Advanced
Journey With
Ada

Gustavo A. Hoffmann
Robert A. Duff

A Flight in Progress

LEARN.
ADACORE.COM
Advanced Journey With Ada: A Flight In Progress

Release 2024-06

Gustavo A. Hoffmann
and Robert A. Duff

Jun 30, 2024
CONTENTS

1 Data types 3

1 Types 5
  1.1 Scalar Types .................................................. 5
    1.1.1 Ranges ................................................... 6
    1.1.2 Predecessor and Successor ............................ 6
    1.1.3 Scalar To String Conversion ......................... 9
    1.1.4 Width attribute ......................................... 10
    1.1.5 Base ...................................................... 11
  1.2 Enumerations .................................................. 15
    1.2.1 Enumerations as functions ............................ 15
    1.2.2 Enumeration overloading ............................... 18
    1.2.3 Position and Internal Code ........................... 22
  1.3 Definite and Indefinite Subtypes ......................... 23
    1.3.1 Constrained Attribute ................................ 31
  1.4 Incomplete types ............................................. 34
  1.5 Type view ..................................................... 35
    1.5.1 Non-Record Private Types ............................. 38
  1.6 Type conversion ............................................. 42
    1.6.1 Value conversion ....................................... 43
    1.6.2 View conversion ....................................... 52
    1.6.3 Implicit conversions ................................... 56
    1.6.4 Conversion of other types ............................. 57
  1.7 Qualified Expressions ....................................... 61
    1.7.1 Verifying subtypes ..................................... 62
  1.8 Default initial values ...................................... 63
  1.9 Deferred Constants .......................................... 66
  1.10 User-defined literals ...................................... 68

2 Types and Representation 77
  2.1 Enumeration Representation Clauses ...................... 77
  2.2 Data Representation ......................................... 79
    2.2.1 Sizes .................................................... 79
    2.2.2 Alignment ............................................... 87
    2.2.3 Overlapping Storage ................................... 90
    2.2.4 Packed Representation ................................ 93
  2.3 Record Representation and storage clauses .............. 97
    2.3.1 Storage Place Attributes ............................... 99
    2.3.2 Using Representation Clauses .......................... 101
    2.3.3 Derived Types And Representation Clauses .......... 103
    2.3.4 Representation on Bit Level ........................... 104
  2.4 Changing Data Representation ............................. 106
    2.4.1 Restrictions ............................................. 110
  2.5 Valid Attribute ............................................... 113
  2.6 Unchecked Union ............................................. 116
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>Shared variable control</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Volatile</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Independent</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Atomic</td>
</tr>
<tr>
<td>2.8</td>
<td>Addresses</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Address attribute</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Address aspect</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Address comparison</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Address to integer conversion</td>
</tr>
<tr>
<td>2.8.5</td>
<td>Address arithmetic</td>
</tr>
<tr>
<td>2.9</td>
<td>Discarding names</td>
</tr>
<tr>
<td>3</td>
<td>Records</td>
</tr>
<tr>
<td>3.1</td>
<td>Default Initialization</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Dependencies</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Initialization Order</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Evaluation</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Defaults and object declaration</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Advanced Usages</td>
</tr>
<tr>
<td>3.2</td>
<td>Mutually dependent types</td>
</tr>
<tr>
<td>3.3</td>
<td>Null records</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Simple Prototyping</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Extending the prototype</td>
</tr>
<tr>
<td>3.3.3</td>
<td>More complex applications</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Implementing the API</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Tagged null records</td>
</tr>
<tr>
<td>3.4</td>
<td>Per-Object Expressions</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Default value</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Restrictions</td>
</tr>
<tr>
<td>4</td>
<td>Aggregates</td>
</tr>
<tr>
<td>4.1</td>
<td>Container Aggregates</td>
</tr>
<tr>
<td>4.2</td>
<td>Record aggregates</td>
</tr>
<tr>
<td>4.2.1</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>4.2.2</td>
<td>others</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Record discriminants</td>
</tr>
<tr>
<td>4.3</td>
<td>Full coverage rules for Aggregates</td>
</tr>
<tr>
<td>4.4</td>
<td>Array aggregates</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Positional and named array aggregates</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Null array aggregate</td>
</tr>
<tr>
<td>4.4.3</td>
<td></td>
</tr>
<tr>
<td>4.4.4</td>
<td>Missing components</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Iterated component association</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Multidimensional array aggregates</td>
</tr>
<tr>
<td>4.4.7</td>
<td>&lt;&gt; and default values</td>
</tr>
<tr>
<td>4.5</td>
<td>Extension Aggregates</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Assignments to objects of derived types</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Example: Points</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Using extension aggregates</td>
</tr>
<tr>
<td>4.5.4</td>
<td>More extension aggregates</td>
</tr>
<tr>
<td>4.5.5</td>
<td>with others</td>
</tr>
<tr>
<td>4.5.6</td>
<td>with null record</td>
</tr>
<tr>
<td>4.5.7</td>
<td>Extension aggregates and descendent types</td>
</tr>
<tr>
<td>4.6</td>
<td>Delta Aggregates</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Delta Aggregates for Tagged Records</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Delta Aggregates for Non-Tagged Records</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Delta Aggregates for Arrays</td>
</tr>
</tbody>
</table>
5 Arrays ........................................... 217
  5.1 Unconstrained Arrays .................................. 217
    5.1.1 Unconstrained Arrays vs. Vectors .................. 218
  5.2 Multidimensional Arrays .................................. 219
    5.2.1 Unconstrained Multidimensional Arrays .............. 224
    5.2.2 Arrays of arrays ................................... 225

6 Strings ........................................... 229
  6.1 Wide and Wide-Wide Strings .............................. 229
    6.1.1 Text I/O ........................................ 231
    6.1.2 Wide and Wide-Wide String Handling ................ 233
    6.1.3 Bounded and Unbounded Wide and Wide-Wide Strings .. 235
  6.2 String Encoding ....................................... 236
    6.2.1 UTF-8 encoding and decoding ......................... 236
    6.2.2 UTF-8 size and length ................................ 238
    6.2.3 UTF-8 encoding in source-code files ................ 240
    6.2.4 UTF-16 encoding and decoding ....................... 244
  6.3 Image attribute ........................................ 246
    6.3.1 Overview ......................................... 246
    6.3.2 Type 'Image and Obj 'Image .......................... 247
    6.3.3 Wider versions of Image ............................ 248
    6.3.4 Image attribute for non-scalar types ................ 249
    6.3.5 Image attribute for tagged types .................... 251
    6.3.6 Image attribute for task and protected types ....... 252
  6.4 Put_ Image aspect ...................................... 253
    6.4.1 Overview ......................................... 254
    6.4.2 Complete Example of Put_ Image ...................... 255
    6.4.3 Relation to the Image attribute ..................... 256
    6.4.4 Put_ Image and derived types ........................ 257
    6.4.5 Put_ Image and tagged types ......................... 259
  6.5 Universal text buffer .................................... 261
    6.5.1 Overview ......................................... 261
    6.5.2 Additional procedures .............................. 262

7 Numerics .......................................... 265
  7.1 Modular Types ......................................... 265
    7.1.1 Modulus Attribute .................................. 265
    7.1.2 Mod Attribute ..................................... 266
    7.1.3 Operations on modular types .......................... 268
  7.2 Numeric Literals ....................................... 270
    7.2.1 Classification ..................................... 270
    7.2.2 Features and Flexibility ............................ 272
  7.3 Floating-Point Types ................................... 277
    7.3.1 Representation-oriented attributes .................. 277
    7.3.2 Primitive function attributes ...................... 283
  7.4 Fixed-Point Types .................................... 291
    7.4.1 Attributes of fixed-point types ..................... 291
    7.4.2 Attributes of decimal fixed-point types ............ 297
  7.5 Big Numbers .......................................... 299
    7.5.1 Overview ......................................... 300
    7.5.2 Factorial ......................................... 302
    7.5.3 Conversions ....................................... 305
    7.5.4 Other features of big integers ...................... 312
    7.5.5 Other operators for big integers .................... 313
    7.5.6 Big real and quotients ............................. 314
    7.5.7 Range checks ...................................... 315
## II Control Flow

### 8 Expressions

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Expressions: Definition</td>
<td>319</td>
</tr>
<tr>
<td>8.1.1 Relations and simple expressions</td>
<td>319</td>
</tr>
<tr>
<td>8.1.2 Numeric expressions</td>
<td>322</td>
</tr>
<tr>
<td>8.1.3 Other expressions</td>
<td>323</td>
</tr>
<tr>
<td>8.1.4 Parenthesized expression</td>
<td>323</td>
</tr>
<tr>
<td>8.2 Conditional Expressions</td>
<td>326</td>
</tr>
<tr>
<td>8.3 Quantified Expressions</td>
<td>328</td>
</tr>
<tr>
<td>8.4 Declare Expressions</td>
<td>332</td>
</tr>
<tr>
<td>8.4.1 Restrictions in the declarative part</td>
<td>334</td>
</tr>
<tr>
<td>8.5 Reduction Expressions</td>
<td>336</td>
</tr>
<tr>
<td>8.5.1 Value sequences</td>
<td>338</td>
</tr>
<tr>
<td>8.5.2 Custom reducers</td>
<td>339</td>
</tr>
<tr>
<td>8.5.3 Other accumulator types</td>
<td>341</td>
</tr>
</tbody>
</table>

### 9 Statements

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Simple and Compound Statements</td>
<td>343</td>
</tr>
<tr>
<td>9.2 Labels</td>
<td>343</td>
</tr>
<tr>
<td>9.2.1 Labels and goto statements</td>
<td>344</td>
</tr>
<tr>
<td>9.2.2 Use-case: Continue</td>
<td>345</td>
</tr>
<tr>
<td>9.2.3 Labels and compound statements</td>
<td>346</td>
</tr>
<tr>
<td>9.3 Exit loop statement</td>
<td>348</td>
</tr>
<tr>
<td>9.4 If, case and loop statements</td>
<td>349</td>
</tr>
<tr>
<td>9.4.1 Case statements and expressions</td>
<td>351</td>
</tr>
<tr>
<td>9.5 Block Statements</td>
<td>353</td>
</tr>
<tr>
<td>9.6 Extended return statement</td>
<td>354</td>
</tr>
<tr>
<td>9.6.1 Other usages of extended return statements</td>
<td>355</td>
</tr>
</tbody>
</table>

### 10 Subprograms

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Parameter Modes and Associations</td>
<td>357</td>
</tr>
<tr>
<td>10.1.1 Formal Parameter Modes</td>
<td>357</td>
</tr>
<tr>
<td>10.1.2 By-copy and by-reference</td>
<td>358</td>
</tr>
<tr>
<td>10.1.3 Bounded errors</td>
<td>363</td>
</tr>
<tr>
<td>10.1.4 Aliased parameters</td>
<td>365</td>
</tr>
<tr>
<td>10.1.5 Parameter Associations</td>
<td>366</td>
</tr>
<tr>
<td>10.2 Operators</td>
<td>370</td>
</tr>
<tr>
<td>10.2.1 User-defined operators</td>
<td>370</td>
</tr>
<tr>
<td>10.3 Expression functions</td>
<td>376</td>
</tr>
<tr>
<td>10.4 Overloading</td>
<td>379</td>
</tr>
<tr>
<td>10.5 Operator Overloading</td>
<td>384</td>
</tr>
<tr>
<td>10.6 Operator Overriding</td>
<td>384</td>
</tr>
<tr>
<td>10.7 Nonreturning procedures</td>
<td>387</td>
</tr>
<tr>
<td>10.8 Inline subprograms</td>
<td>390</td>
</tr>
<tr>
<td>10.9 Null Procedures</td>
<td>392</td>
</tr>
<tr>
<td>10.9.1 Null procedures and overriding</td>
<td>393</td>
</tr>
</tbody>
</table>

### 11 Exceptions

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Asserts</td>
<td>397</td>
</tr>
<tr>
<td>11.2 Assertion policies</td>
<td>399</td>
</tr>
<tr>
<td>11.3 Checks and exceptions</td>
<td>402</td>
</tr>
<tr>
<td>11.3.1 Access Check</td>
<td>402</td>
</tr>
<tr>
<td>11.3.2 Discriminant Check</td>
<td>404</td>
</tr>
<tr>
<td>11.3.3 Division Check</td>
<td>405</td>
</tr>
<tr>
<td>11.3.4 Index Check</td>
<td>406</td>
</tr>
<tr>
<td>11.3.5 Length Check</td>
<td>407</td>
</tr>
<tr>
<td>11.3.6 Overflow Check</td>
<td>407</td>
</tr>
<tr>
<td>11.3.7 Range Check</td>
<td>408</td>
</tr>
</tbody>
</table>
## Modular programming

### 12 Packages

- **12.1 Package renaming**
  - 12.1.1 Grouping packages
  - 12.1.2 Child of renamed package
  - 12.1.3 Backwards-compatibility via renaming
- **12.2 Private packages**
  - 12.2.1 Declaration and usage
  - 12.2.2 Private sibling packages
  - 12.2.3 Outside the package tree
- **12.3 Private with clauses**
  - 12.3.1 Definition and usage
  - 12.3.2 Referring to private child package
- **12.4 Limited Visibility**
  - 12.4.1 Limited visibility and private with clauses
  - 12.4.2 Limited visibility and other elements
- **12.5 Visibility**
  - 12.5.1 Automatic visibility
  - 12.5.2 With clauses and visibility
  - 12.5.3 Circular dependency
  - 12.5.4 Private packages
- **12.6 Use type clause**
  - 12.6.1 Another use clause example
  - 12.6.2 Visibility and Readability
  - 12.6.3 use type
  - 12.6.4 use all type
- **12.7 Use clauses and naming conflicts**
  - 12.7.1 Code example
  - 12.7.2 Naming conflict
  - 12.7.3 Circumventing naming conflicts

### 13 Subprograms and Modularity

- **13.1 Private subprograms**

### 14 Access Types

- **14.1 Access types: Terminology**
  - 14.1.1 Access type, designated subtype and profile
14.1.2 Access object and designated object ........................................ 482
14.1.3 Access value and designated value ........................................ 483
14.2 Access types: Allocation ............................................................ 483
  14.2.1 Pool-specific access types ................................................ 486
  14.2.2 Multiple allocation .......................................................... 488
14.3 Discriminants as Access Values ................................................ 492
  14.3.1 Unconstrained type as designated subtype ................................ 494
  14.3.2 Whole object assignments ................................................ 497
14.4 Parameters as Access Values .................................................... 498
  14.4.1 Changing the referenced object .......................................... 500
  14.4.2 Replace the access value .................................................... 502
  14.4.3 Side-effects on designated objects ....................................... 503
14.5 Self-reference ........................................................................ 509
14.6 Mutually dependent types using access types ............................... 512
14.7 Dereferencing ....................................................................... 512
  14.7.1 Implicit Dereferencing ....................................................... 514
14.8 Ragged arrays ........................................................................... 519
  14.8.1 Uniform multidiimensional arrays ....................................... 519
  14.8.2 Non-uniform multidiimensional array ................................... 521
14.9 Aliasing .................................................................................... 523
  14.9.1 Aliased objects .................................................................... 525
  14.9.2 Aliased components .............................................................. 530
  14.9.3 Aliased parameters .............................................................. 532
14.10 Accessibility Levels and Rules: An Introduction ............................. 534
  14.10.1 Lifetime of objects .............................................................. 534
  14.10.2 Accessibility Levels ............................................................. 535
  14.10.3 Accessibility Rules .............................................................. 536
  14.10.4 Accessibility rules on parameters .......................................... 540
  14.10.5 Dangling References ........................................................... 541
14.11 Unchecked Access ................................................................. 544
14.12 Unchecked Deallocation ............................................................ 546
  14.12.1 Unchecked Deallocation and Dangling References .................... 549
  14.12.2 Dereferencing dangling references ....................................... 551
  14.12.3 Restrictions for Ada.Unchecked_Deallocation ........................ 552
14.13 Null & Not Null Access ............................................................. 554
14.14 Design strategies for access types .............................................. 558
  14.14.1 Abstract data type for access types ...................................... 558
  14.14.2 Controlled type for access types .......................................... 561
14.15 Access to subprograms ............................................................. 566
  14.15.1 Static vs. dynamic calls ....................................................... 566
  14.15.2 Access to subprogram declaration ....................................... 567
  14.15.3 Objects of access-to-subprogram type .................................. 569
  14.15.4 Components of access-to-subprogram type ............................ 570
  14.15.5 Access-to-subprogram as discriminant types .......................... 573
  14.15.6 Access-to-subprograms as formal parameters ....................... 575
  14.15.7 Selecting subprograms ......................................................... 578
  14.15.8 Null exclusion ................................................................. 580
  14.15.9 Access to protected subprograms ........................................ 585
14.16 Accessibility Rules and Access-To-Subprograms ............................ 591
  14.16.1 Unchecked Access .............................................................. 593
14.17 Access and Address ................................................................. 595
  14.17.1 Address and access conversion .......................................... 596

