue;
end Gen_Queues;

Listing 131: gen_queues.adb

package body Gen_Queues is
  protected body Queue is
    function Empty return Boolean is
      (N = 0);
function Full return Boolean is
  (N = A'Length);
entry Push (V : T) when not Full is
begin
  A (Idx) := V;
  Idx := Idx + 1;
  N := N + 1;
end Push;
entry Pop (V : out T) when not Empty is
begin
  N := N - 1;
  V := A (Idx - Queue_Index (N) - 1);
end Pop;
end Queue;
end Gen_Queues;

package Queue_Tests is
  procedure Simple_Test;
  procedure Concurrent_Test;
end Queue_Tests;

package body Queue_Tests is
  Max : constant := 10;
type Queue_Mod is mod Max;
procedure Simple_Test is
  package Queues_Float is new Gen_Queues (Queue_Mod, Float);
  Q_F : Queues_Float.Queue;
  V : Float;
begin
  V := 10.0;
  while not Q_F.Full loop
    Q_F.Push (V);
    V := V + 1.5;
  end loop;
  while not Q_F.Empty loop
    Q_F.Pop (V);
    Put_Line ("Value from queue: " & Float'Image (V));
  end loop;
end Simple_Test;
end Queue_Tests;
end Simple_Test;

procedure Concurrent_Test is
package Queues_Integer is new Gen_QUEUES (Queue_Mod, Integer);
Q_I : Queues_Integer.Queue;
task T_Producer;
task T_Consumer;
task body T_Producer is
  V : Integer := 100;
begin
  for I in 1 .. 2 * Max loop
    Q_I.Push (V);
  end loop;
end T_Producer;
task body T_Consumer is
  V : Integer;
begin
  delay 1.5;
  while not Q_I.Empty loop
    Q_I.Pop (V);
    Put_Line ("Value from queue: " & Integer'Image (V));
    delay 0.2;
  end loop;
end T_Consumer;
end Concurrent_Test;
end Queue_Tests;

Listing 134: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Tests; use Queue_Tests;

procedure Main is
  type Test_Case_Index is (Simple_Queue_Chk, Concurrent_Queue_Chk);
  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Simple_Queue_Chk =>
        Simple_Test;
      when Concurrent_Queue_Chk =>
        Concurrent_Test;
    end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  end if;
elsif Argument_Count > 1 then
  Put_Line("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.12 Design by contracts

118.12.1 Price Range

Listing 135: prices.ads

package Prices is
  type Amount is delta 10.0 ** (-2) digits 12;
  -- subtype Price is Amount range 0.0 .. Amount'Last;
  subtype Price is Amount
    with Static_Predicate => Price >= 0.0;
end Prices;

Listing 136: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Prices; use Prices;

procedure Main is
  type Test_Case_Index is
    (Price_Range_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_Range (A : Amount) is
      P : constant Price := A;
      begin
        Put_Line("Price: " & Price'Image (P));
        Check_Range;
    begin
      case TC is
        when Price_Range_Chk =>
          Check_Range (-2.0);
      end case;
      exception
        when Constraint_Error =>
          Put_Line("Constraint_Error detected (NOT as expected)."); // continue
        when Assert_Failure =>
          Put_Line("Assert_Failure detected (as expected).");
      end Check;
    end Check;

(continues on next page)
begin
if Argument_Count < 1 then
  Put_Line ("ERROR: missing arguments! Exiting...");
  return;
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.12.2 Pythagorean Theorem: Predicate

Listing 137: triangles.ads
package Triangles is
  subtype Length is Integer;
  type Right_Triangle is record
    H  : Length := 0;
    -- Hypotenuse
    C1, C2 : Length := 0;
    -- Catheti / legs
    end record
    with Dynamic_Predicate => H * H = C1 * C1 + C2 * C2;
  function Init (H, C1, C2 : Length) return Right_Triangle is
    ((H, C1, C2));
end Triangles;

Listing 138: triangles-io.ads
package Triangles.IO is
  function Image (T : Right_Triangle) return String;
end Triangles.IO;

Listing 139: triangles-io.adb
package body Triangles.IO is
  function Image (T : Right_Triangle) return String is
    "(" & Length'Image (T.H) & ", " & Length'Image (T.C1) & ", " & Length'Image (T.C2) & ");"
end Triangles.IO;

Listing 140: main.adb
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;

with Triangles.IO; use Triangles.IO;

procedure Main is

    type Test_Case_Index is
      (Triangle_8_6_Pass_Chk,
       Triangle_8_6_Fail_Chk,
       Triangle_10_24_Pass_Chk,
       Triangle_10_24 Fail_Chk,
       Triangle_18_24_Pass_Chk,
       Triangle_18_24_Fail_Chk);

    procedure Check (TC : Test_Case_Index) is
        procedure Check_Triangle (H, C1, C2 : Length) is
            T : Right_Triangle;
            begin
                T := Init (H, C1, C2);
                Put_Line (Image (T));
            exception
                when Constraint_Error =>
                    Put_Line ("Constraint_Error detected (NOT as expected)."));
                when Assert_Failure =>
                    Put_Line ("Assert_Failure detected (as expected)."));
            end Check_Triangle;

            begin
                case TC is
                    when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
                    when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
                    when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
                    when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
                    when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
                    when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
                end case;
            end Check;

            begin
                if Argument_Count < 1 then
                    Put_Line ("ERROR: missing arguments! Exiting..."");
                    return;
                elsif Argument_Count > 1 then
                    Put_Line ("Ignoring additional arguments..."));
                end if;
            Check (Test_Case_Index'Value (Argument (1)));
        end Main;
118.12.3 Pythagorean Theorem: Precondition

Listing 141: triangles.ads

```ada
package Triangles is

  subtype Length is Integer;

  type Right_Triangle is record
    H    : Length := 0;  -- Hypotenuse
    C1, C2 : Length := 0;  -- Catheti / legs
  end record;

  function Init (H, C1, C2 : Length) return Right_Triangle is
    ((H, C1, C2))
    with Pre => H * H = C1 * C1 + C2 * C2;
  end Triangles;

```

Listing 142: triangles-io.ads

```ada
package Triangles.IO is

  function Image (T : Right_Triangle) return String;

end Triangles.IO;
```

Listing 143: triangles-io.adb

```ada
package body Triangles.IO is

  function Image (T : Right_Triangle) return String is
    "(" & Length'Image (T.H) & "", " & Length'Image (T.C1) & "", " & Length'Image (T.C2) & ")")
  end Triangles.IO;
```

Listing 144: main.adb

```ada
with Ada.Command_Line;  use Ada.Command_Line;
with Ada.Text_IO;  use Ada.Text_IO;
with System.Assertions;  use System.Assertions;
with Triangles;  use Triangles;
with Triangles.IO;  use Triangles.IO;

procedure Main is

  type Test_Case_Index is
    Triangle_8_6_Pass_Chk,
    Triangle_8_6_Fail_Chk,
    Triangle_10_24_Pass_Chk,
    Triangle_10_24_Fail_Chk,
    Triangle_18_24_Pass_Chk,
    Triangle_18_24_Fail_Chk;

  procedure Check (TC : Test_Case_Index) is
    (continues on next page)
```
procedure Check_Triangle (H, C1, C2 : Length) is
begin
T := Init (H, C1, C2);
Put_Line (Image (T));
end Check_Triangle;

begin
case TC is
when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
end case;
end Check;

if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting..."inhed return;
elsif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments..."inhed end if;
end Main;

118.12.4 Pythagorean Theorem: Postcondition

Listing 145: triangles.ads

package Triangles is

subtype Length is Integer;

type Right_Triangle is record
H : Length := 0;
-- Hypotenuse
C1, C2 : Length := 0;
-- Catheti / legs
end record;

function Init (H, C1, C2 : Length) return Right_Triangle is
(Init'Result.H, C1, C2)
with Post => (Init'Result.H * Init'Result.H
= Init'Result.C1 * Init'Result.C1
+ Init'Result.C2 * Init'Result.C2);
Learning Ada

Listing 146: triangles-io.ads

```ada
package Triangles.IO is

   function Image (T : Right_Triangle) return String;

end Triangles.IO;
```

Listing 147: triangles-io.adb

```ada
package body Triangles.IO is

   function Image (T : Right_Triangle) return String is
      "(" & Length'Image (T.H) & ", " & Length'Image (T.C1) & ", " & Length'Image (T.C2) & ")";

end Triangles.IO;
```

Listing 148: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

   type Test_Case_Index is
      (Triangle_8_6_Pass_Chk,
       Triangle_8_6_Fail_Chk,
       Triangle_10_24_Pass_Chk,
       Triangle_10_24_Fail_Chk,
       Triangle_18_24_Pass_Chk,
       Triangle_18_24_Fail_Chk);

   procedure Check (TC : Test_Case_Index) is
      procedure Check_Triangle (H, C1, C2 : Length) is
         begin
            T := Init (H, C1, C2);
            Put_Line (Image (T));
         exception
            when Constraint_Error =>
               Put_Line ("Constraint_Error detected (NOT as expected)."");
            when Assert_Failure =>
               Put_Line ("Assert_Failure detected (as expected)."");
         end Check_Triangle;