15 Anonymous Access Types .......................................................... 601
  15.1 Named and Anonymous Access Types ....................................... 601
    15.1.1 Relation to named types .................................................... 602
    15.1.2 Benefits of anonymous access types ................................... 602
  15.2 Anonymous Access-To-Object Types ......................................... 605
15.2.1 Not Null Anonymous Access-To-Object Types .............................. 607
15.2.2 Drawbacks of Anonymous Access-To-Object Types .................. 608
15.3 Access discriminants .......................................................... 615
  15.3.1 Default Value of Access Discriminants .............................. 617
  15.3.2 Benefits of Access Discriminants ................................. 619
  15.3.3 Preventing dangling pointers .......................................... 621
15.4 Self-reference ...................................................................... 622
15.5 Mutually dependent types using anonymous access types ............. 624
15.6 Access parameters .............................................................. 625
  15.6.1 Interfacing To Other Languages ...................................... 628
  15.6.2 Inherited Primitive Operations For Tagged Types ................. 631
15.7 User-Defined References ...................................................... 634
  15.7.1 Dereferencing of tagged types ......................................... 636
  15.7.2 Simple container ......................................................... 637
15.8 Anonymous Access Types and Accessibility Rules ..................... 643
  15.8.1 Conversions between Anonymous and Named Access Types .... 645
  15.8.2 Accessibility rules on access parameters ........................... 647
15.9 Anonymous Access-To-Subprograms ...................................... 648
  15.9.1 Examples of anonymous access-to-subprogram usage .......... 650
  15.9.2 Application of anonymous access-to-subprogram type .......... 656
  15.9.3 Readability ............................................................... 656
15.10 Accessibility Rules and Anonymous Access-To-Subprograms .......... 658
  15.10.1 Named vs. anonymous access-to-subprograms ................... 658
  15.10.2 Named vs. anonymous access-to-subprograms as parameters ... 659
  15.10.3 Iterator ................................................................. 663

16 Limited Types ................................................................. 669
  16.1 Assignment and equality ................................................... 669
  16.1.1 Assignments .................................................................. 672
  16.1.2 Equality ...................................................................... 674
  16.2 Limited private types ....................................................... 677
  16.2.1 Non-Record Limited Types ......................................... 678
  16.2.2 Partial and full view of limited types ............................. 679
  16.2.3 Limited and nonlimited in full view ............................... 682
  16.2.4 Limited private component ......................................... 683
  16.2.5 Tagged limited private types ....................................... 685
  16.3 Explicitly limited types .................................................... 685
  16.4 Subtypes of Limited Types ................................................ 687
  16.5 Deriving from limited types .............................................. 688
  16.5.1 Deriving from limited private types ............................... 689
  16.5.2 Deriving from non-explicitly limited private types .......... 690
  16.6 Immutably Limited Types ................................................ 695
  16.6.1 Non immutably limited types ....................................... 698
  16.7 Limited Types and Discriminants ....................................... 698
  16.8 Record components of limited type ................................... 698
  16.9 Limited types and aggregates ......................................... 699
  16.9.1 Full coverage rules for limited types ............................ 700
  16.10 Constructor functions for limited types ............................. 703
  16.11 Return objects ............................................................. 707
  16.11.1 Extended return statements for limited types ............... 707
  16.11.2 Initialization and function return .............................. 709
  16.12 Building objects from constructors ................................... 713
  16.13 Limited types as parameter .......................................... 716
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[^2]. 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).

This course will teach you advanced topics of the Ada programming language. The Introduction to Ada\(^3\) 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.

---

\(^3\) https://learn.adacore.com/courses/intro-to-ada/index.html#intro-ada-course-index
Part I

Data types
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\(^4\) and image attribute\(^5\)). 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\(^6\)

\(^4\) https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-range-attribute
\(^5\) https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-image-attribute
\(^6\) http://www.ada-auth.org/standards/22rm/html/RM-3-5.html
1.1.1 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:

```
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;
```

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).

1.1.2 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:

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

procedure Show_Succ_Pred_Discrete is
  type State is (Idle, Started, Processing, Stopped);
  Machine_State : constant State := Started;
  I : 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);
end Show_Succ_Pred_Discrete;
```
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;

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):

Listing 3: show_succ_pred_real.adb

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);
  end loop;
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 ("----- Pred -------");

for I in 1 .. 5 loop
  Put_Line (N'Image);
  N := My_Float'Pred (N);
end loop;

for I in 1 .. 5 loop
  Put_Line (N'Image);
  N := My_Float'Succ (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 -------
  0.000000E+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's 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.

### 1.1.3 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:

```ada
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></td>
<td>Wide_Wide_Image</td>
<td>Wide_Wide_String</td>
</tr>
<tr>
<td>Conversion to subtype</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 229).
1.1.4 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:

Listing 5: show_width_attr.adb

```ada
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`. 
### 1.1.5 Base

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`

```ada
package My_Integers is
    subtype One_To_Ten is Integer
        range 1 .. 10;
end My_Integers;
```

**Code block metadata**

MD5: e3f8310ed742e61a65728fecb6caaa557

If we then use the Base attribute — by writing `One_To_Ten'Base` —, we're actually referring to the unconstrained underlying hardware representation selected for `One_To_Ten`. As `One_To_Ten` is a subtype of the `Integer` type, this also means that `One_To_Ten'Base` is equivalent to `Integer'Base`, i.e. they refer to the same base type. (This base type is the underlying hardware type representing the `Integer` type — but is not the `Integer` type itself.)

**For further reading...**

The Ada standard defines that the minimum range of the `Integer` type is 
\[-2^{**15} \div 1 \ldots 2^{**15} \div 1\]. In modern 64-bit systems — where wider types such as `Long_Integer` are defined — the range is at least 
\[-2^{**31} \div 1 \ldots 2^{**31} \div 1\]. Therefore, we could think of the `Integer` type as having the following declaration:

```ada
type Integer is
    range -2 ** 31 .. 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,

```ada
type Integer is
    range -2 ** 31 .. 2 ** 31 - 1;
```

is really as if we said this:

```ada
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)
32 & One_To_Ten'Last/Image);
33   Put_Line ("One_Ten'Base'Last: ",
34 & One_To_Ten'Base'Last/Image);
35 end Show_Base;

Code block metadata

MD5: ce3e9fb3f1619e835e9108ae0a787e7

Build output

show_base.adb:13:22: warning: value not in range of type "One_To_Ten" defined at
my_integers.ads:3 [enabled by default]
show_base.adb:13:22: warning: Constraint_Error will be raised at run time [enabled,
by default]

Runtime output

Increasing value for One_To_Ten...
Exception raised!
Increasing value for One_Ten'Base...
One_Ten'Last: 10
One_Ten'Base'Last: 2147483647

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_Ten'Succ (One_Ten'Last). As
expected, since the last value of the range doesn't have a successor, a constraint ex-
ception is raised.

In the Using_Base block, we're declaring a variable V of One_Ten'Base subtype. In
this case, the next value exists — because the condition One_Ten'Last + 1 <=
One_Ten'Base'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 con-
straint errors:

Listing 9: my_integers.ads

package My_Integers is

   subtype One_To_Ten is Integer range 1 .. 10;

   function Sat_Add (V1, V2 : One_Ten'Base) return One_Ten;
   function Sat_Sub (V1, V2 : One_Ten'Base) return One_Ten;

end My_Integers;

Listing 10: my_integers.adb

-- with Ada.Text_IO; use Ada.Text_IO;
package body My_Integers is

   function Saturate (V : One_Ten'Base) return One_Ten is
      begin
         return One_Ten;
      end Saturate;

(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.

### 1.2 Enumerations

We've introduced enumerations back in the Introduction to Ada course. 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

#### 1.2.1 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:

```ada
package Days is
  type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  -- Essentially, we're declaring
  -- these functions:
  --
```

[7](https://learn.adacore.com/courses/intro-to-ada/chapters/strongly_typed_language.html#intro-ada-enum-types)

[8](http://www.ada-auth.org/standards/22rm/html/RM-3-5-1.html)
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
package Enumeration_Example is

  type Day is (Mon, Tue, Wed,
               Thu, Fri,
               Sat, Sun);

  function Monday    return Day renames Mon;
  function Tuesday   return Day renames Tue;
  function Wednesday return Day renames Wed;
  function Thursday  return Day renames Thu;
  function Friday    return Day renames Fri;
  function Saturday  return Day renames Sat;
  function Sunday    return Day renames Sun;

end Enumeration_Example;
```

Now, we can use both Monday or Mon to refer to Monday of the Day type:

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Enumeration_Example; use Enumeration_Example;
```
procedure Show_Renaming is
   D1 : constant Day := Mon;
   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;

(continues on next page)
function Saturday return Day renames
   Enumeration_Example.Sat;
function Sunday return Day renames
   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;

1.2.2 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:

package Colors is

   type Color is
      (Salmon,
       Firebrick,
       Red,
       Darkred,
       Lime,
       Forestgreen,
       Green,
       Darkgreen,
       Blue,
       Mediumblue,
       Darkblue);

   type Primary_Color is
      (Red,
       Green,
       Blue);

Listing 18: colors.ads
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:

```ada
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;
```

When assigning Red to C1 and C2, it is clear that, in the first case, we're referring to Red of Color type, while in the second case, we're referring to Red of the Primary_Color type. 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:

```ada
package Colors.Shades is
  subtype Blue_Shades is
    Colors range Blue .. Darkblue;
end Colors.Shades;
```

1.2. Enumerations
end Colors.Shades;

Code block metadata
MD5: 9c13508bda487cae02dbf8b403271540

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:

Listing 21: colors.ads

```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;
```

Listing 22: color_loop.adb

```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;
```

Code block metadata
MD5: 82d0d3f28f1fa6b296a4f44db71f41b

Build output
Here, it's not clear whether the range in the loop is of \textit{Color} type or of \textit{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 \texttt{Put\_Line} — gives us a hint about the developer's intention to refer to the \textit{Color} type. In this case, we can use qualification — for example, \texttt{Color'(Red)} — to resolve the ambiguity:

\begin{verbatim}
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'\texttt{Image} (C));
  end loop;
end Color\_Loop;
\end{verbatim}

Note that, in the case of ranges, we can also rewrite the loop by using a range declaration:

\begin{verbatim}
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'\texttt{Image} (C));
  end loop;
end Color\_Loop;
\end{verbatim}
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:

```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;
```

1.2.3 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:

```ada
package Days is
  type Day is (Mon, Tue, Wed,
               Thu, Fri,
               Sat, Sun);
end Days;
```

```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))
  end loop;
end Show_Days;
```
& " position = 
& Integer'Image (Day'Pos (D))); 
Put_Line (Day'Image (D) 
& " internal code = 
& Integer'Image 
(D'Enum_Rep (D))); 
end loop; 
end Show_Days;

1.3 Definite and Indefinite Subtypes

Indefinite types were mentioned back in the Introduction to Ada course. 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:

---

9 https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-indefinite-subtype
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;

end Unconstrained_Types;

Code block metadata

锦标: Indefinite_Types
MD5: e569dc7315b0b834c9315b14d46c0ac79

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:

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;

Code block metadata

锦标: Indefinite_Types_Error
MD5: cf73d308ddbb4a8c2503146ecd550a791

Build output

unconstrained_types.ads:6:30: error: discriminants must have a discrete or access
...

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 492).

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;
```

(continues on next page)
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:

Listing 32: using_unconstrained_type.adb

```ada
with Unconstrained_Types; use Unconstrained_Types;

procedure Using_Unconstrained_Type is
  subtype Integer_Array_5 is
    Integer_Array (1 .. 5);
  A1 : Integer_Array_5;
  A2 : Integer_Array (1 .. 5);

  subtype Simple_Record_Ext is
    Simple_Record (Extended => True);
  R1 : Simple_Record_Ext;
  R2 : Simple_Record (Extended => True);

begin
  null;
end Using_Unconstrained_Type;
```

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:

Listing 33: show_integer_array.ads

```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
   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

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.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
```

(continues on next page)
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 Using_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;

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_Header (A_5, "First example");
  Show_Integer_Array_Header (A_10, "Second example");
 end Using_Unconstrained_Type;
```
Indefinite Types

MD5: dd09f8c4089c6ad4c18410879f80f731

Runtime output

First example
1: 1
2: 2
3: 3
4: 4
5: 5
--------

Second example
1: 1
2: 2
3: 3
4: 4
5: 5
6: 99
7: 99
8: 99
9: 99
10: 99
--------

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
```

(continues on next page)
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

with Unconstrained_Types; use Unconstrained_Types;

with Show_Integer_Array_Plus;

procedure Using_Unconstrained_Type is
   A_5 : constant Integer_Array (1 .. 5) := (1, 2, 3, 4, 5);
begin
   Show_Integer_Array_Plus (A_5, 5);
end Using_Unconstrained_Type;

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\(^{10}\)

\(^{10}\) http://www.ada-auth.org/standards/22rm/html/RM-3-3.html
1.3.1 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:

Listing 42: unconstrained_types.ads

```ada
package Unconstrained_Types is
  type Simple_Record is (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

```ada
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:

```
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);
  Show_Rs;
end Using_Constrained_Attribute;
```

**Code block metadata**

MD5: f7517fc3d3c68a784f5506f188d4e7bb

**Runtime output**
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:

```
-- [...]
begin
  Put_Line ("---- Reset: R'Constrained => 
& R'Constrained'Image);
  R := Zero_Extended;
end Reset;
```

Running the code now generates a runtime exception:

```
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 constrained. 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

---

11 http://www.ada-auth.org/standards/22rm/html/RM-3-7-2.html
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;
  -- Incomplete type declaration!

  type R is record
    I : Integer;
  end record;
  -- type R is now complete!

end Incomplete_Type_Example;
```

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:

```ada
package Incomplete_Type_Example is
  private
    type R;
    -- Incomplete type declaration!

end Incomplete_Type_Example;
```

```ada
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:

```ada
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;
```

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations

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;
```

12 http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html
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:

Listing 50: partial_full_views.ads

```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;
```

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:

Listing 51: main.adb

```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:

```
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;

end Limited_Private_Example;
```

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:

```
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.
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

1.5.1 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:

Listing 54: private_integers.ads

```
package Private_Integers is

  -- Partial view of private Integer type:
  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;

(continues on next page)
```

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: 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;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: 5933779ce5f0802b448df96c42e65a8d

Build output

show_private_integers.adb:4:07: warning: variable "B" is read but never assigned [-gnatw]
show_private_integers.adb:6:09: warning: "A" may be referenced before it has a value [enabled by default]

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

```ada
package Private_Arrays is

  type Private_Unconstrained_Array is private;

  type Private_Constrained_Array (L : Positive) is private;

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;

  -- NOTE: using an array type fails as well:
  --
  -- type Private_Constrained_Array
  -- (L : Positive) is
  --   Integer_Array (1 .. L);

end Private_Arrays;
```

Build output

private_arrays.ads:13:09: error: full view of "Private_Unconstrained_Array" not compatible with declaration at line 3
private_arrays.ads:13:09: error: one is constrained, the other unconstrained
private_arrays.ads:17:07: error: elementary or array type cannot have discriminants
gprbuild: *** compilation phase failed

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

```ada
package Private_Arrays is

  type Private_Constrained_Array (L : Positive) is private;
```

(continues on next page)
private

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

    type Private_Constrained_Array
       (L : Positive) is
       record
          Arr : Integer_Array (L .. 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;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Array
MD5: 3830721499a59d85efddd4989aa7c288

Build output

declare_private_array.adb:4:03: warning: variable "Arr" is never read and never assigned [-gnatwv]

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:

Listing 61: private_arrays.ads

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;

1.5. Type view
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.

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. 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 319);
- 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

---

14 https://learn.adacore.com/courses/intro-to-ada/chapters/strongly_typed_language.html#intro-ada-type-conversion
1.6. Type conversion

1.6.1 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:

Listing 64: custom_integers.ads

```ada
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;
```

Listing 65: 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 := 1;
  Integer_Var : Integer := 2;
begin
  -- Int to Integer conversion
  Integer_Var := Integer (Int_Var);
  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

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

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

procedure Show_Conversion is
  Int_Var : Int := Int (Integer_Var);
begin
  Integer_Var := 0;
  Int_Var := Int (Integer_Var);
  Put_Line ("Int_Var : ", Int_Var'Image);
end Show_Conversion;
```

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

```ada
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

```ada
package Fixed_PointDefs 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_PointDefs;
```

Listing 69: show_conversion.adb

```ada
with Fixed_PointDefs; use Fixed_PointDefs;
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);
```

(continues on next page)
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:

```
package Custom_Enumerations is

    type Priority is (Low, Mid, High);

    type Important_Priority is new
        Priority range Mid .. High;

end Custom_Enumerations;
```

```
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);
```

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;
```

In this example, an exception is raised because Low is not in the Important_Priority type's range.

1.6. Type conversion
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
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Arrays; use Custom_Arrays;

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

- 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);
```

(continues on next page)
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 AF_1 := Float_Array_1 (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: [for I in AF_1'Range => Float (AI_1 (I))]; (We discuss this topic later in another chapter (page 182).)

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

```ada
generic
  type T is private;
package Custom_Arrays is
  type T_Array is
    array (Positive range <>) of T;
end Custom_Arrays;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Generic_Array_Type_Conversion
MD5: 8b3a963a1292a90d99d83c6d81ce3995

We could instantiate this generic package and reuse parts of the previous code example:

Listing 79: show_conversion.adb

```ada
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;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Generic_Array_Type_Conversion
MD5: 956186d864763924b93b6a9d807525b6

Runtime output

```
AI_1: [ 1, 2, 3]
AI_2: [ 4, 5, 6]
```

As we can see in this example, each of the instantiated CA_Int_1 and CA_Int_2 packages

1.6. Type conversion
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.

### 1.6.2 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:

```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;
```

```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 & "=” & 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;
```

We can use the types from the Points package and convert between each other:
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

procedure Double (X : in out Float);

Listing 85: double.adb

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: 31f4409d9faeaf213c5940de65eeb014

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:
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: 2256d3c120d569789dcdf4c959ed9f0f

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:

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: 3b90caf78952710ee4214a7b60968

Runtime output

1.6. Type conversion
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.

### 1.6.3 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: dbbe498fa66701ca94f48119b1bca91
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.

### 1.6.4 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;
end
```

**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;
```

### 1.6. Type conversion
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;
    }

    // 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:

```
package Custom_Defs is
  type T1 is private;
  function Init (X : Float)
end Custom_Defs;
```

(continues on next page)
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
   end Display;

private
   type T1 is record
      X : Float;
   end record;

   function Init (X : Float) return T1 is (X => X);

   type T3 is record
      X, Y, Z : Float;
   end record;

   function Init (X, Y, Z : Float) return T3 is (X => X, Y => Y, Z => Z);

end Custom_Defs;

Listing 93: custom_defs.adb
Listing 94: show_conversion.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Defs; use Custom_Defs;
procedure Show_Conversion is
   T_3 : constant T3 := Init (0.5, 0.4, 0.6);
   T_1 : T1 := Init (0.0);
   F   : Float;
begin
   -- Explicit conversion from
   -- T3 to Float type
   F := To_Float (T_3);
   Put_Line ("F : " & F'Image);
   -- Explicit conversion from
   -- T3 to T1 type
   T_1 := To_T1 (T_3);
   Put ("T_1 : ");
   Display (T_1);
end Show_Conversion;
```

Code block metadata

MD5: b3e7be5488fb8026b4386063ba16aeb

Runtime output

F : 5.00000E-01
T_1 : (X => 5.00000E-01)

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.)

### 1.7 Qualified Expressions

We already saw qualified expressions in the *Introduction to Ada* 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:

Listing 95: simple_integers.ads

```ada
package Simple_Integers is
```

---

16 https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_types.html#intro-ada-qualified-expressions
**Advanced Journey With Ada: A Flight In Progress**

(continued from previous page)

```ada
3  type Int is new Integer;
4
5  subtype Int_Not_Zero is Int
6    with Dynamic_Predicate => Int_Not_Zero /= 0;
7
8 end Simple_Integers;
```

Listing 96: show_qualified_expressions.adb

```ada
1 with Simple_Integers; use Simple_Integers;
2 procedure Show_Qualified_Expressions is
3   I : Int;
4 begin
5   -- Using qualified expression Int'(N)
6   I := Int'(0);
7 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

### 1.7.1 Verifying subtypes

**Note:** This feature was introduced in Ada 2022.

We can use qualified expressions to verify a subtype's predicate:

```ada
1 with Simple_Integers; use Simple_Integers;
2 procedure Show_Qualified_Expressions is
3   I : Int;
4 begin
5   I := Int_Not_Zero'(0);
6 end Show_Qualified_Expressions;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example

MD5: 3c4ab8ad7bf75ae029047f673aa15d70

**Build output**

show_qualified_expressions.adb:6:23: warning: expression fails predicate check on "Int_Not_Zero" [enabled by default]
show_qualified_expressions.adb:6:23: warning: check will fail at run time [-gnatw.a]

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 sub-type. (This predicate check fails at runtime.)

### 1.8 Default initial values

In the Introduction to Ada course\(^\text{18}\), 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: e6cd8261b099278ceeb5fda91d318f6e

\(^{18}\) https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-default-values
Note that we cannot specify a default value for a subtype:

```ada
package Defaults is

     subtype T is Integer
         with Default_Value => -1;

-- ERROR!!

end Defaults;
```

**Code block metadata**

MD5: beef68e4a7a3714cfa3e547bdcda9a0c

**Build output**

defaults.ads:4:11: error: aspect "Default_Value" cannot apply to subtype
gprbuild: *** compilation phase failed

For array types, we use the Default_Component_Value aspect:

```ada
package Defaults is

    type Arr is
        array (Positive range <>) of Integer
         with Default_Component_Value => -1;

end Defaults;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_4
MD5: 2c390e3900e4af42498381025a37955e

This is a package containing the declarations we've just seen:

```ada
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;
```

(continues on next page)
type Arr is
   array (Positive range <>) of Integer
   with Default_Component_Value => -1;
end Defaults;

Code block metadata

MD5: e9263ff5b96523c129a3d2d9bb5a4dd

In the example below, we declare variables of the types from the Defaults package:

Listing 103: use_defaults.adb

```ada
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;
```

Code block metadata

MD5: f8e55d31c6da2447fe14eb07eaad1975

Runtime output

Enumeration: E1
Integer type: -1
Record type: 1, 10
Array type: -1 -1 -1 -1 -1

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

  http://www.ada-auth.org/standards/22rm/html/RM-3-5.html

1.8. Default initial values
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:

```
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;
```

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:

```
double light = 299792458.0;
```

Then, we can import this constant in the Deferred_Constants package:

```
package Deferred_Constants is
  -- (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:

```ada
package Deferred_Constants is
  type Speed is new Long_Float;
  subtype Positive_Speed is Speed range 0.0 .. Speed'Last;
  Light : constant Speed with Import, Convention => C;
    -- ^ deferred constant declaration
    -- declaration; imported from C file
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:
package Deferred_Constants is
  type Speed is new Long_Float;
  Light : constant Speed;
    ^ 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;

Here, we call the Calculate_Light function — declared in the private part of the Deferred_Constants package — for the full declaration of the Light constant.

In the Ada Reference Manual
  • 7.4 Deferred Constants

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;

Code block metadata

21 http://www.ada-auth.org/standards/22rm/html/RM-7-4.html
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:

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

procedure Simple_Enumeration is
  type Activation_State is (Unknown, Off, On);
  S : Activation_State;
begin
  S := On;
  Put_Line (Activation_State'Image (S));
end Simple_Enumeration;
```

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

```ada
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;
```

Code block metadata

MD5: 67b6d96f049ab6cde962aefda96bffca

Based on this specification, we can now use an integer literal to initialize an object S of Activation_State type:

```ada
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

```ada
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;
```

Code block metadata

MD5: 104a835915b93ea3b860bce03fd709a3

Let's look at a complete example that makes use of all three kinds of literals:

Listing 113: activation_states.ads

```ada
package Activation_States is
  type Activation_State is (Unknown, Off, On)
end Activation_States;
```
with String_Literal =>
    To_Activation_State,
Integer_Literal =>
    Integer_To_Activation_State,
Real_Literal =>
    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;
end if;
end Real_To_Activation_State;
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
else
  return On;
end if;
end Real_To_Activation_State;
end Activation_States;
```

Listing 115: activation_examples.adb

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

procedure Activation_Examples is
  S : Activation_State;
begin
  S := "Off";
  Put_Line ("String: Off => " & Activation_State'Image (S));

  S := 1;
  Put_Line ("Integer: 1 => " & Activation_State'Image (S));

  S := 1.5;
  Put_Line ("Real: 1.5 => " & Activation_State'Image (S));
end Activation_Examples;
```

Code block metadata

| Project: Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.Activation_States | MD5: 186b7b898e4c16b6fd0dcd683e8f0379d |

Runtime output

<table>
<thead>
<tr>
<th>String: Off =&gt; OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer: 1 =&gt; ON</td>
</tr>
<tr>
<td>Real: 1.5 =&gt; ON</td>
</tr>
</tbody>
</table>

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: 6ca6aa79b8b058801688fc2dfb186091

#### 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;
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:

1.10. User-defined literals  73
Listing 118: 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: 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
(continues on next page)
```
function To_Silly (S : Wide_Wide_String) 
  return Silly 
is 
begin 
  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; 

Code block metadata

MD5: 2a077045f058a8d5c09c43f66fc128be

Runtime output

42, 4.20000E+01

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

1.10. User-defined literals

22 http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html
2.1 Enumeration Representation Clauses

We have talked about the internal code of an enumeration in another section (page 22). 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:

```ada
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;
```

```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 = "
```
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;
```

Listing 3: days.ads
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 course.

In the Ada Reference Manual

- 13.2 Packed Types
- 13.3 Operational and Representation Attributes
- 13.5.3 Bit Ordering

2.2.1 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 4: 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 5: 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: " & UInt_7'Image);
  Put_Line ("UInt_7'Object_Size: " & UInt_7'Object_Size'Image);
  Put_Line ("V1'Size: " & V1'Size'Image);
  New_Line;

  Put_Line ("UInt_7_S32'Size: " & UInt_7_S32'Size'Image);
  Put_Line ("UInt_7_S32'Object_Size: " & UInt_7_S32'Object_Size'Image);
  Put_Line ("V2'Size: " & V2'Size'Image);

end Show_Sizes;
```

Depending on your target architecture, you may see this output:
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:

```
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 7: 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_1'Size: " & Arr_1/Size'Image);
  New_Line;

  Put_Line ("UInt_7_Array_Comp_32'Object_Size: " & UInt_7_Array_Comp_32/Object_Size'Image);
  Put_Line ("UInt_7_Array_Comp_32'Object_Size: " & UInt_7_Array_Comp_32/Object_Size'Image);
  Put_Line ("UInt_7_Array_Comp_32'Component_Size: " & UInt_7_Array_Comp_32/Component_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;
```

Code block metadata

Sizes
MD5: e316bcb827e014075dfbf044935827ae

Build output

show_sizes.adb:6:04: warning: variable "Arr_1" is read but never assigned [-gnatwv]
show_sizes.adb:7:04: warning: variable "Arr_2" is read but never assigned [-gnatwv]

Runtime output

UInt_7_Array'Size: 17179869176
UInt_7_Array'Object_Size: 17179869176
(continues on next page)
Depending on your target architecture, you may see this output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Object Size</th>
<th>Component Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint_7_Array</td>
<td>17179869176</td>
<td>17179869176</td>
<td>8</td>
</tr>
<tr>
<td>Arr_1</td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Arr_1'Size</td>
<td>160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uint_7_Array_Comp_32'</td>
<td>68719476704</td>
<td>68719476704</td>
<td>32</td>
</tr>
<tr>
<td>Arr_2</td>
<td>32</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Arr_2'Size</td>
<td>640</td>
<td></td>
<td></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:

```ada
type Uint_7_Array_Comp_32 is
  array (Positive range <>) of Uint_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 8: custom_types.ads

```ada
package Custom_Types is
  type UInt_7 is range 0 .. 127;
  type UInt_7_Access is access UInt_7;
end Custom_Types;
```

Listing 9: show_sizes.adb

```ada
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: "
     & 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

Sizes

(continues on next page)
2.2. Data Representation

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_7Reserved_Access is access UInt_7
    with Storage_Size => 8;
end Custom_Types;
```
Listing 11: show_sizes.adb

with Ada.Text_IO;  use Ada.Text_IO;
with System;

with Custom_Types; use Custom_Types;

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);  
  Put_Line ("UInt_7 Reserved Access'Object_Size:
  & UInt_7 Reserved Access'Object_Size'Image);  
  Put_Line ("RAV1'Size:
  & RAV1'Size'Image);
  New_Line;
  Put_Line ("Allocating RAV1...");
  RAV1 := new UInt_7;
  Put_Line ("Allocating RAV2...");
  RAV2 := new UInt_7;
  New_Line;
end Show_Sizes;

Code block metadata

Sizes
MD5: 6ac085d8467a61ba4f9cd138c024442d

Runtime output

UInt_7_Reserved_Access'Storage_Size:  8
UInt_7_Reserved_Access'Storage_Size (bits):  64
UInt_7_Reserved_Access'Size:  64
UInt_7_Reserved_Access'Object_Size:  64
RAV1'Size:  64

Allocating RAV1...
Allocating RAV2...

raised STORAGE_ERROR : s-poosiz.adb:108 explicit raise

Depending on your target architecture, you may see this output:

UInt_7_Reserved_Access'Storage_Size:  8
UInt_7_Reserved_Access'Storage_Size (bits):  64

(continues on next page)
In this case, we’re reserving 8 storage elements in the declaration of \texttt{UInt\_7\_Reserved\_Access}.

\begin{verbatim}
type UInt_7Reserved_Access is access UInt_7
  with Storage_Size => 8;
end Custom_Types;
\end{verbatim}

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 \texttt{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.

### 2.2.2 Alignment

For many algorithms, it’s important to ensure that we’re using the appropriate alignment. This can be done by using the \texttt{Alignment} attribute and the \texttt{Alignment} aspect. Let’s look at this example:

\begin{verbatim}
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;
\end{verbatim}

\begin{verbatim}
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: ");
\end{verbatim}

(continues on next page)
& 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: 
& Aligned(V'Alignment'Image);
Put_Line
("Aligned(V'Size: 
& Aligned(V'Size'Image);
New_Line;
end Show_Alignment;

Code block metadata
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.Alignment
MD5: a2fea340559193c293ccaee226de2558

Runtime output
UInt_7'Alignment: 1
UInt_7'Size: 7
UInt_7'Object_Size: 8
V'Alignment: 1
V'Size: 8
Aligned(UInt_7'Alignment: 4
Aligned(UInt_7'Size: 7
Aligned(UInt_7'Object_Size: 32
Aligned(V'Alignment: 4
Aligned(V'Size: 32

Depending on your target architecture, you may see this output:

UInt_7'Alignment: 1
UInt_7'Size: 7
UInt_7'Object_Size: 8
V'Alignment: 1
V'Size: 8

(continues on next page)
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:

```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;
```

2.2. Data Representation
2.2.3 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.
  - 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 15: 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 16: int_array_processing.adb

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

package body Int_Array_Processing is

(continues on next page)```
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
    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:
end Main;
-- same array, but not sharing any storage

Process (A, B);
-- Out-of-place processing:
-- two different arrays

dend 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
 appropriated 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...

==== PROCESS ====
Info: X and Y don't have the same storage.
Info: X and Y don't overlap.
Out-of-place processing...

==== PROCESS ====
Info: X and Y don't have the same storage.
Info: X and Y don't overlap.
Out-of-place processing...

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.

### 2.2.4 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 \(\text{UInt}_7'\text{Size}\) was 7 bits, while \(\text{UInt}_7'\text{Object}\_\text{Size}\) was 8 bits. The most extreme case is the one for the \(\text{Boolean}\) type: in this case, \(\text{Boolean}'\text{Size}\) is 1 bit, while \(\text{Boolean}'\text{Object}\_\text{Size}\) 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 \(\text{Boolean}\) 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 \(\text{Boolean}\) components. For example:

```adalah
package Flag_Definitions is
  type Flags is array (Positive range <>) of Boolean;
end Flag_Definitions;
```

```adalah
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;
```

---

**Code block metadata**

- **Project:** Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
- **MD5:** 6fd7a913e3c6717e846c2e822c1cbad7

**Build output**

```
show_flags.adb:5:04: warning: variable "Flags_1" is read but never assigned [-wgnatwv]
```
Runtime output

- Boolean'Size: 1
- Boolean'Object_Size: 8
- Flags_1'Size: 64
- Flags_1'Component_Size: 8

Depending on your target architecture, you may see this output:

- Boolean'Size: 1
- Boolean'Object_Size: 8
- Flags_1'Size: 64
- Flags_1'Component_Size: 8

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 20: 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 21: 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;
```

---

94 Chapter 2. Types and Representation
Depending on your target architecture, you may see this output:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean'Size:</td>
<td>1</td>
</tr>
<tr>
<td>Boolean'Object_Size:</td>
<td>8</td>
</tr>
<tr>
<td>Flags_1'Size:</td>
<td>64</td>
</tr>
<tr>
<td>Flags_1'Component_Size:</td>
<td>8</td>
</tr>
<tr>
<td>Flags_2'Size:</td>
<td>8</td>
</tr>
<tr>
<td>Flags_2'Component_Size:</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

```ada
package Flag_Definitions is

  type Flags is
    array (Positive range <>) of Boolean;

  type Packed_Flags is
    array (Positive range <>) of Boolean
    with Pack;

  constant Default_Flags : Flags :=
    (True, True, False, True,
     False, False, True, True);

end Flag_Definitions;
```

---

2.2. Data Representation 95
Listing 23: show_flag_conversion.adb

```ada
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;
```

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 106).
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.

### 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* 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 24: 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:

<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>
</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>
</tr>
<tr>
<td>component</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;
    B at 4 range 0 .. 31;
  end record;

end P;
```

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:

```
for Record_Type use record
  Component_Name at Start_Position
```
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](http://www.ada-auth.org/standards/22rm/html/RM-13-5-1.html)

### 2.3.1 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:

```ada
package P is

    type R is record
        A : Integer;
        B : Integer;
    end record;

end P;
```

---

First of all, we see that the size of the R type is 64 bits, which can be explained by those
Advanced Journey With Ada: A Flight In Progress

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.