      begin
         case TC is
            when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
            when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
            when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
            when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
            when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
            when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
         end case;
      end Check;
```

(continues on next page)
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.12.5 Pythagorean Theorem: Type Invariant

Listing 149: triangles.ads

package Triangles is
  subtype Length is Integer;

  type Right_Triangle is private
    with Type_Invariant => Check (Right_Triangle);

  function Check (T: Right_Triangle) return Boolean;

  function Init (H, C1, C2 : Length) return Right_Triangle;

private

  type Right_Triangle is record
    H    : Length := 0;
    -- Hypotenuse
    C1, C2 : Length := 0;
    -- Catheti / legs
  end record;

  function Init (H, C1, C2 : Length) return Right_Triangle is
    ((H, C1, C2));

  function Check (T: Right_Triangle) return Boolean is
    (T.H * T.H = T.C1 * T.C1 + T.C2 * T.C2);
end Triangles;

Listing 150: triangles-io.ads

package Triangles.IO is
  function Image (T: Right_Triangle) return String;
end Triangles.IO;

Listing 151: triangles-io.adb

package body Triangles.IO is
  (continues on next page)
function Image (T : Right_Triangle) return String is
  ("(
    & Length'Image (T.H)
  
  & ", 
  & Length'Image (T.C1)
  
  & ", 
  & Length'Image (T.C2)
  
  & ")
);
end Triangles.IO;

Listing 152: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is
  type Test_Case_Index is
    (Triangle_8_6_Pass_Chk,
    Triangle_8_6_Fail_Chk,
    Triangle_10_24_Pass_Chk,
    Triangle_10_24_Fail_Chk,
    Triangle_18_24_Pass_Chk,
    Triangle_18_24_Fail_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
      exception
        when Constraint_Error =>
          Put_Line ("Constraint_Error detected (NOT as expected).".Build);
        when Assert_Failure =>
          Put_Line ("Assert_Failure detected (as expected).".Build);
      end Check_Triangle;
        begin
          case TC is
            when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
            when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
            when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
            when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
            when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
            when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
          end case;
        end Check;
        begin
          if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
          elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
          end if;
          Check (Test_Case_Index'Value (Argument (1)));
(continues on next page)
end Main;

118.12.6 Primary Colors

Listing 153: color_types.ads

package Color_Types is

  type HTML_Color is
    (Salmon,
     Firebrick,
     Red,
     Darkred,
     Lime,
     Forestgreen,
     Green,
     Darkgreen,
     Blue,
     Mediumblue,
     Darkblue);

  subtype Int_Color is Integer range 0 .. 255;

  function Image (I : Int_Color) return String;

  type RGB is record
    Red : Int_Color;
    Green : Int_Color;
    Blue : Int_Color;
  end record;

  function To_RGB (C : HTML_Color) return RGB;

  function Image (C : RGB) return String;

  type HTML_Color_RGB_Array is array (HTML_Color) of RGB;

  To_RGB_Lookup_Table : constant HTML_Color_RGB_Array :=
    (Salmon  => (16#FA#, 16#80#, 16#72#),
     Firebrick => (16#B2#, 16#22#, 16#22#),
     Red => (16#FF#, 16#00#, 16#00#),
     Darkred => (16#8B#, 16#00#, 16#00#),
     Lime => (16#00#, 16#FF#, 16#00#),
     Forestgreen => (16#22#, 16#8B#, 16#22#),
     Green => (16#00#, 16#80#, 16#00#),
     Darkgreen => (16#00#, 16#64#, 16#00#),
     Blue => (16#00#, 16#00#, 16#FF#),
     Mediumblue => (16#00#, 16#00#, 16#CD#),
     Darkblue => (16#00#, 16#00#, 16#8B#));

  subtype HTML_RGB_Color is HTML_Color
    with Static_Predicate => HTML_RGB_Color in Red | Green | Blue;

  function To_Int_Color (C : HTML_Color;
    S : HTML_RGB_Color) return Int_Color;

-- Convert to hexadecimal value for the selected RGB component S

end Color_Types;
with Ada.Integer_Text_IO;

package body Color_Types is

 function To_RGB (C : HTML_Color) return RGB is
 begin
   return To_RGB_Lookup_Table (C);
 end To_RGB;

 function To_Int_Color (C : HTML_Color; S : HTML_RGB_Color) return Int_Color is
 C_RGB : constant RGB := To_RGB (C);
 begin
   case S is
     when Red => return C_RGB.Red;
     when Green => return C_RGB.Green;
     when Blue => return C_RGB.Blue;
   end case;
 end To_Int_Color;

 function Image (I : Int_Color) return String is
 S : String (Str_Range);
 begin
   Ada.Integer_Text_IO.Put (To => S, Item => I, Base => 16);
   return S;
 end Image;

 function Image (C : RGB) return String is
 begin
   return ("(Red => " & Image (C.Red) & ", Green => " & Image (C.Green) & ", Blue => " & Image (C.Blue) & ")");
 end Image;

end Color_Types;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
type Test_Case_Index is
  (HTML_Color_Red_Chk, HTML_Color_Green_Chk, HTML_Color_Blue_Chk);

 procedure Check (TC : Test_Case_Index) is

 procedure Check_HTML_Colors (S : HTML_RGB_Color) is
 begin
   Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
   for I in HTML_Color'Range loop
 (continues on next page)
Put_Line (HTML_Color'Image (I) & " => " & Image (To_Int_Color (I, S)) & ".");
end loop;
end Check_HTML_Colors;

begin
  case TC is
    when HTML_Color_Red_Chk =>
      Check_HTML_Colors (Red);
    when HTML_Color_Green_Chk =>
      Check_HTML_Colors (Green);
    when HTML_Color_Blue_Chk =>
      Check_HTML_Colors (Blue);
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
  return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.13 Object-oriented programming

118.13.1 Simple type extension

Listing 156: type_extensions.ads

package Type_Extensions is
  type T_Float is tagged record
    F : Float;
  end record;

  function Init (F : Float) return T_Float;

  function Init (I : Integer) return T_Float;

  function Image (T : T_Float) return String;

  type T_Mixed is new T_Float with record
    I : Integer;
  end record;

  function Init (F : Float) return T_Mixed;

  function Init (I : Integer) return T_Mixed;

  function Image (T : T_Mixed) return String;
end Type_Extensions;
<table>
<thead>
<tr>
<th>Listing 157: type_extensions.adb</th>
</tr>
</thead>
<tbody>
<tr>
<td>package body Type_Extensions is</td>
</tr>
<tr>
<td>function Init (F : Float) return T_Float is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>return ((F =&gt; F));</td>
</tr>
<tr>
<td>end Init;</td>
</tr>
<tr>
<td>function Init (I : Integer) return T_Float is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>return ((F =&gt; Float (I)));</td>
</tr>
<tr>
<td>end Init;</td>
</tr>
<tr>
<td>function Init (F : Float) return T_Mixed is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>return ((F =&gt; F,</td>
</tr>
<tr>
<td>I =&gt; Integer (F)));</td>
</tr>
<tr>
<td>end Init;</td>
</tr>
<tr>
<td>function Init (I : Integer) return T_Mixed is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>return ((F =&gt; Float (I),</td>
</tr>
<tr>
<td>I =&gt; I));</td>
</tr>
<tr>
<td>end Init;</td>
</tr>
<tr>
<td>function Image (T : T_Float) return String is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>return &quot;{ F =&gt; &quot; &amp; Float'Image (T.F) &amp; &quot; }&quot;;</td>
</tr>
<tr>
<td>end Image;</td>
</tr>
<tr>
<td>function Image (T : T_Mixed) return String is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>return &quot;{ F =&gt; &quot; &amp; Float'Image (T.F)</td>
</tr>
<tr>
<td>&amp; &quot;, I =&gt; &quot; &amp; Integer'Image (T.I) &amp; &quot; }&quot;;</td>
</tr>
<tr>
<td>end Image;</td>
</tr>
<tr>
<td>end Type_Extensions;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 158: main.adb</th>
</tr>
</thead>
<tbody>
<tr>
<td>with Ada.Command_Line; use Ada.Command_Line;</td>
</tr>
<tr>
<td>with Ada.Text_IO; use Ada.Text_IO;</td>
</tr>
<tr>
<td>with Type_Extensions; use Type_Extensions;</td>
</tr>
<tr>
<td>procedure Main is</td>
</tr>
<tr>
<td>type Test_Case_Index is</td>
</tr>
<tr>
<td>(Type_Extension_Chk);</td>
</tr>
<tr>
<td>procedure Check (TC : Test_Case_Index) is</td>
</tr>
<tr>
<td>F1, F2 : T_Float;</td>
</tr>
<tr>
<td>M1, M2 : T_Mixed;</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>case TC is</td>
</tr>
<tr>
<td>when Type_Extension_Chk =&gt;</td>
</tr>
<tr>
<td>F1 := Init (2.0);</td>
</tr>
<tr>
<td>F2 := Init (3);</td>
</tr>
<tr>
<td>M1 := Init (4.0);</td>
</tr>
</tbody>
</table>
| M2 := Init (5); | (continues on next page)
if \( M_2 \) in \( T_{\text{Float}}'\text{Class} \) then
\[
\text{Put Line ("T_Mixed is in T_Float'Class as expected")};
\]
\end if;

\[
\begin{align*}
\text{Put Line ("F1: " & Image (F1));} \\
\text{Put Line ("F2: " & Image (F2));} \\
\text{Put Line ("M1: " & Image (M1));} \\
\text{Put Line ("M2: " & Image (M2));}
\end{align*}
\]
\end case;
\end Check;
\begin
\begin{align*}
\text{if Argument_Count < 1 then} & \text{ Put Line ("ERROR: missing arguments! Exiting...");} \\
\text{return;} & \text{ end if;} \\
\text{elsif Argument_Count > 1 then} & \text{ Put Line ("Ignoring additional arguments...");} \\
\text{end if;}
\end{align*}
\end{align*}
\]
\text{Check (Test_Case_Index'Value (Argument (1)))};
\end Main;

### 118.13.2 Online Store

Listing 159: online_store.ads