In the Ada Reference Manual

• 13.5.2 Storage Place Attributes

2.3.2 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:

This is the code that implements that:

Listing 28: 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 29: 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

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 30: p.ads

```ada
package P is
  type R is record
    A : Integer;
    B : Integer;
  end record
  with Object_Size => 72;
  for R use record
    A at 0 range 0 .. 31;
    B at 5 range 0 .. 31;
  end record;
end P;
```

Listing 31: 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;
```

If the code compiles, R'Size and R'Object_Size should now have the same value.
### 2.3.3 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;
```

```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;
```

```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**

2.3. Record Representation and storage clauses

---

2.3. Advanced Journey With Ada: A Flight In Progress
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.

### 2.3.4 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 component</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>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td></td>
<td></td>
<td></td>
<td></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;
```

---

**Runtime output**

| R'Size: | 64 |
| R'Object_Size: | 64 |
| R_New'Size: | 72 |
| R_New'Object_Size: | 72 |

---

**Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Derived_Rep_Clauses_Empty_Byte**

MD5: 3a1e0837f8bd8250f20fc7b274b869d5
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
  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;
```

```
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;
```

2.3. Record Representation and storage clauses

```
Listing 36: show_simple_reg.adb

Listing 37: show_simple_reg.adb
```

Runtime output

Simple_Reg'Size: 8
Simple_Reg'Object_Size: 8

R.S: READY
R.V1: 4
As we can see in the example, to retrieve the current status of the register, we just have to write \texttt{R.S}. To update the \texttt{V1} field of the register with the value 4, we just have to write \texttt{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.

### 2.4 Changing Data Representation

\textbf{Note: } This section was originally written by Robert Dewar and published as G\textsuperscript{em} \#27: Changing Data Representation\textsuperscript{30} and G\textsuperscript{em} \#28\textsuperscript{31}.

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 38: 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;
```

\textbf{Code block metadata}


MD5: cbd7f5547c5b045b8853ac21d03aa41f8

\textbf{Build output}

```
communication.ads:12:11: warning: component clause forces biased representation, ...
```

which lays out the fields of a record, and in the case of \texttt{Val}, forces a biased representation in which all zero bits represents 100. Another example is:

\textsuperscript{30} https://www.adacore.com/gems/gem-27

\textsuperscript{31} 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:

```
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 speci-
fications, 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 spec-
ified format. Ada provides a very convenient method for doing this, as described in RM 13.6
"Change of Representation".32

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:

```
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

**Project:** Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Array_Rep
**MD5:** 7eb17fc2cd415acb7c53a363fa336807

---

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 42: using_array_for_io.adb

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 auto-
matic.

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:
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 13.1^33 (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:

```ada
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;
```

2.4.1 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.

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 gobbledygook 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 13.134 (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:

---

Listing 45: my_ints.ads

```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;
```

Listing 46: my_ints.adb

```ada
package body My_Ints is

  function Odd (Arg : My_Int_1) return Boolean is
    (True);
  -- Dummy implementation!

end My_Ints;
```

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:

Listing 47: using_my_int.adb

```ada
with My_Ints; use My_Ints;

procedure Using_My_Int is
  Var : My_Int_2;
begin
  if Odd (Var) then
    -- ^ Calling Odd function
    -- for My_Int_2 type.
    null;
  end if;
end Using_My_Int;
```

Build output
The compiler translates this as:

```
with My_Ints; use My_Ints;

procedure Using_My_Int is
  Var : My_Int_2;
begin
  if Odd (My_Int_1 (Var)) then
    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.
### 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 49: create_test_file.ads

```ada
procedure Create_Test_File (File_Name : String);
```

Listing 50: 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 51: states.ads

```ada
with Ada.Sequential_IO;

package States is
    type State is (Off, On, Waiting);
    with Size => Integer'Size;
```

(continues on next page)
for State use (Off => 1, On => 2, Waiting => 4);

package State_Sequencial_IO is new Ada.Sequential_IO (State);

procedure Read_Display_States (File_Name : String);
end States;

Listing 52: states.adb

with Ada.Text_IO; use Ada.Text_IO;

package body States is

procedure Read_Display_States (File_Name : String)
is
  use State_Sequencial_IO;
  F : State_Sequencial_IO.File_Type;
  S : State;

procedure Display_State (S : State) is
begin
  -- Before displaying the value,
  -- check whether it's valid or not.
  if S'Valid then
    Put_Line (S'Image);
  else
    Put_Line ("Invalid value detected!");
  end if;
end Display_State;

begin
  Open (F, In_File, File_Name);
  while not End_Of_File (F) loop
    Read (F, S);
    Display_State (S);
  end loop;
  Close (F);
end Read_Display_States;
end States;

Listing 53: show_states_from_file.adb

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);
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 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:

```ada
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

---

2.6 Unchecked Union

We’ve introduced variant records back in the Introduction to Ada course[^1]. 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:

```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;

  procedure Display_State_Value
    (V : State_Or_Integer);
end States;
```

```ada
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;
```

[^1]: https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_records.html#intro-ada-variant-records

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** ex-

[^1]: https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_records.html#intro-ada-variant-records
ception 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 56: show_variant_rec_error.adb

```ada
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:

Listing 57: show_variant_rec_error.adb

```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;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_Or_Integer
MD5: 985a84faccc3d590ac767e914bea0c1d

Build output

show_variant_rec_error.adb:4:04: warning: variable "V" is never read and never assigned [-gnatwv]
show_variant_rec_error.adb:6:05: warning: component not present in subtype of "State_Or_Integer" defined at line 4 [enabled by default]
show_variant_rec_error.adb:6:05: warning: Constraint_Error will be raised at run time [enabled by default]

Runtime output

raised CONSTRAINT_ERROR : show_variant_rec_error.adb:6 discriminant check failed

2.6. Unchecked Union
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.

Listing 58: states.ads

```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)
    with Unchecked_Union;

procedure Display_State_Value
  (V : Unchecked_State_Or_Integer);

end States;
```

Listing 59: states.adb

```ada
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:
Listing 60: show_unchecked_union.adb

```ada
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;
```

Code block metadata

Unchecked_State_Or_Integer
MD5: 331cc1ab6709ab7e0062d64c55a75a6c

Runtime output

State: ON
Value: 2

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 61: 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

Unchecked_State_Or_Integer
MD5: bb472e91c5e7b7e0062d6246dbc5226a0

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:

2.6. Unchecked Union
Listing 62: 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

Unchecked_State_Or_Integer
MD5: 74ac11a3effdaff395f9fface295a86da

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]

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 63: 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;
```

Code block metadata
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:

```
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;
end States;
```

```
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

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[38]

2.7.1 Volatile

A volatile[39] 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[40], 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:

Listing 66: 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

- MD5: aa1e276e64e69813bfc3e3ef39f3dd47

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:

---

[40] https://en.wikipedia.org/wiki/Memory-mapped_I/O
Listing 67: 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 68: 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_Type
- **MD5**: 0d31156d47b2edc7f94debd016c8bb87

**Runtime output**

```
Val:  9.99000000000000E+05
```

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:

Listing 69: show_volatile_array_components.adb

```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): " & Long_Float'Image (Arr (1)));
  Put_Line ("Arr (2): " & Long_Float'Image (Arr (2)));
end Show_Volatile_Array_Components;
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

17 & Long_Float'Image (Arr (2)));
18 end Show_Volatile_Array_Components;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Volatile_Array_Components
MD5: 05b3ee20f08c5a85f5872727a61c148d

Runtime output

Arr (1): 9.99000000000000E+05
Arr (2): 4.99500000000000E+06

Note that it's possible to use the Volatile aspect for the array declaration as well:

Listing 70: shared_var_types.ads

package Shared_Var_Types is

private

Arr : array (1 .. 2) of Long_Float
with Volatile;

end Shared_Var_Types;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_Control.Volatile_Array
MD5: c9b7b9f94f1fac295753c7e7b9426fb2

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.

2.7.2 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:

Listing 71: shared_var_types.ads

package Shared_Var_Types is

I : Integer with Independent;
Similarly, we can use this aspect when declaring types:

Listing 72: shared_var_types.ads

```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 `Flags` record type. Let's now derive the `Flags` type and use a representation clause for the derived type:

Listing 73: shared_var_types-representation.ads

```ada
package Shared_Var_Types.Representation is

  type Rep_Flags is new Flags;

  for Rep_Flags use record
    F1 at 0 range 0 .. 0;
    F2 at 0 range 1 .. 1;
    -- ^ ERROR: start position of
    -- F2 is wrong!
    -- ^ ERROR: F1 and F2 share the
    -- same storage unit!
  end record;

end Shared_Var_Types.Representation;
```
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:

```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 1 range 0 .. System.Storage_Unit - 1;
  end record;

end Shared_Var_Types.Representation;
```

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 ...:

```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;
```

(continues on next page)
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:

```ada
package Shared_Var_Types is
  Flags : array (1 .. 8) of Boolean
    with Independent_Components;
end Shared_Var_Types;
```

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:

```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;
```

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
```

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
Advanced Journey With Ada: A Flight In Progress

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 78: shared_var_types.ads

```adada
package Shared_Var_Types is
  type Flags is
    array (Positive range <>) of Boolean
    with Independent_Components, Pack;
end Shared_Var_Types;
```

Listing 79: show_flags_size.adb

```adada
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;
```

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.
2.7.3 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:

Listing 80: shared_var_types.ads

```
with System;

package Shared_Var_Types is
private
R : Integer
    with Atomic,
    Address =>
        System'To_Address (16#FFFF00A0#);
end Shared_Var_Types;
```

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 133) and the GNAT-specific System'To_Address attribute (page 134).
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;
```

This example shows the declaration of 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
  Arr: array (1..2) of Integer
    with Atomic_Components;
end Shared_Var_Types;
```

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:

```
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;
```
(continues on next page)
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:

```
Listing 84: shared_var_types.ads

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;
```

### 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 481) in another chapter. Also, later on in that chapter, we discuss the relation between access types and addresses (page 595).) 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 135) and address arithmetic (page 137) (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

---

2.8.1 Address attribute

The **Address** attribute allows us to get the address of an object. For example:

```ada
with System; use System;

procedure Use_Address is
  I : aliased Integer := 5;
  A : Address;
begin
  A := I'Address;
end Use_Address;
```

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:

```ada
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\(^{42}\)

2.8.2 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:

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

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:

```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;
```

---

2.8. Addresses

---

Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

begin
    S := State_2;
    Put_Line ("I = " & Integer'Image (I));
end Simple_Overlay;

Code block metadata
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'To_Address attribute when using the Address aspect, as we did while talking about the Atomic (page 129) aspect:

Listing 89: shared_var_types.ads

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.Addresses.Show_Access_Address
MD5: 5c2d8e0a9615084c2a15f896c61adaa6

In this case, R will refer to the address in memory that we're specifying (16#FFFF00A0# in this case).

As explained in the GNAT Reference Manual\textsuperscript{44}, the System'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 To_Address function (page 136) function later on.)

In the Ada Reference Manual
- 13.3 Operational and Representation Attributes\textsuperscript{45}
- 13.7 The Package System\textsuperscript{46}

\textsuperscript{44} https://gcc.gnu.org/onlinedocs/gnat_rm/Attribute-To_005fAddress.html
\textsuperscript{46} http://www.ada-auth.org/standards/22rm/html/RM-13-7.html

134 Chapter 2. Types and Representation
2.8.3 Address comparison

We can compare addresses using the common comparison operators. For example:

Listing 90: 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 : ");
    Put_Line (System.Address_Image (I'Address));
    Put_Line ("J'Address : ");
    Put_Line (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;
```

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

---

2.8.4 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:

Listing 91: 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
   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));
end Show_Address;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_Arith_Ada
MD5: 69e053886fb8e0571d6c94247dc9f30f

Runtime output

A1 : 00007FFFB2EEB1AC
IA : 140736195375532
A2 : 00007FFFB2E81AC

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\(^{49}\)

\(^{49}\) http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html
2.8.5 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:

```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;
```

Listing 92: show_address.adb
In this example, we initialize the address A by retrieving the address of the first component of the array Arr. (Note that we could have written Arr(1)'Address instead of Arr'Address. In any case, the language guarantees that Arr'Address gives us the address of the first component, i.e. Arr'Address = Arr(1)'Address.)

Then, in the loop, we declare an overlay Curr using the current value of the A address. We can then operate on this overlay — here, we assign I to Curr. Finally, in the loop, we increment address A and make it point to the next component in the Arr array — to do so, we calculate the size of an Integer component in storage units. (For details on storage units, see the section on storage size attribute (page 84).)

In other languages
The code example above corresponds (more or less) to the following C code:

```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++)
```
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

{    
    printf("arr[%d]: %d\n", i, arr[i]);
}

return 0;
}

**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_Arith_C

MD5: 7aa709a4d7ed6ce2346dbabc853e28c0

**Runtime output**

curr address: 0x7fffc38f0cf4
curr address: 0x7fffc38f0cf8
curr address: 0x7fffc38f0cfc
curr address: 0x7fffc38f0d00
curr address: 0x7fffc38f0d04
curr address: 0x7fffc38f0d08
curr address: 0x7fffc38f0d0c
curr address: 0x7fffc38f0d10
curr address: 0x7fffc38f0d14
curr address: 0x7fffc38f0d18
arr[0]: 0
arr[1]: 1
arr[2]: 2
arr[3]: 3
arr[4]: 4
arr[5]: 5
arr[6]: 6
arr[7]: 7
arr[8]: 8
arr[9]: 9

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
- 13.7.1 The Package System.Storage_Elements

---


2.8. Addresses
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:

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

procedure Show_Enumeration_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 Show_Enumeration_Image;
```

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

procedure Show_Enumeration_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) return String is

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";
    end case;
end Months_Image;
```

This is similar to having this code:

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

procedure Show_Enumeration_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) return String is

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";
    end case;
end Months_Image;
```
when October => return "OCTOBER";
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 96: show_discard_names.adb

procedure ShowDiscardNames is
  pragma Warnings (Off, "is not referenced");

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

  M : constant Months := January;
begin
  null;
end ShowDiscardNames;

In this example, the compiler attempts to not store strings associated with the Months type duration compilation.

Note that the Discard_NAMES aspect is available for enumerations, exceptions, and tagged types.

In the GNAT toolchain

2.9. Discarding names
If we add this statement to the Show_Discard_Names procedure above:

```ada
Put_Line ("Month: " & Months'Image (M));
```

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[^52]

3.1 Default Initialization

As mentioned in the *Introduction to Ada* course, record components can have default initial values. Also, we’ve seen that other kinds of types can have *default values* (page 63).

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 1: show_default_initialization.ads

```ada
package Show_Default_Initiation is

  function Init return Integer is (42);

  type Rec is record
    A : Integer := Init;
  end record;

end Show_Default_Initiation;
```

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

---

53 https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-default-values
### 3.1.1 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 2: 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;
```

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.)

### 3.1.2 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 3: 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 4: 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 5: 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

- **Project:** Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.
- **MD5:** e3ab92ea9b2a99815cea8c2ea11cbbfb

Runtime output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>1</td>
</tr>
<tr>
<td>B:</td>
<td>2</td>
</tr>
</tbody>
</table>

When running this code example, you might see this:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>1</td>
</tr>
<tr>
<td>B:</td>
<td>2</td>
</tr>
</tbody>
</table>

However, the compiler is allowed to rearrange the operations, so this output is possible as well:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B:</td>
<td>2</td>
</tr>
<tr>
<td>A:</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

3.1. Default Initialization
3.1.3 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:

```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 Types\(^{55}\)

---

3.1.4 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{56}.

Consider the following type declaration:

```
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;
end record;
procedure Do_Something;
end Type_Defaults;
```

If we want to say, "make Count equal 100, but initialize Color and Is_Gnarly to their defaults", we can do this:

```
package body Type_Defaults is
  Object_100 : constant T :=
    (Color => <>,
     Is_Gnarly => <>,
     Count = 100);
procedure Do_Something is null;
end Type_Defaults;
```

Historically
Prior to Ada 2005, the following style was common:

\textsuperscript{56} https://www.adacore.com/gems/ada-gem-12
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;
```

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
  begin
      Object_100.Count := 100;
  end Do_Something;

end Type_Defaults;
```
Therefore, you should be careful and think twice before using others.

### 3.1.5 Advanced Usages

In addition to expressions such as subprogram calls, we can use per-object expressions (page 163) for the default value of a record component. (We discuss this topic later on in more details.)

For example:

```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
  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.