```ada
with Ada.Calendar; use Ada.Calendar;

package Online_Store is

  type Amount is delta 10.0**(2) digits 10;

  subtype Percentage is Amount range 0.0 .. 1.0;

  type Member is tagged record
    Start : Year_Number;
  end record;

  type Member_Access is access Member'Class;

  function Get_Status (M : Member) return String;

  function Get_Price (M : Member; P : Amount) return Amount;

  type Full_Member is new Member with record
    Discount : Percentage;
  end record;

  function Get_Status (M : Full_Member) return String;

  function Get_Price (M : Full Member; P : Amount) return Amount;

end Online_Store;
```
Listing 160: online_store.adb

package body Online_Store is

  function Get_Status (M : Member) return String is
    "Associate Member";

  function Get_Status (M : Full Member) return String is
    "Full Member";

  function Get_Price (M : Member;
      P : Amount) return Amount is (P);

  function Get_Price (M : Full Member;
      P : Amount) return Amount is
      (P * (1.0 - M.Discount));

end Online_Store;

Listing 161: online_store-tests.ads

package Online_Store.Tests is

  procedure Simple_Test;

end Online_Store.Tests;

Listing 162: online_store-tests.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Online_Store.Tests is

  procedure Simple_Test is

    type Member_Due_Amount is record
      Member : Member_Access;
      Due_Amount : Amount;
    end record;

    function Get_Price (MA : Member_Due_Amount) return Amount is begin
      return MA.Member.Get_Price (MA.Due_Amount);
    end Get_Price;

    type Member_Due_Amounts is array (Positive range <>) of Member_Due_Amount;

    DB : constant Member_Due_Amounts (1 .. 4) := ((Member => new Member'(Start => 2010),
      Due_Amount => 250.0),
      (Member => new Full Member'(Start => 1998,
        Discount => 0.1),
      Due_Amount => 160.0),
      (Member => new Full Member'(Start => 1987,
        Discount => 0.2),
      Due_Amount => 400.0),
      (Member => new Member'(Start => 2013),
      Due_Amount => 110.0));

    begin
      for I in DB'Range loop
        Put_Line ("Member #" & Positive'Image (I));
      end loop;
    end for I in DB'Range loop

end Simple_Test;

(continues on next page)
Learning Ada

(continued from previous page)

```ada
Put_Line ("Status: " & DB (I).Member.Get_Status);
Put_Line ("Since: " & Year_Number'Image (DB (I).Member.Start));
Put_Line ("Due Amount: " & Amount'Image (Get_Price (DB (I))));
end loop;
end Simple_Test;
end Online_Store.Tests;
```

Listing 163: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Online_Store; use Online_Store;
with Online_Store.Tests; use Online_Store.Tests;
procedure Main is

  type Test_Case_Index is
    (Type_CHK, Unit_Test_CHK);

  procedure Check (TC : Test_Case_Index) is
    function Result_Image (Result : Boolean) return String is
      (if Result then "OK" else "not OK");
    begin
      case TC is
        when Type_CHK =>
          declare
            AM : constant Member := (Start => 2002);
            FM : constant Full_Member := (Start => 1990,
                                          Discount => 0.2);
          begin
            Put_Line ("Testing Status of Associate Member Type => 
                          & Result_Image (AM.Get_Status = "Associate Member");
            Put_Line ("Testing Status of Full Member Type => 
                          & Result_Image (FM.Get_Status = "Full Member");
            Put_Line ("Testing Discount of Associate Member Type => 
                          & Result_Image (AM.Get_Price (100.0) = 100.0));
            Put_Line ("Testing Discount of Full Member Type => 
                          & Result_Image (FM.Get_Price (100.0) = 80.0));
          end;
        when Unit_Test_CHK =>
          Simple_Test;
        end case;
      end Check;
    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;
```

2038 Chapter 118. Solutions
118.14 Standard library: Containers

118.14.1 Simple todo list

Listing 164: todo_lists.ads

```ada
with Ada.Containers.Vectors;

package Todo_Lists is

    type Todo_Item is access String;

    package Todo_List_Pkg is new Ada.Containers.Vectors
        (Index_Type => Natural,
         Element_Type => Todo_Item);

    subtype Todo_List is Todo_List_Pkg.Vector;

    procedure Add (Todos : in out Todo_List;
                   Item    : String);

    procedure Display (Todos : Todo_List);

end Todo_Lists;
```

Listing 165: todo_lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Todo_Lists is

    procedure Add (Todos : in out Todo_List;
                   Item    : String) is
    begin
        Todos.Append (new String'(Item));
    end Add;

    procedure Display (Todos : Todo_List) is
    begin
        Put_Line ("TO-DO LIST");
        for T of Todos loop
            Put_Line (T.all);
        end loop;
    end Display;

end Todo_Lists;
```

Listing 166: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is
    type Test_Case_Index is
        (Todo_List_Chk);

    procedure Check (TC : Test_Case_Index) is
        T : Todo_List;
```

(continues on next page)
begin
  case TC is
    when Todo_List_Chk =>
      Add (T, "Buy milk");
      Add (T, "Buy tea");
      Add (T, "Buy present");
      Add (T, "Buy tickets");
      Add (T, "Pay electricity bill");
      Add (T, "Schedule dentist appointment");
      Add (T, "Call sister");
      Add (T, "Revise spreadsheet");
      Add (T, "Edit entry page");
      Add (T, "Select new design");
      Add (T, "Create upgrade plan");
      Display (T);
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.14.2 List of unique integers

Listing 167: ops.ads

with Ada.Containers.Ordered_Sets;

package Ops is

  type Int_Array is array (Positive range <>) of Integer;

  package Integer_Sets is new Ada.Containers.Ordered_Sets
    (Element_Type => Integer);

  subtype Int_Set is Integer_Sets.Set;

  function Get_Unique (A : Int_Array) return Int_Set;
  function Get_Unique (A : Int_Array) return Int_Array;

end Ops;

Listing 168: ops.adb

package body Ops is

  function Get_Unique (A : Int_Array) return Int_Set is
    S : Int_Set;
    begin
      for E of A loop

function Get_Unique (A : Int_Array) return Int_Array is
  S : constant Int_Set := Get_Unique (A);
  AR : Int_Array (1.. Positive (S.Length));
  I : Positive := 1;
begin
  for E of S loop
    AR (I) := E;
    I := I + 1;
  end loop;

  return AR;
end Get_Unique;

procedure Main is
  type Test_Case_Index is
    (Get_Unique_Set_Chk, Get_Unique_Array_Chk);
  procedure Check (TC : Test_Case_Index; A : Int_Array) is
     S : constant Int_Set := Get_Unique (A);
      procedure Display_Unique_Set (A : Int_Array) is
        begin
          for E of S loop
            Put_Line (Integer'Image (E));
          end loop;
        end Display_Unique_Set;

      procedure Display_Unique_Array (A : Int_Array) is
        AU : constant Int_Array := Get_Unique (A);
        begin
          for E of AU loop
            Put_Line (Integer'Image (E));
          end loop;
        end Display_Unique_Array;
  begin
    case TC is
      when Get_Unique_Set_Chk => Display_Unique_Set (A);
      when Get_Unique_Array_Chk => Display_Unique_Array (A);
    end case;  
  end Check;
begin
  if Argument_Count < 3 then
Learning Ada

Put_Line ("ERROR: missing arguments! Exiting...");
return;
else
declare
  A : Int_Array (1 .. Argument_Count - 1);
begin
  for I in A'Range loop
    A (I) := Integer'Value (Argument (1 + I));
  end loop;
  Check (Test_Case_Index'Value (Argument (1)), A);
end;
end if;
end Main;

118.15 Standard library: Dates & Times

118.15.1 Holocene calendar

Listing 170: to_holocene_year.adb

with Ada.Calendar; use Ada.Calendar;
function To_Holocene_Year (T : Time) return Integer is
begin
  return Year (T) + 10_000;
end To_Holocene_Year;

Listing 171: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;

procedure Main is
  type Test_Case_Index is (Holocene_Chk);
  procedure Display_Holocene_Year (Y : Year_Number) is
    HY : Integer;
  begin
    HY := To_Holocene_Year (Time_Of (Y, 1, 1));
    Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));
    Put_Line ("Year (Holocene): " & Integer'Image (HY));
  end Display_Holocene_Year;
  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Holocene_Chk =>
        Display_Holocene_Year (2012);
        Display_Holocene_Year (2020);
    end case;
  end Check;
begin
if Argument_Count < 1 then
  Put_Line ("ERROR: missing arguments! Exiting...");
  return;
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.15.2 List of events

Listing 172: events.ads

with Ada.Containers.Vectors;
package Events is
  type Event_Item is access String;
  package Event_ItemContainers is new Ada.Containers.Vectors
    (Index_Type => Positive,
     Element_Type => Event_Item);
  subtype Event_Items is Event_ItemContainers.Vector;
end Events;

Listing 173: events-lists.ads

with Ada.Calendar;
  use Ada.Calendar;
with Ada.Containers.Ordered_Maps;
package Events.Lists is
  type Event_List is tagged private;
  procedure Add (Events : in out Event_List;
                  Event_Time : Time;
                  Event : String);
  procedure Display (Events : Event_List);
private
  package Event_Time_ItemContainers is new Ada.Containers.Ordered_Maps
    (Key_Type => Time,
     Element_Type => Event_Items,
     "=" => Event_ItemContainers."=");
  type Event_List is new Event_Time_ItemContainers.Map with null record;
end Events.Lists;
Listing 174: events-lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

package body Events.Lists is

   procedure Add (Events : in out Event_List;
       Event_Time : Time;
       Event : String) is
     use Event_Item_Containers;
     E : constant Event_Item := new String'(Event);
     begin
       if not Events.Contains (Event_Time) then
         Events.Include (Event_Time, Empty_Vector);
       end if;
       Events (Event_Time).Append (E);
     end Add;

   function Date_Image (T : Time) return String is
     Date_Img : constant String := Image (T);
     begin
       return Date_Img (1 .. 10);
     end;

   procedure Display (Events : Event_List) is
     use Event_Time_Item_Containers;
     T : Time;
     begin
       Put_Line ("EVENTS LIST");
       for C in Events.Iterate loop
         T := Key (C);
         Put_Line ("- " & Date_Image (T));
         for I of Events (C) loop
           Put_Line (" - " & I.all);
         end loop;
       end loop;
     end Display;

end Events.Lists;
```

Listing 175: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Events.Lists; use Events.Lists;

procedure Main is
  type Test_Case_Index is
    (Event_List_Chk);

  procedure Check (TC : Test_Case_Index) is
    EL : Event_List;
    begin
      case TC is
        when Event_List_Chk =>
          EL.Add (Time_Of (2018, 2, 16),
                  "Final check");
```

(continues on next page)
EL.Add (Time_Of (2018, 2, 16),
"Release");
EL.Add (Time_Of (2018, 12, 3),
"Brother's birthday" );
EL.Add (Time_Of (2018, 1, 1),
"New Year's Day");
EL.Display;
end case;
end Check;
begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting... ");
return;
elif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

118.16 Standard library: Strings

118.16.1 Concatenation

Listing 176: str_concat.ads


package Str_Concat is

    type Unbounded_Strings is array (Positive range <>) of Unbounded_String;

    function Concat (USA : Unbounded_Strings;
        Trim_Str : Boolean;
        Add_Whitespace : Boolean) return Unbounded_String;

    function Concat (USA : Unbounded_Strings;
        Trim_Str : Boolean;
        Add_Whitespace : Boolean) return String;

end Str_Concat;