### 3.2 Mutually dependent types

In this section, we discuss how to use incomplete types (page 34) to declare mutually dependent types. Let's start with this example:

```ada
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:

```ada
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:

```ada
package Mutually_Dependent is

    type T1;
    type T1_Access is access T1;

    type T2;

end Mutually_Dependent;
```
Later on, we'll see that these code examples can be written using anonymous access types (page 624).

In the Ada Reference Manual
• 3.10.1 Incomplete Type Declarations

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:

Listing 16: null_recs.ads

    package Null_Recs is
    
    type Null_Record is record
        null;
    end record;

    end Null_Recs;

Code block metadata
MD5: 3c82da822710342354134fa71a03452a

Note that the syntax can be simplified to is null record, which is much more common than the previous form:

Listing 17: null_recs.ads

    package Null_Recs is
    
    type Null_Record is null record;

    end Null_Recs;

(continues on next page)
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:

```
package Null_Recs is
  type Null_Record is null record;
  function "+" (A, B : Null_Record) return Null_Record;
end Null_Recs;
```

```
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;
```

```
with Null_Recs; use Null_Recs;
procedure Show_Null_Rec is
  A, B : Null_Record;
begins
  B := A + A;
  A := A + B;
end Show_Null_Rec;
```

In the Ada Reference Manual

- 4.3.1 Record Aggregates

---

3.3.1 Simple Prototyping

A null record doesn't provide much functionality on itself, 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 21: devices.ads
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 22: show_device.adb
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: 7d2fce20ac33607f7081381b307a564a

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 392).

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.

### 3.3.2 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;
```

(Listing 23: devices.ads)
Listing 24: devices.adb

```ada
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;
```

Listing 25: show_device.adb

```ada
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;
```

3.3. Null records
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.

### 3.3.3 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:

Listing 26: many_devices.ads

```ada
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;
```

Listing 27: show-derived_device.adb

```ada
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.

### 3.3.4 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:

Listing 28: devices.ads

```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;
```

Listing 29: devices.adb

```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 30: 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
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.

### 3.3.5 Tagged null records

A null record may be tagged, as we can see in this example:

```
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:

```
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);

(continues on next page)
```
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 consider using interfaces instead of null records. We'll discuss this topic later on in the course.

### 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:

Listing 33: 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);
    end record;
end Rec_Per_Object_Expressions;
```
In this example, we see the Stack record type with a discriminant S. In the declaration of the Arr component of the 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 601):

```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: 8b404688be1e103773c28a6977785836
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 35: 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
  -- ^^^^^^^^^
  -- 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

---

3.4.1 Default value

We can also use per-object expressions to calculate the default value of a record component:

```
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 164) 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:

```
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;
end Rec_Per_Object_Expressions;
```

(continues on next page)
### 3.4.2 Restrictions

There are some important restrictions on per-object constraints:

- Per-object range constraints such as `1 .. T'Size` are not allowed.

  For example, the following code example doesn't compile:

```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'Size);
  -- ^^^^^^^
  -- ERROR: per-object range constraint
  -- using the Size attribute
  -- is illegal.
end record;

end Rec_Per_Object_Expressions;
```

---

In this example, we can identify multiple per-object expressions that use a computation: \( S \times 9 / 10 \), \((S + 1) / 2\), and \( S - 1 \).
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 \((1 \ldots S)\) — as we've seen in a previous example. However, writing \((1 \ldots S - 1)\) would be illegal.
- For example, the following adaptation to the previous code example doesn't compile:

```
Listing 39: rec_per_object_expressions.ads
```

```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);
    Top : Integer := -1;
  end record;

end Rec_Per_Object_Expressions;
```

2. We can only use access attributes (T'Access and T'Unchecked_Access) in per-object constraints.
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:

```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:
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 3: show_named_container_aggregate.adb
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
Indexed_Named_V : constant Float_Vec.Vector :=
  [1 => 1.0,
   2 => 2.0,
   3 => 3.0];

-- Named container aggregate
-- using a key
Keyed_Named_V : constant
  Float_Hashed_Maps.Map :=
  ["Key_1" => 1.0,
   "Key_2" => 2.0,
   "Key_3" => 3.0];

pragma Unreferenced (Indexed_Named_V,
                      Keyed_Named_V);

begin
  null;
end Show_Named_Container_Aggregate;

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\(^6\)

\(^6\) http://www.ada-auth.org/standards/22rm/html/RM-4-3-5.html
4.2 Record aggregates

We’ve already seen record aggregates in the Introduction to Ada\textsuperscript{61} 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 4: points.ads

```ada
package Points is

  type Point_3D is record
    X, Y, Z : Integer;
  end record;

  procedure Display (P : Point_3D);

end Points;
```

Listing 5: 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 => ",
               & Integer'Image (P.X),
               ", ");
    Put_Line (" Y => ",
               & Integer'Image (P.Y),
               ", ");
    Put_Line (" Z => ",
               & Integer'Image (P.Z),
               ")");
  end Display;

end Points;
```

We can use positional or named record aggregates when assigning to an object P of Point_3D type:

Listing 6: show_record_aggregates.adb

```ada
with Points; use Points;

procedure Show_Record_Aggregates is
  P : Point_3D;
begin
  -- Positional component association
  P := (0, 1, 2);
```

\textsuperscript{61} https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-aggregates
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 7: show_record_aggregates.adb

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: 493a2a87b4b28d8b0882ad73acf84710

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.

4.2. Record aggregates
In addition, we can choose multiple components at once and assign the same value to them. For that, we use the | syntax:

```
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;
```

Here, we assign 5 to both X and Y.

**In the Ada Reference Manual**

- 4.3.1 Record Aggregates

### 4.2.1 <>

We can use the <> 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:

```
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);
end Show_Record_Aggregates;
```

---

62 [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)
15 -- Specifying Y and Z components.
16 P := (X => <>,
17     Y => 10,
18     Z => 20);
19 Display (P);
20 end Show_Record_Aggregates;

Here, as the components of Point_3D don't have a default value, those components that have <> are not initialized:

- when we write (X => 42, Y => <<>, Z => <>), only X is initialized;
- when we write (X => <<>, Y => 10, Z => 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 => 42, Y => <<>, Z => <>)) keeps the value of Y and Z intact, while (X => <<>, Y => 10, Z => 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:

Listing 10: points.ads

```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;
```

Code block metadata

4.2. Record aggregates
Then, writing <> makes use of those default values we’ve just specified:

```
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 => <>, Z => <>);
  Display (P);
end Show_Record_Aggregates;
```

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:

```
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;
```

(continues on next page)
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

MD5: 508d7f5e7d02da1677485f7d588847f6

Then, writing <> makes use of the default value of the Point_Value type:

Listing 14: show_record_aggregates.adb

   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;

Code block metadata

MD5: 895799077af4a295c250480c32954a2c

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 <>.

4.2. Record aggregates
### 4.2.2 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:

Listing 15: show_record_aggregates.adb

```ada
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;
```

Runtime output

(X => 42,
 Y => 100,
 Z => 100)

(X => 256,
 Y => 256,
 Z => 256)

When we write `P := (X => 42, others => 100)`, we're assigning 42 to X and 100 to all other components (Y and Z in this case). Also, when we write `P := (others => 256)`, all components have the same value (256).

Note that writing a specific value in `others` — such as `(others => 256)` — only works when all components have the same type. In this example, all components of `Point_3D` have the same type: `Integer`. If we had components with different types in the components selected by `others`, say `Integer` and `Float`, then `(others => 256)` would trigger a compilation error. For example, consider this package:

Listing 16: custom_records.ads

```ada
package Custom_Records is
  type Integer_Float is record
    A, B : Integer := 0;
    Y, Z : Float := 0.0;
  end record;
end Custom_Records;
```

Code block metadata

MD5: 3146363eb36ab4485ce7755794fb78bbc
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 \( Z \):

Listing 17: show_record_aggregates_others.adb

```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;
```

We can fix this compilation error by making sure that \( \text{others} \) only refers to components of the same type:

Listing 18: show_record_aggregates_others.adb

```ada
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;
```

In any case, writing \( \text{others} \Rightarrow <> \) is always accepted by the compiler because it simply selects the default value of each component, so the type of those values is unambiguous:

4.2. Record aggregates
Listing 19: show_record_aggregates_others.adb

```
with Custom_Records; use Custom_Records;

procedure Show_Record_Aggregates_Others is
  Dummy : Integer_Float;
begin
  Dummy := (others => <>);
end Show_Record_Aggregates_Others;
```

This code compiles because <> uses the appropriate default value of each component.

### 4.2.3 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 20: points.ads

```
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 21: points.adb

```
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)
                  & ")");
```

(continues on next page)
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:

Listing 22: show_rec_aggregate_discriminant.adb

```ada
with Points; use Points;

procedure Show_Rec_Aggregate_Discriminant is
  -- Positional component association
  constant P1 : Point := (Dim_1, 0);
  -- Named component association
  constant P2 : Point := (D => Dim_2, X2 => 3, Y2 => 4);
  -- Positional / named component association
  constant P3 : Point := (Dim_3, X3 => 3, Y3 => 4, Z3 => 5);
begin
  Display (P1);
  Display (P2);
end Show_Rec_Aggregate_Discriminant;
```

4.2. Record aggregates
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.

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\(^63\).

One interesting feature of Ada are the full coverage rules for aggregates. For example, suppose we have a record type:

```
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:

\(^63\) https://www.adacore.com/gems/gem-1
Listed 24: 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;
```

Code block metadata

Full_Coverage_Rules
MD5: 681e665b76265eff4c4d870ec011ba37

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:

Listed 25: 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;
```

Code block metadata

Full_Coverage_Rules
MD5: 5fc5b93748d92932bfc9e0f15c0228b7

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 aggregates (page 182) and case statements, but occasionally useful for record aggregates (page 170)):

Listed 26: show_aggregate_init_others.adb

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

with Persons; use Persons;

procedure Show_Aggregate_Init_Others is
(continues on next page)
```

4.3. Full coverage rules for Aggregates
According to the Ada RM, \texttt{others} here means precisely the same thing as \texttt{Age | Shoe\_Size}. But that's wrong: what \texttt{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 \texttt{others} unless you're pretty sure it should apply to all the cases that haven't been invented yet.

Later on, we'll discuss \textit{full coverage rules for limited types} (page 700).

### 4.4 Array aggregates

We've already discussed array aggregates in the Introduction to Ada\textsuperscript{64} course. Therefore, this section just presents some details about this topic.

**In the Ada Reference Manual**

- 4.3.3 Array Aggregates\textsuperscript{65}

### 4.4.1 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 \textit{record aggregates} (page 170), array aggregates can be positional or named. Consider this package:

```
package Points is
  type Point_3D is array (1 .. 3) of Integer;
  procedure Display (P : Point_3D);
end Points;
```

\textsuperscript{64} [https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-array-type-declaration](https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-array-type-declaration)
\textsuperscript{65} [http://www.ada-auth.org/standards/22rm/html/RM-4-3-3.html](http://www.ada-auth.org/standards/22rm/html/RM-4-3-3.html)
Listing 28: points.adb

```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

We can write positional or named aggregates when assigning to an object P of Point_3D type:

Listing 29: show_array_aggregates.adb

```ada
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;
```

Code block metadata

MD5: 5913ef6f43ea873de4e3f0760265de4b

Runtime output

4.4. Array aggregates
In this example, we assign a positional array aggregate ([1, 2, 3]) 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:

```
Listing 30: show_array_aggregates.adb

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];
  Display (P);
end Show_Array_Aggregates;
```

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).

### Historically

In the first versions of Ada, we could only write array aggregates using parentheses.

```
Listing 31: show_array_aggregates.adb

pragma Ada_2012;
(continues on next page)
```
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;

Code block metadata
MD5: 3d9f1fda006f1d566ae2743240568879

Runtime output
(X => 0, Y => 1, Z => 2) 
(X => 3, Y => 4, Z => 5)

This syntax is considered obsolescent since Ada 2022: brackets ([1, 2, 3]) should be used instead.

4.4.2 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 32: integer_arrays.ads

package Integer_Arrays is
  type Integer_Array is
    array (Positive range <>) of Integer;
  procedure Display (A : Integer_Array);
end Integer_Arrays;
Listing 33: 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;
```

Code block metadata

MD5: 412ebe9de1dfb9157f5379d31162554d

We can initialize an object N of Integer_Array type with a null array aggregate:

Listing 34: show_array_aggregates.adb

```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;
```

Code block metadata

MD5: 8cdb9a004ea16f716bf2e2ad5a65358e

Runtime output

```
Length = 0 ()
```
In this example, when we call the Display procedure, we confirm that N doesn't have any components.

### 4.4.3 |, <>, others

We've seen the following syntactic elements when we were discussing record aggregates (page 170): |, <> 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 := [others => 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
  -- have a value of five.
  P := [1 => <>,
        others => 5];
  Display (P);
end Show_Array_Aggregates;
```

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 197).

4.4. Array aggregates
4.4.4 ..

We can also use the range syntax (..) with array aggregates:

Listing 36: show_array_aggregates.adb

```
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
    -- have a value of five.
    P := [1    => <>,
          2 .. 3 => 5];
    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.
4.4.5 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:

Listing 37: show_array_aggregates.adb

```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;
```

Code block metadata

MD5: 40d3a65f7fc060278ae07769

Build output

show_array_aggregates.adb:8:09: warning: too few elements for type "Point_3D"
  defined at points.ads:3 [enabled by default]
show_array_aggregates.adb:8:09: warning: expected 3 elements; found 1 element
  [enabled by default]
show_array_aggregates.adb:8:09: warning: Constraint_Error will be raised at run time [enabled by default]

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 38: 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

4.4. Array aggregates
However, **others** can only be used when the range is known — compilation fails otherwise:

```
pragma Ada_2022;
with Integer_Arrays; use Integer_Arrays;
procedure Show_Array_Aggregates is
  N1 : Integer_Array := [others => 0];
  -- ERROR: range is unknown
  N2 : Integer_Array (1 .. 3) := [others => 0];
  -- OK: range is known
begin
  Display (N1);
  Display (N2);
end Show_Array_Aggregates;
```

Of course, we could fix the declaration of `N1` by specifying a range — e.g. `N1 : Integer_Array (1 .. 10) := [others => 0];`.

### 4.4.6 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:

```
-- All components have a value of zero
P := [for I in 1 .. 3 => 0];
```

Let's see a complete example:
Listing 40: 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 := [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:

4.4. Array aggregates 191
Listing 41: show_array_aggregates.adb

```ada
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;
```

**Code block metadata**

MD5: b8c1878c1fa516005d1861f1a37c4fb0

**Runtime output**

(X => 2,
 Y => 4,
 Z => 6)
(X => 3,
 Y => 5,
 Z => 7)

In this example, we use iterators in different ways:

1. We write \([\text{for E of P => E } \times \text{ 2}])\) to double the value of each component.
2. We write \([\text{for E of P => E } + \text{ 1}])\) to increase the value of each component by one.

Of course, we could write more complex operations on \(E\) in the iterators.

### 4.4.7 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 42: matrices.ads

```ada
package Matrices is
  type Matrix is array (Positive range <>,
                        Positive range <>)
                    of Integer;
  procedure Display (M : Matrix);
end Matrices;
```
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;
   end Display_Row;

   begin
       Put_Line ("Length (1) = "
                 & M'Length (1)'Image);
       Put_Line ("Length (2) = "
                 & 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;
   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 44: show_array_aggregates.adb

```ada
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: 78e1fad3b90c4f44f0f9d45f299e5ae10

Runtime output

Length (1) = 2
Length (2) = 3
(  
  (   1 => 0,   
       2 => 1,   
       3 => 2   
  ),
  (   1 => 3,   
       2 => 4,   
       3 => 5   
  )
)
Length (1) = 2
Length (2) = 3
(  
  (   1 => 3,   
       2 => 4,   
       3 => 5   
  ),
  (   1 => 6,   
       2 => 7,   
       3 => 8   
  )
)
The first aggregate we use in this example is $[[0, 1, 2], [3, 4, 5]]$. 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:

```ada
package String_Lists is
  type String_List is array (Positive range <>,
                             Positive range <>) of Character;
  procedure Display (SL : String_List);
end String_Lists;
```

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
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 Display_Row;
```

(continues on next page)
```ada
   end if;
   end loop;
   Put_Line ("\n");
   end Display;
end String_Lists;
```

### Code block metadata

**Project:** Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.String_Aggregates  
**MD5:** 8bd7e593cab823218a39c07d85f40c22

Then, when assigning to an object `SL` of String_List type, we can use strings in the aggregates:

Listing 47: show_array_aggregates.adb

```ada
pragma Ada_2022;
with String_Lists; use String_Lists;
procedure Show_Array_Aggregates is
   SL : String_List (1 .. 2, 1 .. 3);
begin
-- Positional component association
   SL := ["ABC", "DEF"];
   Display (SL);

-- Named component associations
   SL := [{1 => 'A', 2 => 'B', 3 => 'C'}];
   Display (SL);
   SL := [{1 => 'X', 2 => 'Y', 3 => 'Z'}, [others => ' ']];
   Display (SL);
end Show_Array_Aggregates;
```

### Code block metadata

**Project:** Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.String_Aggregates  
**MD5:** 82e376269e3be935d5cbd662620f26ec7

### Runtime output

Length (1) = 2  
Length (2) = 3  
(ABC),
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.)

4.4.8 <> 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 48: points.ads

```ada
package Points is
    subtype Point_Value is Integer;

    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 49: points.adb

```ada
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 => ",)
```

(continues on next page)
& 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;
end Display;
end Points;

-- Array components are
-- uninitialized.
PA := [ (X => 3,
  Y => 4,
  Z => 5),
(X => 6,
  Y => 7,
  Z => 8) ];
Display (PA);

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);

PA := [ 1 => <>,
  2 => <> ];
Display (PA);
end Show_Record_Aggregates;

Then, let's use <> for the array components:
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:

Listing 51: 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 52: 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);
```

(continues on next page)
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 64)) in the declaration of the array type:

```
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 => <>];
```
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 63)) and Default_Component_Value aspects at the same time. In this case, the value specified in the Default_Component_Value aspect has higher priority:

Listing 55: integer_arrays.ads

```ada
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;
```

Listing 56: show_array_aggregates.adb

```ada
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 => <>];
end Show_Array_Aggregates;
```
Display (N);
end Show_Array_Aggregates;

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.

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.

In the Ada Reference Manual
- 4.3.2 Extension Aggregates

4.5.1 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.

http://www.ada-auth.org/standards/22rm/html/RM-4-3-2.html
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.

### 4.5.2 Example: Points

To present a more concrete example, let's start with a package that defines one, two and three-dimensional point types:

**Listing 57: 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 58: 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
    Put_Line ("(X => " & P.X'Image & " , Y => " & P.Y'Image & ", Z => " & P.Z'Image & ")");
  end Display;
end Points;
```

(continues on next page)
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 59: 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;"
  P_2D.Y := 0.7;
  Display (P_2D);
end Show_Points;

Code block metadata

MD5: 68ae6fa8e6f779aebea97085bd75e082

Runtime output

(X => 5.00000E-01)
(Y => 5.00000E-01, Y => 7.00000E-01)

In this example, we're initializing P_2D using the information stored in P_1D. By writing Point_1D (P_2D) on the left side of the assignment, we specify that we want to limit our focus on the Point_1D view of the P_2D object. Then, we assign P_1D to the Point_1D view of the P_2D object. This assignment initializes the X component of the P_2D object. The Point_2D 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 Point_1D type only has the X component.) Finally, in the next line, we initialize the Y component with 0.7.
4.5.3 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 `with` keyword — for example: `(Obj1 with Y => 0.5)`. This allows us to assign to an object with information from another object 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;
```

When we write `P_2D := (P_1D with Y => 0.7)`, we're initializing `P_2D` using:

- the information from the `P_1D` object — of `Point_1D` type, which is an ancestor of the `Point_2D` type —, and
- the information from the record component association list for the remaining components of the `Point_2D` type. (In this case, the only remaining component of the `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 `Point_2D'(...)` to indicate that we expect the `Point_2D` type for the extension aggregate.

```
-- Explicitly state that the type of the
-- extension aggregate is Point_2D:
P_2D := Point_2D'(P_1D with Y => 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 with 0.7);
```

4.5. Extension Aggregates
4.5.4 More extension aggregates

We can use extension aggregates for descendants of the Point_2D type as well. For example, let’s extend our previous code example by declaring an object of Point_3D type (called P_3D) and use extension aggregates in assignments to this object:

Listing 61: show_points.adb

```ada
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);
  Display (P_1D);
  P_2D := (P_1D with Y => 0.7);
  Display (P_2D);
  P_3D := (P_2D with Z => 0.3);
  Display (P_3D);
  P_3D := (P_1D with Y | Z => 0.1);
  Display (P_3D);
end Show_Points;
```

Code block metadata

MD5: 2ec6831557c43f697b7c8e8496962b53

Runtime output

(X => 5.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01, Z => 3.00000E-01)
(X => 5.00000E-01, Y => 1.00000E-01, Z => 1.00000E-01)

In the first assignment to P.3D in the example above, we’re initializing this object with information from P.2D and specifying the value of the Z component. Then, in the next assignment to the P.3D object, we’re using an aggregate with information from P.1D and specifying values for the Y and Z components. (Just as a reminder, we can write Y | Z => 0.1 to assign 0.1 to both Y and Z components.)

4.5.5 with others

Other versions of extension aggregates are possible as well. For example, we can combine keywords and write with others to focus on all remaining components of an extension aggregate.

Listing 62: show_points.adb

```ada
with Points; use Points;

procedure Show_Points is
  P_1D : Point_1D;
P_2D : Point_2D;
end Show_Points;
```

(continues on next page)
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;

---

In this example, the first assignment to P_3D has an aggregate with information from P_1D, while the remaining components — in this case, Y and Z — are just set to 0.6.

Continuing with this example, in the next assignment to P_3D, we're using information from P_2 in the extension aggregate. This covers the Point_2D part of the P_3D object — components X and Y, to be more specific. The Point_3D specific components of 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 => <>.

4.5.6 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.

Listing 63: points-extensions.ads

package Points.Extensions is
  type Point_3D_Ext is new
    Point_3D with null record;
end Points.Extensions;

---

4.5. Extension Aggregates  207
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;

Code block metadata
MD5: 8ec3ddb3a1f2a6e550ac4d622e97124c

Runtime output
(X => 0.00000E+00, Y => 5.00000E-01, Z => 4.00000E-01)

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.

### 4.5.7 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:

```ada
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;
```

Code block metadata
MD5: ae5e88a36c58b1eb495d5ba8752e50e7

Runtime output
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 descendant type (Point_3D), which contains more components.

### 4.6 Delta Aggregates

**Note:** This feature was introduced in Ada 2022.

Previously, we’ve discussed *extension aggregates* (page 205), which are used to assign an object Obj_From of a tagged type to an object Obj_To of a descendant 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.

### 4.6.1 Delta Aggregates for Tagged Records

Let’s reuse the Points package from a previous example:

Listing 66: 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 67: 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
```

(continues on next page)
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:
end Show_Points;

Code block metadata

MD5: 30e62d564d1b35829a5002223966c101

Runtime output

(X => 1.00000E-01, Y => 2.00000E-01, Z => 3.00000E-01)
(X => 0.00000E+00, Y => 0.00000E+00, Z => 3.00000E-01)

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 | 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:

Listing 70: show_points.adb

```ada
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;
```

4.6. Delta Aggregates 211
4.6.2 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.

Listing 71: points.ads

```ada
package Points is
    type Point_3D is record
        X : Float;
        Y : Float;
        Z : Float;
    end record;

    procedure Display (P : Point_3D);
end Points;
```

Listing 72: 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 73: 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;
```
4.6.3 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:

```ada
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;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Points is
  procedure Display (P : Point_3D) is
    begin
      Put ("(");
      for I in P’Range loop
        Put (I’Image
        & " => ",
        & P (I)’Image);
      end loop;
      Put_Line (")");
    end Display;
end Points;
```

```ada
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);
(continues on next page)
```
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;

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 77: show_points.adb

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;

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.
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:

```ada
Listing 78: float_arrays.ads

package Float_Arrays is

  type Float_Array is
    array (Positive range <>) of Float;

  procedure Display (P : Float_Array);

end Float_Arrays;

Listing 79: float_arrays.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Float_Arrays is

  procedure Display (P : Float_Array) is
  begin
    Put ("(");
    for I in P'Range loop
      Put (I'Image & " => " & P (I)'Image);
    end loop;
    Put_Line (")");
  end Display;

end Float_Arrays;

Listing 80: 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;
```

**Code block metadata**

4.6. Delta Aggregates
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\(^{67}\)

\(^{67}\) http://www.ada-auth.org/standards/22rm/html/RM-4-3-4.html
5.1 Unconstrained Arrays

In the Introduction to Ada course\(^{68}\), 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:

Listing 1: measurement defs.ads

```ada
package MeasurementDefs is
  type Measurements is
    array (Positive range <>) of Float;
  -- ^ Bounds are of type Positive,
    but not known at this point.
end MeasurementDefs;
```

Listing 2: show_measurements.adb

```ada
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;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Unconstrained_Arrays.Unconstrained_Array_Example
MD5: a5cdd74dd61e36476431cf675452d1d5

Build output

```
show_measurements.adb:6:04: warning: variable "M" is read but never assigned [-gnatwv]
```

Runtime output

\(^{68}\) https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-types
In this example, the Measurements array type from the Measurement_Defs package is unconstrained. In the Show_Measurements procedure, we declare a constrained object (M) of this type.

The Introduction to Ada course\(^69\) 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\(^70\)

#### 5.1.1 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 3: 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 4: 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
```

\(^69\) [https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-type-instance-bound](https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-type-instance-bound)

```ada
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;
```

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
- A.18.2 The Generic Package Containers.Vectors

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`:

---


Listing 5: multidimensional_arrays_decl.ads

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 A2 has 5 components in the first dimension and 10 components in the second dimension. The three-dimensional array A3 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 A1, A2 and A3 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'Length (2), we're referring to the length of the second dimension of a multidimensional array A. Note that A'Length is equivalent to A'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:

Listing 6: show_multidimensional_arrays.adb

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;
As this simple example shows, we can easily retrieve the length of each dimension. Also, as we've just mentioned, `A1'Length` is equal to `A1'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 MeasurementDefs 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 MeasurementDefs;
```

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

package body MeasurementDefs is

  procedure Show_Indices (M : Measurements) is
  begin
    Put_Line ("---- Indices ----");

    (continues on next page)
```
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 Line (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);
    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 MeasurementDefs;

LISTING 9: show_measurements.adb
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
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 1 2 3 4 5 6 7 8 9 10 11
MON 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
TUE 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
WED 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
THU 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
FRI 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SAT 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SUN 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00

We recommend that you spend some time analyzing this example. Also, we'd like to high-
light 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 for D in M'Range (1) loop and
for H in M'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:
(others => (others => 0.0)).

In the Ada Reference Manual

• 3.6 Array Types

73 http://www.adac.org/standards/22rm/html/RM-3-6.html
5.2.1 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:

Listing 10: multidimensional_arrays_decl.ads

```ada
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;
```

Code block metadata

Unconstrained_Multidimensional_Arrays
MD5: 8637e93db355fddafa3ffe5ce453a0e1

Here, we're declaring the one-dimensional type F1, the two-dimensional type F2 and the three-dimensional type F3.

As is the case with unidimensional arrays, we must specify the bounds when declaring objects of unconstrained multidimensional array types:

Listing 11: 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;
```

Code block metadata

Unconstrained_Multidimensional_Arrays
MD5: 9fb007abbfe230345d80cb315bb834c9

Build output
5.2.2 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 of an array by first specifying a one-dimensional array type T1, and then specifying another one-dimensional array type T2 where each component of T2 is of T1 type:

```
package Array_Of_Arrays_Decl is

  type T1 is array (Positive range <>) of Float;

  type T2 is array (Positive range <>) of T1 (1 .. 10);

end Array_Of_Arrays_Decl;
```

Code block metadata

MD5: fd67739bb21f202615180aa02f5284aa

Note that, in the declaration of T2, we must set the bounds for the T1 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:

```
package MeasurementDefs is

  type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  type Hours is range 0 .. 11;

  subtype Measurement is Float;
```

(continues on next page)
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 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
   begin
   Set_IO_Defaults;
   Put_Line ("---- Values ----");
   for H in M (M'First)'Range loop
      H_IO.Put (H);
   end loop;
   New_Line;
   end Show_Values;
end Measurement_Defs;

Listing 14: measurement_defs.adb
Listing 15: show_measurements.adb

```ada
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

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  1  2  3  4  5  6  7  8  9  10  11
MON 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
TUE 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
WED 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
THU 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
FRI 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SAT 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SUN 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: \( M \left(D\right) \left(H\right) \) 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 multi-dimensional arrays. In fact, we can still use an aggregate with this form: `(others => (others => 0.0)).`
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: ", Character'Size);
  Put_Line ("Wide_Character'Size: ", Wide_Character'Size);
  Put_Line ("Wide_Wide_Character'Size: ", Wide_Wide_Character'Size);
end Show_Wide_Char_Types;
```

(continues on next page)
& Integer'Image
(Wide_Character'Size));
Put_Line ("WideWide_Character'Size: 
& Integer'Image
(WideWide_Character'Size));
end Show_Wide_Char_Types;

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.WideWideWide_Wide_Wide_Wide_Strings.Wide_Char_Types
MD5: a0e9fb9e8d43e9fa707dc8c57f7562f8

Runtime output

Character'Size: 8
Wide_Character'Size: 16
WideWide_Character'Size: 32

Let's look at another example, this time using wide strings:

Listing 2: showWide_string_types.adb

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'Component_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'Component_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 ("------------------------");
Here, all strings (S, WS 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

6.1.1 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:

<table>
<thead>
<tr>
<th>Scalar Type</th>
<th>Text I/O Packages</th>
</tr>
</thead>
</table>
| **Integer**       | • Ada.Integer_Text_IO  
                   | • Ada.Integer_Wide_Text_IO  
                   | • Ada.Integer_Wide_Wide_Text_IO |
| **Long_Integer**  | • Ada.Long_Integer_Text_IO  
                   | • Ada.Long_Integer_Wide_Text_IO  
                   | • Ada.Long_Integer_Wide_Wide_Text_IO |
| **Long_Long_Integer** | • Ada.Long_Long_Integer_Text_IO  
                        | • Ada.Long_Long_Integer_Wide_Text_IO  
                        | • Ada.Long_Long_Integer_Wide_Wide_Text_IO |
| **Float**         | • Ada.Float_Text_IO  
                   | • Ada.Float_Wide_Text_IO  
                   | • Ada.Float_Wide_Wide_Text_IO |
| **Long_Float**    | • Ada.Long_Float_Text_IO  
                   | • Ada.Long_Float_Wide_Text_IO  
                   | • Ada.Long_Float_Wide_Wide_Text_IO |
| **Long_Long_Float** | • Ada.Long_Long_Float_Text_IO  
                        | • Ada.Long_Long_Float_Wide_Text_IO  
                        | • Ada.Long_Long_Float_Wide_Wide_Text_IO |

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>
</table>
| **Integer types** | • Ada.Text_IO.Integer_IO  
                   | • Ada.Wide_Text_IO.Integer_IO  
                   | • Ada.Wide_Wide_Text_IO.Integer_IO |
| **Real types**    | • Ada.Text_IO.Float_IO  
                   | • Ada.Wide_Text_IO.Float_IO  
                   | • Ada.Wide_Wide_Text_IO.Float_IO |

In the Ada Reference Manual

- A.10 Text Input-Output\(^75\)

6.1.2 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, 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 from the Introduction to Ada course, which we adapted for wide-wide strings:

Listing 3: 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
                & Whitespace'Wide_Wide_Image
                & L + I);
end Show_Find_Words;
```

(continues on next page)
while I in S'Range loop
  Find_Token
    (Source  => S,
     Set    => Whitespace,
     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;

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 248).)

In the Ada Reference Manual
  • A.4.6 String-Handling Sets and Mappings\(^{82}\)
  • A.4.7 Wide_String Handling\(^ {83}\)
  • A.4.8 Wide_Wide_String Handling\(^ {84}\)

6.1.3 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 and unbounded strings — 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded_Wide_String</td>
<td>Ada.Strings.Wide_Bounded</td>
</tr>
<tr>
<td>Bounded_Wide_Wide_String</td>
<td>Ada.Strings.Wide_Wide_Bounded</td>
</tr>
<tr>
<td>Unbounded_Wide_String</td>
<td>Ada.Strings.Wide_Unbounded</td>
</tr>
<tr>
<td>Unbounded_Wide_Wide_String</td>
<td>Ada.Strings.Wide_Wide_Unbounded</td>
</tr>
</tbody>
</table>

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>Text I/O Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded_Wide_String</td>
<td>Ada.Wide_Text_IO.Wide_Bounded_IO</td>
</tr>
<tr>
<td>Bounded_Wide_Wide_String</td>
<td>Ada.Wide_Wide_Text_IO.Wide_Wide_Bounded_IO</td>
</tr>
<tr>
<td>Unbounded_Wide_String</td>
<td>Ada.Wide_Text_IO.Wide_Unbounded_IO</td>
</tr>
<tr>
<td>Unbounded_Wide_Wide_String</td>
<td>Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO</td>
</tr>
</tbody>
</table>

Let's look at a simple example:

```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;
```

Let's look at a simple example:
Unbounded wide-wide string: Hello hello

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
- A.4.8 Wide_Wide_String Handling
- A.11 Wide Text Input-Output and Wide Wide Text Input-Output

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

6.2.1 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_Strings</td>
<td>Wide_String type</td>
</tr>
<tr>
<td>.Wide_Wide_Strings</td>
<td>Wide_Wide_String type</td>
</tr>
</tbody>
</table>

Let's look at an example:

90 https://unicode.org/faq/utf_bom.html#gen2
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.Wide_Wide_Unbounded;
use Ada.Strings.Wide_Wide_Unbounded;

procedure Show_WW_UTF_String is
  function To_UWWS
    (Source : Wide_Wide_String)
    return Unbounded_Wide_Wide_String
  renames To_Unbounded_Wide_Wide_String;

  function To_WWS
    (Source : Unbounded_Wide_Wide_String)
    return Wide_Wide_String
  renames To_Wide_Wide_String;

  Hello World Arabic : constant
    UTF_8_String := "عالمي ورحبا";
  WWS_Hello_World_Arabic : constant
    Wide_Wide_String :=
      Decode (Hello World Arabic);

  UWWS : Unbounded_Wide_Wide_String;

  begin
    UWWS := "Hello World:"
     & 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;

(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_String 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.