Listing 177: str_concat.adb

with Ada.Strings; use Ada.Strings;

package body Str_Concat is

    function Concat (USA : Unbounded_Strings;
        Trim_Str : Boolean;
        Add_Whitespace : Boolean) return Unbounded_String is

    function Retrieve (USA : Unbounded_Strings;
        Trim_Str : Boolean;
        Index : Positive) return Unbounded_String is
            US_Internal : Unbounded_String := USA (Index);
...
begin
  if Trim_Str then
    US_Internal := Trim (US_Internal, Both);
  end if;
  return US_Internal;
end Retrieve;

US : Unbounded_String := To_Unbounded_String ("");

begin
  for I in USA'First .. USA'Last - 1 loop
    US := US & Retrieve (USA, Trim_Str, I);
    if Add_Whitespace then
      US := US & " ";
    end if;
  end loop;
  US := US & Retrieve (USA, Trim_Str, USA'Last);
  return US;
end Concat;

function Concat (USA : Unbounded_Strings;
                 Trim_Str : Boolean;
                 Add_Whitespace : Boolean) return String is
begin
  return To_String (Concat (USA, Trim_Str, Add_Whitespace));
end Concat;

procedure Main is
  type Test_Case_Index is
    (Unbounded_Concat_No_Trim_No_WS_Chk,
     Unbounded_Concat_Trim_No_WS_Chk,
     String_Concat_Trim_WS_Chk,
     Concat_Single_Element);

  procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Unbounded_Concat_No_Trim_No_WS_Chk =>
      declare
        S : constant Unbounded_Strings := (To_Unbounded_String ("Hello"),
                                To_Unbounded_String (" World"),
                                To_Unbounded_String ("!"));
      begin
        Put_Line (To_String (Concat (S, False, False)));
      end;
    when Unbounded_Concat_Trim_No_WS_Chk =>
      declare
        S : constant Unbounded_Strings := (To_Unbounded_String (" This "),
                                To_Unbounded_String (" _is_ "));
Learning Ada

To_Unbounded_String (" a "),
To_Unbounded_String (" _check "));
begin
   Put_Line (To_String (Concat (S, True, False)));  
end;
when String_Concat_Trim_WS_Chk =>
   declare
      S : constant Unbounded_Strings := (  
         To_Unbounded_String (" This "),  
         To_Unbounded_String (" is a "),  
         To_Unbounded_String (" test. "));
   begin
      Put_Line (Concat (S, True, True));  
   end;
when Concat_Single_Element =>
   declare
      S : constant Unbounded_Strings := (  
         1 => To_Unbounded_String (" Hi "));
   begin
      Put_Line (Concat (S, True, True));
   end;
end case;
end Check;
end case;
end Main;

118.16.2 List of events

Listing 179: events.ads

with Ada.Containers.Vectors;

package Events is
   subtype Event_Item is Unbounded_String;

   package Event_Item_Containers is new  
      Ada.Containers.Vectors
         (Index_Type    => Positive,  
          Element_Type => Event_Item);

   subtype Event_Items is Event_Item_Containers.Vector;

end Events;

Listing 180: events-lists.ads

with Ada.Calendar; use Ada.Calendar;

(continues on next page)
with Ada.Containers.Ordered_Maps;

package Events.Lists is

    type Event_List is tagged private;

    procedure Add (Events : in out Event_List;
                   Event_Time :     Time;
                   Event      :     String);

    procedure Display (Events : Event_List);

private

package Event_Time_Item_Containers is new
   Ada.Containers.Ordered_Maps
      (Key_Type       => Time,
       Element_Type   => Event_Items,
       "="            => Event_Item_Containers."=");

    type Event_List is new Event_Time_Item_Containers.Map with null record;

end Events.Lists;

Listing 181: events-lists.adb

with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

package body Events.Lists is

    procedure Add (Events : in out Event_List;
                   Event_Time :     Time;
                   Event      :     String) is

        use Event_Item_Containers;
        E : constant Event_Item := To_Unbounded_String (Event);

    begin
        if not Events.Contains (Event_Time) then
            Events.Include (Event_Time, Empty_Vector);
        end if;
        Events (Event_Time).Append (E);
    end Add;

    function Date_Image (T : Time) return String is
        Date_Img : constant String := Image (T);
    begin
        return Date_Img (1 .. 10);
    end;

    procedure Display (Events : Event_List) is
        use Event_Time_Item_Containers;
        T : Time;

    begin
        Put_Line ("EVENTS LIST");
        for C in Events.Iterate loop
            T := Key (C);
            Put_Line ("- " & Date_Image (T));
            for I of Events (C) loop
                Put_Line ("  " & To_String (I));
            end loop;
        end loop;
    end Display;

(continues on next page)
procedure Main is
  type Test_Case_Index is
    (Unbounded_String_Chk,
     Event_List_Chk);

  procedure Check (TC : Test_Case_Index) is
    EL : Event_List;
    begin
      case TC is
        when Unbounded_String_Chk =>
          declare
            S : constant Events.Event_Item := To_Unbounded_String ("Checked");
          begin
            Put_Line (To_String (S));
          end;
        when Event_List_Chk =>
          EL.Add (Time_Of (2018, 2, 16),
            "Final check");
          EL.Add (Time_Of (2018, 2, 16),
            "Release");
          EL.Add (Time_Of (2018, 12, 3),
            "Brother's birthday");
          EL.Add (Time_Of (2018, 1, 1),
            "New Year's Day");
          EL.Display;
      end case;
      end Check;
      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting..."中介机构); 
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;
        Check (Test_Case_Index'Value (Argument (1)));
118.17 Standard library: Numerics

118.17.1 Decibel Factor

Learning Ada

Listing 183: decibels.ads

```ada
package Decibels is
  subtype Decibel is Float;
  subtype Factor is Float;
  function To_Decibel (F : Factor) return Decibel;
  function To_Factor (D : Decibel) return Factor;
end Decibels;
```

Listing 184: decibels.adb

```ada
package body Decibels is
  function To_Decibel (F : Factor) return Decibel is
    begin
      return 20.0 * Log (F, 10.0);
    end To_Decibel;
  function To_Factor (D : Decibel) return Factor is
    begin
      return 10.0 ** (D / 20.0);
    end To_Factor;
end Decibels;
```

Listing 185: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Decibels; use Decibels;
procedure Main is
  type Test_Case_Index is
    (Db_Chk,
     Factor_Chk);
  procedure Check (TC : Test_Case_Index; V : Float) is
    package F_IO is new Ada.Text_IO.Float_IO (Factor);
    package D_IO is new Ada.Text_IO.Float_IO (Decibel);
    procedure Put_Decibel_Cnv (D : Decibel) is
      F : constant Factor := To_Factor (D);
    begin
      D_IO.Put (D, 0, 2, 0);
      Put (" dB => Factor of ");
      F_IO.Put (F, 0, 2, 0);
      New_Line;
    end;
```
procedure Put_Factor_Cnvt (F : Factor) is
  D : constant Decibel := To_Decibel (F);
begin
  Put ("Factor of ");
  F_IO.Put (F, 0, 2, 0);
  Put (" \Rightarrow ");
  D_IO.Put (D, 0, 2, 0);
  Put_Line (" dB");
end;

begin
  case TC is
    when Db_Chk =>
      Put_Decibel_Cnvt (Decibel (V));
    when Factor_Chk =>
      Put_Factor_Cnvt (Factor (V));
  end case;
end Check;

begin
  if Argument_Count < 2 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
end Main;

118.17.2 Root-Mean-Square

Listing 186: signals.ads

package Signals is
  subtype Sig_Value is Float;
  type Signal is array (Natural range <>) of Sig_Value;
  function Rms (S : Signal) return Sig_Value;
end Signals;

Listing 187: signals.adb


package body Signals is
  function Rms (S : Signal) return Sig_Value is
    Acc : Float := 0.0;
  begin
    for V of S loop
      Acc := Acc + V * V;
    end loop;
    return Sqrt (Acc / Float (S'Length));
  end function Rms;
end body Signals;
end Signals;

Listing 188: signals-std.ads

package Signals.Std is
  Sample_Rate : Float := 8000.0;
  function Generate_Sine (N : Positive; Freq : Float) return Signal;
  function Generate_Square (N : Positive) return Signal;
  function Generate_Triangular (N : Positive) return Signal;
end Signals.Std;

Listing 189: signals-std.adb

with Ada.Numerics; use Ada.Numerics;

package body Signals.Std is
  function Generate_Sine (N : Positive; Freq : Float) return Signal is
    S : Signal (0 .. N - 1);
    begin
      for I in S'First .. S'Last loop
        S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
      end loop;
      return S;
    end;
  end;

  function Generate_Square (N : Positive) return Signal is
    S : constant Signal (0 .. N - 1) := (others => 1.0);
    begin
      return S;
    end;

  function Generate_Triangular (N : Positive) return Signal is
    S : Signal (0 .. N - 1);
    S_Half : constant Natural := S'Last / 2;
    begin
      for I in S'First .. S_Half loop
        S (I) := 1.0 * (Float (I) / Float (S_Half));
      end loop;
      for I in S_Half .. S'Last loop
        S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
      end loop;
      return S;
    end;
end Signals.Std;
Listing 190: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Signals; use Signals;
with Signals.Std; use Signals.Std;

procedure Main is
  type Test_Case_Index is
    (Sine_Signal_Chk, Square_Signal_Chk, Triangular_Signal_Chk);

  procedure Check (TC : Test_Case_Index) is
    package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
    N : constant Positive := 1024;
    S_Si : constant Signal := Generate_Sine (N, 440.0);
    S_Sq : constant Signal := Generate_Square (N);
    S_Tr : constant Signal := Generate_Triangular (N + 1);
    begin
      case TC is
        when Sine_Signal_Chk =>
          Put ("RMS of Sine Signal: ");
          Sig_IO.Put (Rms (S_Si), 0, 2, 0);
          New_Line;
        when Square_Signal_Chk =>
          Put ("RMS of Square Signal: ");
          Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
          New_Line;
        when Triangular_Signal_Chk =>
          Put ("RMS of Triangular Signal: ");
          Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
          New_Line;
      end case;
    end Check;

    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
    end if;

    Check (Test_Case_Index'Value (Argument (1)));
  end Main;
```

118.17.3 Rotation

Listing 191: rotation.ads

```ada
with Ada.Numerics.Complex_Types;
use Ada.Numerics.Complex_Types;

package Rotation is
  type Complex_Points is array (Positive range <>) of Complex;
(continues on next page)
```

118.17. Standard library: Numerics 2053
function Rotation (N : Positive) return Complex_Points;
end Rotation;

Listing 192: rotation.adb

with Ada.Numerics; use Ada.Numerics;

package body Rotation is

    function Rotation (N : Positive) return Complex_Points is
            C_Angle : constant Complex :=
             Compose_From_Polar (1.0, 2.0 * Pi / Float (N));
    begin
        return C : Complex_Points (1 .. N + 1) do
            C (1) := Compose_From_Cartesian (1.0, 0.0);
            for I in C'First + 1 .. C'Last loop
                C (I) := C (I - 1) * C_Angle;
            end loop;
        end return;
    end;
end Rotation;

Listing 193: angles.ads

with Rotation; use Rotation;

package Angles is

    subtype Angle is Float;

    type Angles is array (Positive range <>) of Angle;

    function To_Angles (C : Complex_Points) return Angles;
end Angles;

Listing 194: angles.adb

with Ada.Numerics; use Ada.Numerics;

package body Angles is

    function To_Angles (C : Complex_Points) return Angles is
    begin
        return A : Angles (C'Range) do
            for I in A'Range loop
                A (I) := Argument (C (I)) / Pi * 180.0;
            end loop;
        end return;
    end To_Angles;
end Angles;
package Rotation.Tests is
  procedure Test_Rotation (N : Positive);
  procedure Test_Angles (N : Positive);
end Rotation.Tests;

with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Text_IO.Complex_IO; use Ada.Text_IO.Complex_IO (Complex_Types);
with Ada.Numerics;  use Ada.Numerics;
with Angles;  use Angles;

package body Rotation.Tests is

  package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
  package F_IO is new Ada.Text_IO.Float_IO (Float);