### 6.2.2 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);
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;

Code block metadata

MD5: 67653dfd377f04b32421cf09b25939fe

Runtime output

Wide_Wide_String Length: 1
Wide_Wide_String Size: 32
UTF-8 String Length: 4
UTF-8 String Size: 32
UTF-8 String: 𝚡

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 x symbol from the Mathematical Alphanumeric Symbols block\(^92\), not the standard "x" from the Basic Latin block\(^93\).)

\(^{92}\) https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols
\(^{93}\) https://en.wikipedia.org/wiki/Basic_Latin_(Unicode_block)
6.2.3 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:

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

with Ada.Strings.UTF_Encoding;
use Ada.Strings.UTF_Encoding;

procedure Show_UTF_8_Strings is

  Symbols_UTF_8 : constant
  UTF_8_String := "♥♫";

begin
  Put_Line ("UTF_8_String: 
             & Symbols_UTF_8);
  Put_Line ("Length: 
             & Symbols_UTF_8'Length'Image);
end Show_UTF_8_Strings;
```

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 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.

---

94 https://en.wikipedia.org/wiki/UTF-8
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:

Listing 8: show_wws_strings.adb

```
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;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WWS_Strings_W8
MD5: 1e5e38e62b412de48d3fa4271bb40bf1

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.)

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 9: show_wws_strings.adb

```
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;
```

(continues on next page)
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:

Listing 10: 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;
```
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:

```ada
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:

```ada
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(code)` function for the UTF-8 code that corresponds to the "★" symbol.
6.2.4 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:

```ada
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:

```ada
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
```

(continues on next page)
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 13: show_ww_utf16_string.adb
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;

In this example, we're calling the \texttt{Wide_Character'Val} function to specify the UTF-16 BE code of the "♥" and "♫" symbols. We're then using the \texttt{Decode} function to convert between the \texttt{UTF_16_Wide_String} and the \texttt{Wide_Wide_String} types.

### 6.3 Image attribute

#### 6.3.1 Overview

In the \textit{Introduction to Ada}\textsuperscript{95} course, we've seen that the \texttt{Image} attribute returns a string that contains a textual representation of an object. For example, we write \texttt{Integer'Image (V)} to get a string for the integer variable \texttt{V}:

\begin{verbatim}
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;
\end{verbatim}

\textsuperscript{95} https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-image-attribute
Runtime output

V: 10

Naturally, we can use the Image attribute with other scalar types. For example:

Listing 15: 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;
```

Code block metadata

MD5: d3369518b610b7bf6c8dcefdecdb0c44

Runtime output

V: 1.00000E+01
S: UNKNOWN

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\(^{96}\)

### 6.3.2 Type'Image and Obj'Image

We can also apply the Image attribute to an object directly:

Listing 16: 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;
```

\(^{96}\) http://www.ada-auth.org/standards/22rm/html/RM-4-10.html
In this example, the \texttt{Integer'Image} (\texttt{V}) and \texttt{V'Image} forms are equivalent.

### 6.3.3 Wider versions of Image

Although we've been talking only about the \texttt{Image} attribute, it's important to mention that each of the wider versions of the string types also has a corresponding \texttt{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>\texttt{Image}</td>
<td>\texttt{String}</td>
</tr>
<tr>
<td>\texttt{Wide_Image}</td>
<td>\texttt{Wide_String}</td>
</tr>
<tr>
<td>\texttt{Wide_Wide_Image}</td>
<td>\texttt{Wide_Wide_String}</td>
</tr>
</tbody>
</table>

Let's see a simple example:

Listing 17: show_wide_wide_image.adb

```ada
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 \texttt{Wide_Wide_Image} attribute to retrieve a string of \texttt{Wide_Wide_String} type for the floating-point variable \texttt{F}.  

248 Chapter 6. Strings
### 6.3.4 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:

**Listing 18: simple_records.ads**

```ada
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;
```

**Listing 19: show_non_scalar_image.adb**

```ada
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);
```

(continues on next page)
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

Runtime output
R_A: (access 10fb2c0)
R_A.all:
(F => 1.00000E+01,
I => 4)
N_R: (NULL RECORD)
R_A: null

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 Types\(^{97}\) section of the Ada Reference Manual, as it was only applied to those types. Now, it is part of the new Image Attributes\(^{98}\) section.

Let's see another example, this time with arrays:

Listing 20: show_array_image.adb

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;

\(^{97}\) [http://www.ada-auth.org/standards/22rm/html/RM-3-5.html]
\(^{98}\) [http://www.ada-auth.org/standards/22rm/html/RM-4-10.html]
6.3.5 Image attribute for tagged types

In addition to untagged types, we can also use the Image attribute with tagged types. For example:

Listing 21: simple_records.ads

```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;
```

Listing 22: show_tagged_image.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Simple_Records; use Simple_Records;
```

(continues on next page)
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_A: " & R_C'Image);
end Show_Tagged_Image;

Code block metadata

MD5: 164bd17c99115a5cafb09c99f40c1578c

Runtime output

R: {SIMPLE_RECORDS.RECobject}
R_Class: SIMPLE_RECORDS.REC'{SIMPLE_RECORDS.RECobject}
R_A: {SIMPLE_RECORDS.REC_CHILDobject}

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.

6.3.6 Image attribute for task and protected types

We can also apply the Image attribute to protected objects and tasks:

Listing 23: simple_tasking.ads

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 24: simple_tasking.adb

package body Simple_Tasking is

  protected body Protected_Float is

  end Protected_Float;

  protected body Protected_Null is
Listing 25: 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;
```

6.4 Put_Image aspect

In this example, we display information about the protected object PF, the componentless protected object PN and the task T1.

Note: This feature was introduced in Ada 2022.
6.4.1 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:

```
pragma Ada_2022;
with Ada.Strings.Text_Buffers;
package Show_Put_Image is
type T is null record
  with Put_Image => Put_Image_T;
  -- ^ Custom version of Put_Image
use Ada.Strings.Text_Buffers;
procedure Put_Image_T
  (Buffer : in out Root_Buffer_Type'Class;
   Arg : T);
end Show_Put_Image;
```

```
package body Show_Put_Image is
procedure Put_Image_T
  (Buffer : in out Root_Buffer_Type'Class;
   Arg : T) is
pragma Unreferenced (Arg);
begin
  -- Call Put with customized
  -- information
  Buffer.Put ("<custom info>"ัญ);  
end Put_Image_T;
end Show_Put_Image;
```

Code block metadata

MD5: cbdd77a9e6cc30f3604c0901536d87aa

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 261).)

**In the Ada Reference Manual**
- Image Attributes[^99]

### 6.4.2 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 28: 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 29: 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 Put_Float_Integer;

end Custom_Numerics;
```

[^99]: http://www.ada-auth.org/standards/22rm/html/RM-4-10.html

---

6.4. Put_Image aspect 255
Advanced Journey With Ada: A Flight In Progress

Listing 30: show_put_image.adb

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);
  Put_Line ("V = ",
            & V'Image);
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.)

6.4.3 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:

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:

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

(continues on next page)
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 246) about the Image attribute.)

### 6.4.4 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:

Listing 31: untagged_put_image.ads

```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 32: untagged_put_image.adb

```ada
package body Unagged_Put_Image is
  procedure Put_Image_T
      (Buffer : in out Root_Buffer_Type'Class;
       Arg : T) is
    pragma Unreferenced (Arg);
```

(continues on next page)
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 Untagged_Put_Image;

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;

Code block metadata

MD5: b0a115967ec5f2deea19967d22266b4

Runtime output

T'Image : Put_Image_T
T_Derived_1'Image : Put_Image_T
T_Derived_2'Image : Put_Image_T_Derived_2

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.
6.4.5 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 34: tagged_put_image.ads

```ada
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; 
    Arg : T);

  type T_Child_1 is new T with record
    I1 : Integer;
  end record;

  type T_Child_2 is new T with null record;

  type T_Child_3 is new T with record
    I3 : Integer := 0;
  end record
  with Put_Image => Put_Image_T_Child_3;

  procedure Put_Image_T_Child_3
    (Buffer : in out Root_Buffer_Type; 
    Arg : T_Child_3);

end Tagged_Put_Image;
```

Listing 35: tagged_put_image.adb

```ada
package body Tagged_Put_Image is

  procedure Put_Image_T
    (Buffer : in out Root_Buffer_Type; 
    Arg : T) is
    pragma Unreferenced (Arg);
  begin
    Buffer.Wide_Wide_Put ("Put_Image_T");
  end Put_Image_T;

  procedure Put_Image_T_Child_3
    (Buffer : in out Root_Buffer_Type; 
    Arg : T_Child_3) is
    pragma Unreferenced (Arg);
  begin
    Buffer.Wide_Wide_Put ("Put_Image_T_Child_3");
  end Put_Image_T_Child_3;
```

(continues on next page)
Listing 36: show_tagged_put_image.adb

```ada
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;
  Obj_T_Child_2 : T_Child_2;
  Obj_T_Child_3 : T_Child_3;
begin
  Put_Line ("T'Image : " & Obj_T'Image);
  Put_Line ("--------------------");
  Put_Line ("T_Child_1'Image : " & Obj_T_Child_1'Image);
  Put_Line ("--------------------");
  Put_Line ("T_Child_2'Image : " & Obj_T_Child_2'Image);
  Put_Line ("--------------------");
  Put_Line ("T_Child_3'Image : " & Obj_T_Child_3'Image);
  Put_Line ("--------------------");
  Put_Line ("T'Class'Image : " & T'Class (Obj_T_Child_1)'Image);
end Show_Tagged_Put_Image;
```

In this example, we declare the type T and its derived types T_Child_1, T_Child_2 and T_Child_3. When running this code, we see that:

- for both T_Child_1 and T_Child_2, the parent's Put_Image aspect (the Put_Image_T procedure) is called and its information is combined with the information from the type extension;
  - The information from the parent's Put_Image_T procedure is presented in an aggregate syntax — in this case, this results in (Put_Image_T).
- For the T_Child_1 type, the I1 component of the type extension is displayed by calling a default version of the Put_Image procedure for that component — (Put_Image_T with I1 => 0) is displayed.

- For the T_Child_2 type, no additional information is displayed because this type has a null extension.

- For the T_Child_3 type, the Put_Image_T_Child_3 procedure, which was indicated in the Put_Image aspect of the type, is used.

Finally, class-wide types (such as T'Class) include additional information. Here, the tag of the specific derived type is displayed first — in this case, the tag of the T_Child_1 type — and then the actual information for the derived type is displayed.

6.5 Universal text buffer

In the previous section (page 253), 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.

6.5.1 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

6.5.2 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 37: custom_numerics.ads

```ada
pragma Ada_2022;
with Ada.Strings.Text_Buffers;

package Custom_Numerics is

  type Float_Integer is record
    F : Float;
    I : Integer;
  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 38: custom_numerics.adb

```ada
package body Custom_Numerics is

procedure Put_Float_Integer
  (Buffer : in out Root_Buffer_Type'Class;
   Arg : Float_Integer)
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
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

Listing 39: 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);
  Put_Line ("V = 
            & V'Image);  
end Show_Put_Image;
```

Code block metadata

MD5: af95f9fe4064e8a9d7aebe14d7f561f7

Runtime output

```
V = (F : 1.00200E+02
     I : 100)
```

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.

6.5. Universal text buffer
7.1 Modular Types

In the Introduction to Ada course, we've seen that Ada has two kinds of integer type: `signed` and `modular` 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

7.1.1 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:

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

with Num_Types; use Num_Types;

procedure Show_Modular is
```

---

101 https://learn.adacore.com/courses/intro-to-ada/chapters/strongly.Typed_language.html#intro-ada-integers
102 https://learn.adacore.com/courses/intro-to-ada/chapters/stronglyTyped_language.html#intro-ada-unsigned-types
103 http://www.ada-auth.org/standards/22rm/html/RM-3-5-4.html
When we run this example, we get 4294967296, which is equal to $2^{32}$.

### 7.1.2 Mod Attribute

**Note:** This section was originally written by Robert A. Duff and published as *Gem #26: The Mod Attribute*.

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'Bases, 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](page11).)

Operations on modular integers use modular (wraparound) arithmetic. For example:

```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);
   X := -X;
   Put_Line (X'Image);
end Show_Modular;
```

**Code block metadata**

MD5: e9ac61d2e43585f002fe2b79544ef9d7

**Runtime output**

4294967296

104 https://www.adacore.com/gems/gem-26
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`:

```
Listing 4: show_modular.adb

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:

```
Listing 5: show_modular.adb

with Ada.Text_IO; use Ada.Text_IO;
with Num_Types; use Num_Types;

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:

```ada
generic
    type Formal_Modular is mod <>;
package Mod_Attribute is
    function F return Formal_Modular;
end Mod_Attribute;
```

```ada
package body Mod_Attribute is
    A_Signed_Integer : Integer := -1;
    function F return Formal_Modular is
        begin
            return Formal_Modular'Mod
                (A_Signed_Integer);
        end F;
end Mod_Attribute;
```

**Code block metadata**

**Project**: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Mod_Attribute  
**MD5**: b2f227b8d4f14cd36508bf33c403f751

In this example, `F` will return the all-ones bit pattern, for whatever modular type is passed to `Formal_Modular`.

### 7.1.3 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 8: 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 9: 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;

  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;
    end Display;

end Crc_Defs;
```

(continues on next page)
(continued from previous page)

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;

Listing 10: show_crc.adb

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 ** 8 and / 2 ** 8, which shift left and right, respectively, by eight bits. We also use the xor operator.

7.2 Numeric Literals

7.2.1 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 11: real_integer_literals.adb
procedure Real_Integer_Literals is
  Integer_Literal : constant := 365;
  Real_Literal   : constant := 365.2564;
begin
  Put_Line ("Integer Literal: 
    & Integer_Literal'Image);
  Put_Line ("Real Literal: 
    & Real_Literal'Image);
end Real_Integer_Literals;

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:

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#;
  Based_Integer_Exp : constant := 5#243#e1;
  Based_Real   : constant := 2#1_0110_1101.0100_0001_1010_0011_0111#;
  Based_Real_Exp : constant := 7#1.031_153_643#e3;
begin
  F_IO.Default_Fore := 3;
  F_IO.Default_Aft := 4;
  F_IO.Default_Exp := 0;
  Put_Line ("Dec_Integer: 
    & Dec_Integer'Image);
35 Put ("Dec.Real: ");
36 F_IO.Put (Item => Dec_Real);
37 New_Line;
38
40 F_IO.Put (Item => Dec_Real_Exp);
41 New_Line;
42
43 Put_Line ("Based_Integer: "
44 & Based_Integer'Image);
45 Put_Line ("Based_Integer_Exp: "
46 & Based_Integer_Exp'Image);
47
48 Put ("Based.Real: ");
49 F_IO.Put (Item => Based_Real);
50 New_Line;
51
52 Put ("Based.Real.Exp: ");
53 F_IO.Put (Item => Based_Real_Exp);
54 New_Line;
55
56 end Decimal_Based_Literals;

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.

7.2.2 Features and Flexibility

Note: This section was originally written by Franco Gasperoni and published as Gem #7: The Beauty of Numeric Literals in Ada\textsuperscript{105}.