  -- Adapt value due to floating-point inaccuracies
  --
  function Adapt (C : Complex) return Complex is
    function Check_Zero (F : Float) return Float is
      (if F <= 0.0 and F >= -0.01 then 0.0 else F);
    begin
      return C_Out : Complex := C do
        C_Out.Re := Check_Zero (C_Out.Re);
        C_Out.Im := Check_Zero (C_Out.Im);
      end return;
    end Adapt;

    function Adapt (A : Angle) return Angle is
      (if A <= -179.99 and A >= -180.01 then 180.0 else A);
    begin
      C : constant Complex_Points := Rotation (N);
      put_line ("---- Points for " & Positive'Image (N) & " slices ----");
      for V of C loop
        C_IO.Put (Adapt (V), 0, 1, 0);
        New_Line;
      end loop;
      Test_Rotation;

      C : constant Complex_Points := Rotation (N);
      A : constant Angles.Angles := To_Angles (C);
      put_line ("---- Angles for " & Positive'Image (N) & " slices ----");
      for V of A loop
        F_IO.Put (Adapt (V), 0, 2, 0);
        put_line (" degrees");
      end loop;
    end Test_Angles;
endRotation.Tests;
end Test_Angles;
end Rotation.Tests;

Listing 197: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Rotation.Tests; use Rotation.Tests;

procedure Main is
  type Test_Case_Index is
    (Rotation_Chk, Angles_Chk);
  procedure Check (TC : Test_Case_Index; N : Positive) is
    begin
      case TC is
        when Rotation_Chk =>
          Test_Rotation (N);
        when Angles_Chk =>
          Test_Angles (N);
      end case;
    end Check;

    begin
      if Argument_Count < 2 then
        Put_Line ("ERROR: missing arguments! Exiting..." );
        return;
      elsif Argument_Count > 2 then
        Put_Line ("Ignoring additional arguments..." );
      end if;
      Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2))) ;
    end Main;
Part XII

Bug Free Coding with SPARK Ada
Workshop project: Learn to write maintainable bug-free code with SPARK Ada.

This document was written by Robert Tice.

**Note:** The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

---

387 http://creativecommons.org/licenses/by-sa/4.0
In this lab we will build a stack data structure and use the SPARK provers to find the errors in the below implementation.

### 119.1 Background

**So, what is a stack?**

A stack is like a pile of dishes...

1. The pile starts out empty.
2. You add (push) a new plate (data) to the stack by placing it on the top of the pile.
3. To get plates (data) out, you take the one off the top of the pile (pop).
4. Our stack has a maximum height (size) of 9 dishes

**Pushing items onto the stack**

Here's what should happen if we pushed the string MLH onto the stack.
The list starts out empty. Each time we push a character onto the stack, Last increments by 1.

Popping items from the stack
Here's what should happen if we popped 2 characters off our stack & then clear it.

**Step 0:**
Start
1: M
2: L
3: H
4:
5:
Last = 3

**Step 1:**
Pop()
1: M
2: L
3: H
4:
5:
Last = 2
returns: ‘H’

**Step 2:**
Pop()
1: M
2: L
3: H
4:
5:
Last = 1
returns: ‘L’

**Step 3:**
Clear()
1: M
2: L
3: H
4:
5:
Last = 0

Note that pop and clear don't unset the Storage array's elements, they just change the value of Last.
119.2 Input Format

N inputs will be read from stdin/console as inputs, C to the stack.

119.3 Constraints

1 <= N <= 1000
C is any character. Characters d and p will be special characters corresponding to the below commands:
p => Pops a character off the stack
d => Prints the current characters in the stack

119.4 Output Format

If the stack currently has the characters "M", "L", and "H" then the program should print the stack like this:
[M, L, H]

119.5 Sample Input

M L H d p d p d p d

119.6 Sample Output

[M, L, H] [M, L] [M] []

Listing 1: stack.ads

package Stack with SPARK_Mode => On is

    procedure Push (V : Character)
    with Pre => not Full,
        Post => Size = Size'Old + 1;

    procedure Pop (V : out Character)
    with Pre => not Empty,
        Post => Size = Size'Old - 1;

    procedure Clear
    with Post => Size = 0;

    function Top return Character
    with Post => Top'Result = Tab(Last);

    Max_Size : constant := 9;

(continues on next page)
-- The stack size.

Last : Integer range 0 .. Max_Size := 0;
-- Indicates the top of the stack. When 0 the stack is empty.

Tab : array (1 .. Max_Size) of Character;
-- The stack. We push and pop pointers to Values.

function Full return Boolean is (Last = Max_Size);
function Empty return Boolean is (Last < 1);
function Size return Integer is (Last);

end Stack;

Listing 2: stack.adb

package body Stack with SPARK_Mode => On is

----------
-- Clear --
----------

procedure Clear
is
begin
  Last := Tab'First;
end Clear;

----------
-- Push --
----------

procedure Push (V : Character)
is
begin
  Tab (Last) := V;
end Push;

----------
-- Pop --
----------

procedure Pop (V : out Character)
is
begin
  Last := Last - 1;
  V := Tab (Last);
end Pop;

----------
-- Top --
----------

function Top return Character
is
begin
  return Tab (1);
end Top;

(continues on next page)
end Stack;

Listing 3: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Stack; use Stack;

procedure Main with SPARK_Mode => Off is

-----------
-- Debug --
-----------

procedure Debug is

if not Stack.Empty then
  Put ("[");
  for I in Stack.Tab'First .. Stack.Size - 1 loop
    Put (Stack.Tab (I) & ", ");
  end loop;
  Put_Line (Stack.Tab (Stack.Size) & "]");
else
  Put_Line ("[");
end if;

S : Character;

begin

----------
-- Main --
----------

for Arg in 1 .. Argument_Count loop
  if Argument (Arg)'Length /= 1 then
    Put_Line ("Argument (Arg) & ", " is an invalid input to the stack.");
  else
    S := Argument (Arg)(Argument (Arg)'First);
    if S = 'd' then
      Debug;
    elsif S = 'p' then
      if not Stack.Empty then
        Stack.Pop (S);
      else
        Put_Line ("Nothing to Pop, Stack is empty!");
      end if;
    else
      if not Stack.Full then
        Stack.Push (S);
      else
        Put_Line ("Could not push ", S & ", Stack is full!");
      end if;
    end if;
end if;

(continues on next page)
end if;
end loop;
end Main;


[SEI-Java] The Software Engineering Institute. SEI CERT Oracle Coding Standard for Java


---


---

2069