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:

\textsuperscript{105} https://www.adacore.com/gems/ada-gem-7

Chapter 7. Numerics
Advanced Journey With Ada: A Flight In Progress

Listing 13: ada_numeric_literals.adb

```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);

use Ada.Text_IO;

begin
  Put_Line ("Z = " & Float'Image (Z));
end Ada_Numeric_Literals;
```

Code block metadata

MD5: 8f6516730fa98f08234bb59488431aaf

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:

```
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);

use Ada.Text_IO;

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;
```

Code block metadata

MD5: 0959ec5e4aafcd245c5a15597ac9b7e

7.2. Numeric Literals 273
Runtime output

Bin_136 = 136
Oct_136 = 136
Dec_136 = 136
Hex_136 = 136

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:

```ada
Lights_On : constant := 2#1000_1000#;
Lights_Off : constant := 2#0111_0111#;
```

and have the ability to turn on/off the lights as follows:

```ada
Output_Devices := Output_Devices or Lights_On;
Output_Devices := Output_Devices and Lights_Off;
```

Here’s the complete example:

```ada
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

  use Ada.Text_IO;

begin
  Output_Devices := Output_Devices or
                   Lights_On;
  Output_Devices := Output_Devices and
                   Lights_Off;
  Put_Line ("Output_Devices (lights on ) = "
            & Byte'Image (Output_Devices));
  Output_Devices := Output_Devices or
                   Lights_On;
  Output_Devices := Output_Devices and
                   Lights_Off;
  Put_Line ("Output_Devices (lights off) = "
            & Byte'Image (Output_Devices));
end Ada_Numeric_Literals;
```

Listing 15: ada_numeric_literals.adb
Of course, we can also use records with representation clauses (page 97) 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:

```ada
package Literal_Binaries is
    Kilobyte   : constant := 2#1#e10;
    Megabyte   : constant := 2#1#e20;
    Gigabyte   : constant := 2#1#e30;
    Terabyte   : constant := 2#1#e40;
    Petabyte   : constant := 2#1#e50;
    Exabyte    : constant := 2#1#e60;
    Zettabyte  : constant := 2#1#e70;
    Yottabyte  : constant := 2#1#e80;
end Literal_Binaries;
```

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, \(2#1#e10 = 1 \times 2^{10} = 1,024\) (in base 10), whereas \(16#F#e+2 = 15 \times 16^2 = 15 \times 256 = 3,840\) (in base 10).

Based numbers apply equally well to real literals. We can, for instance, write:

```ada
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.

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 \(32l\) 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 := 
  0.3333333333333333333333333333333333;
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 17: ada_numeric_literals.adb

with Ada.Text_IO;

procedure Ada_Numeric_Literals is
  One_Third : constant := 3#1.0#e-1;
  -- same as 1.0/3.0
  Zero : constant := 1.0 - 3.0 * One_Third;
  pragma Assert (Zero = 0.0);
  One_Third_Approx : constant := 
    0.3333333333333333333333333333333333;
  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

Zero = 0.00000E+00
Zero_Approx = 1.00000E-29

Along these same lines, we can write:

Listing 18: ada_numeric_literals.adb

with Ada.Text_IO;

with Literal_Binaries; use Literal_Binaries;

procedure Ada_Numeric_Literals is
  Big_Sum : constant := 1 +
    Kilobyte +
(continues on next page)
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;

Code block metadata
MD5: 7bda6442e608271d12b0827b563f0d702

Runtime output
Nil = 0

and be guaranteed that Nil is equal to zero.

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

7.3.1 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 19: show_machine_radix.adb

```ada
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;
```

**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 20: show_machine_mantissa.adb

```ada
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);
  Put_Line (
    "Long_Long_Float'Machine_Mantissa: " & Long_Long_Float'Machine_Mantissa'Image);
end Show_Machine_Mantissa;
```

**Code block metadata**

MD5: 88680df680f1db4ff803912850370551

Chapter 7. Numerics
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:

```adb
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;
```

### Code block metadata

MD5: 9766e06faaf1fc2ca48dd0bc0461b247

### Runtime output

- 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 \texttt{Float'Machine_Emin} and \texttt{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 \texttt{Machine_Radix} attribute that we've seen previously. Let's calculate the minimum and maximum value of the exponent for the \texttt{Float} type on a typical PC:

- Value of minimum exponent: \texttt{Float'Machine_Radix ** Float'Machine_Emin}.
  - In our target platform, this is $2^{-125} = 2.3509887016457501594 \times 10^{-38}$.
- Value of maximum exponent: \texttt{Float'Machine_Radix ** Float'Machine_Emax}.
  - In our target platform, this is $2^{128} = 3.40282366920938463463 \times 10^{38}$.

\textbf{Attribute: Digits}

\texttt{Digits} is an attribute that returns the requested decimal precision of a floating-point subtype. Let's see an example:

\begin{verbatim}
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;
\end{verbatim}

\textbf{Code block metadata}

MD5: cd1c88054f7d54703760a852d08acb6d

\textbf{Runtime output}

\begin{itemize}
  \item Float'Digits: 6
  \item Long_Float'Digits: 15
  \item Long_Long_Float'Digits: 18
\end{itemize}

Here, the requested decimal precision of the \texttt{Float} type is six digits.

Note that we said that \texttt{Digits} is the \textit{requested} level of precision, which is specified as part of declaring a floating point type. We can retrieve the actual decimal precision with \texttt{Base'Digits}. For example:

\begin{verbatim}
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;
\end{verbatim}
Advanced Journey With Ada: A Flight In Progress

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\(^{108}\).
- 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 24: show_boolean_attributes.adb

```ada
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);
  Put_Line ("Float'Signed_Zeros: " & Float'Signed_Zeros'Image);
  Put_Line ("Long_Float'Signed_Zeros: " & Long_Float'Signed_Zeros'Image);
  Put_Line ("Long_Long_Float'Signed_Zeros: " & Long_Long_Float'Signed_Zeros'Image);
  Put_Line ("Float'Machine_Rounds: " & Float'Machine_Rounds'Image);
  Put_Line ("Long_Float'Machine_Rounds: " & Long_Float'Machine_Rounds'Image);
  Put_Line ("Long_Long_Float'Machine_Rounds: " & Long_Long_Float'Machine_Rounds'Image);
```

\(^{108}\) https://en.wikipedia.org/wiki/Subnormal_number

7.3. Floating-Point Types 281
& Long_Float'Machine_Rounds'Image);  
Put_Line  
("Long_Long_Float'Machine_Rounds: "  
& Long_Long_Float'Machine_Rounds'Image);  
Put_Line  
("Float'Machine_Overflows: "  
& Float'Machine_Overflows'Image);  
Put_Line  
("Long_Float'Machine_Overflows: "  
& Long_Float'Machine_Overflows'Image);  
Put_Line  
("Long_Long_Float'Machine_Overflows: "  
& Long_Long_Float'Machine_Overflows'Image);  
end Show_Boolean_Attributes;

Code block metadata

Rounds_Overflows  
MD5: b3f72c212cf00e697fe144a87eb72339

Runtime output

Float'Denorm: TRUE  
Long_Float'Denorm: TRUE  
Long_Long_Float'Denorm: TRUE  
Float'Signed_Zeros: TRUE  
Long_Float'Signed_Zeros: TRUE  
Long_Long_Float'Signed_Zeros: TRUE  
Float'Machine_Rounds: TRUE  
Long_Float'Machine_Rounds: TRUE  
Long_Long_Float'Machine_Rounds: TRUE  
Float'Machine_Overflows: FALSE  
Long_Float'Machine_Overflows: FALSE  
Long_Long_Float'Machine_Overflows: FALSE

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).
### 7.3.2 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:

Listing 25: show_exponent_fraction_compose.adb

```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**

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5: d2e61b6b9a7a50861145f6b65e9fac39</td>
</tr>
</tbody>
</table>

**Runtime output**

```
Float'Fraction (1.0): 5.00000E-01
Float'Fraction (0.25): 5.00000E-01
```
To understand this code example, we have to take this formula into account:

\[ \text{Value} = \text{Fraction} \times \text{Machine\_Radix}^{\text{Exponent}} \]

Considering that the value of \text{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:

\[ 0.5 \times 2^1 = 1.0 \]

For the value 0.25, we get a fraction of 0.5 and a machine exponent of -1, which is the result of \[ 0.5 \times 2^{-1} = 0.25 \]. We can use the \text{Compose} attribute to perform this calculation. For example, \text{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.

\textbf{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:

```ada
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;
```

The scaling is calculated with this formula:

\[ \text{scaling} = \text{value} \times \text{Machine\_Radix}^{\text{machine exponent}} \]

For example, on a typical PC with a machine radix of two, \text{Float\_Scaling (0.25, 3)} = 2.0 corresponds to

\[ 0.25 \times 2^3 = 2.0 \]
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 27: 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;
```

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 28: 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): ",
```

(continues on next page)
& Float'Unbiased_Rounding (0.5)'Image);
Put_Line
("Float'Unbiased_Rounding (1.5): 
& Float'Unbiased_Rounding (1.5)'Image);
Put_Line
("Float'Machine_Rounding (0.5): 
& Float'Machine_Rounding (0.5)'Image);
Put_Line
("Float'Machine_Rounding (1.5): 
& Float'Machine_Rounding (1.5)'Image);
end Show_Roundings;

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'Rounding (4.5) = Float'Ceiling (4.5) = 5.0.
  - -4.5 is rounded-down to -5.0, i.e. Float'Rounding (-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, \( \text{Float'Remainder (1.25, 0.5)} = 0.25 \). Let's briefly discuss the details of this operations. 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 \times 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:

Listing 29: show_truncation_remainder_adjacent.adb

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

procedure Show_Truncation_Remainder_Adjacent is
begin
  Put_Line ("Float'Truncation (1.55): " & Float'Truncation (1.55)'Image);
  Put_Line ("Float'Truncation (-1.55): " & Float'Truncation (-1.55)'Image);
  Put_Line ("Float'Remainder (1.25, 0.25): " & Float'Remainder (1.25, 0.25)'Image);
  Put_Line ("Float'Remainder (1.25, 0.5): " & Float'Remainder (1.25, 0.5)'Image);
  Put_Line ("Float'Remainder (1.25, 1.0): " & Float'Remainder (1.25, 1.0)'Image);
  Put_Line ("Float'Remainder (1.25, 2.0): " & Float'Remainder (1.25, 2.0)'Image);
  Put_Line ("Float'Adjacent (1.0e-83, 0.0): " & Float'Adjacent (1.0e-83, 0.0)'Image);
  Put_Line ("Float'Adjacent (1.0e-83, 1.0): " & Float'Adjacent (1.0e-83, 1.0)'Image);
end Show_Truncation_Remainder_Adjacent;
```

7.3. Floating-Point Types
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 30: show_copy_sign_leading_part_machine.adb

```ada
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);
```

(continues on next page)
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 \(1,000,000,000,000,000.0\)):

Listing 31: 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 \( V \gets 1.0E+15 \) assignment of the code example as \( V \gets \text{Float}'\text{Machine} (1.0E+15) \), 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 \( 1.0 \times 10^{15} \). Of course, writing \( V \gets 1.0E+15 \) or \( V \gets \text{Float}'\text{Machine} (1.0E+15) \) 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 \( 1.0 \times 10^{15} \) and the actual value that is assigned to \( V \). We can do this by using the Machine attribute in the calculation:

```
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 - \text{Float}'\text{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 \( 1.3009 \times 10^7 \). (Actually, the value that we might see is \( 1.3008896 \times 10^7 \), which is a version of \( 1.3009 \times 10^7 \) that is representable on the target machine.)

When we write \( 1.0E+15 - \text{Float}'\text{Machine} (1.0E+15) \):
• the first value in the operation is the universal real value $1.0 \times 10^{15}$, while
• the second value in the operation is a version of the universal real value $1.0 \times 10^{15}$
that is representable on the target machine.

This also means that, in the assignment to $V$, we're actually writing $V := \text{Float}'\text{Machine}$
$(1.0E+15 - \text{Float}'\text{Machine} \ (1.0E+15))$, 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 $1.3008896 \times 10^7$ instead of $1.3009 \times 10^7$ when we run this
application.

### 7.4 Fixed-Point Types

In this section, we discuss various attributes and operations related to fixed-point types.

**In the Ada Reference Manual**

- 3.5.10 Operations of Fixed Point Types\(^{109}\)
- A.5.4 Attributes of Fixed Point Types\(^{110}\)

### 7.4.1 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:

```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;
```

---


Advanced Journey With Ada: A Flight In Progress

Runtime output

| T3_D3'Machine_Radix: 2 |
| TQ31'Machine_Radix: 2 |

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 34: 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 35: 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 36: show_fixed_type_info.adb

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

procedure Show_Fixed_Type_Info is
begin
  Put_Line ("Size : 
            & Fixed'Size'Image);
  Put_Line ("Small : 
            & Fixed'Small'Image);
  Put_Line ("Delta : 
            & Fixed'Delta'Image);
  Put_Line ("First : 
            & Fixed'First'Image);
  Put_Line ("Last : 
            & Fixed'Last'Image);
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 \(340,282,366,762,482,138,434,845,932,244,680,310,784.0\), 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:

7.4. Fixed-Point Types
• 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: \( T\text{'Small} \leq T\text{'Delta} \).

• 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 37: 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 38: 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);
end Show_Fixed_Small_Delta;
```

(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 \( TQ15'\text{Small} (2^{-31}) \) to \( V \). This value is smaller than the type's \( \text{delta} (2^{-15}) \). Even though \( V'\text{Size} \) is 32 bits, \( V'\text{Delta} \) indicates 16-bit precision, and \( TQ15'\text{Small} \) 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 \( TQ15'\text{Delta} (2^{-15}) \) to \( V \) — we see, as expected, that \( V \) has the same value as the \( \text{delta} \).

**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:

```
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".

### 7.4.2 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:

**Listing 41: show_decimal_digits.adb**

```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: "
           & T3_D6'Digits'Image);
  Put_Line ("T3_D2'Digits: "
           & T3_D2'Digits'Image);
end Show_Decimal_Digits;
```

7.4. Fixed-Point Types
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 42: 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'Scale'image);
   Put_Line ("T3_D6'Scale: " & T3_D6'Scale'image);
   Put_Line ("T9_D12'Scale: " & T9_D12'Scale'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:

Listing 43: show_decimal_round.adb

```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.

### 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. 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:

In the Ada Reference Manual

- Big Numbers
- Big Integers
- Big Reals

7.5.1 Overview

Let's start with a simple declaration of big numbers:

```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;
```

Runtime output

- Big Numbers
- Big Integers
- Big Reals

---

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 \( (+, -, \text{abs}) \) and binary operators \( (+, -, *, /, **, \text{Min} \) and \( \text{Max} \) \) are available to us. For example:

**Listing 45: show_simple_big_numbers_operators.adb**

```ada
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;

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.
7.5.2 Factorial

A typical example is the factorial\(^\text{115}\): a sequence of the factorial of consecutive small numbers can quickly lead to big numbers. Let’s take this implementation as an example:

Listing 46: factorial.ads

```ada
function Factorial (N : Integer) return Long_Long_Integer;
```

Listing 47: factorial.adb

```ada
function Factorial (N : Integer) return Long_Long_Integer is
begin
Fact := 1;
for I in 2 .. N loop
Fact := Fact * Long_Long_Integer (I);
end loop;
return Fact;
end Factorial;
```

Listing 48: 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
```

\(^\text{115}\) https://en.wikipedia.org/wiki/Factorial
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!):
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.
7.5.3 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 an 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:

```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_Integer_Conversion is
  BI : Big_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

**Project:** Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Integer_Conversion  
**MD5:** 84f55568b26bf6c1c6f0b06391e8ac0f

**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:

```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_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)
begin
  BI := To_Big_Integer (LLI);
  Put_Line ("BI: " & BI'Image);
  LLI := From_Big_Integer (BI + 1);
  Put_Line ("LLI: " & LLI'Image);
  BI := To_Big_Integer (U_32);
  Put_Line ("BI: " & BI'Image);
  U_32 := From_Big_Integer (BI + 1);
  Put_Line ("U_32: " & U_32'Image);
end Show_Arbitrary_Big_Integer_Conversion;

In this example, 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:

Listing 54: show_big_real_floating_point_conversion.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Reals;
use Ada.Numerics.Big_Numbers.Big_Reals;
procedure Show_Big_Real_Floating_Point_Conversion is
  type D10 is digits 10;
```

(continues on next page)
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'Image);

  LF := From_Big_Real (BR + 1.0);
  Put_Line ("LF: " & LF'Image);

  BR := To_Big_Real (F10);
  Put_Line ("BR: " & BR'Image);

  F10 := From_Big_Real (BR + 0.1);
  Put_Line ("F10: " & F10'Image);
end Show_Big_Real_Floating_Point_Conversion;

Code block metadata
MD5: 531c59a06b46c2074bc5378b5dcdd35

Runtime output
BR: 2.000
LF: 3.00000000000000E+00
BR: 1.999
F10: 2.099000000E+00

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;
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):

```
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;
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

14  BI : Big_Integer;
15  BR : Big_Real;

begin
18  I := 12345;
19  BR := To_Real (I);
20  Put_Line ("BR (from I): " & BR’Image);
21
22  BI := 123456;
23  BR := To_Big_Real (BI);
24  Put_Line ("BR (from BI): " & BR’Image);
25
26  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 57: 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 function 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 68), 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.
7.5.4 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:

```
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;
begín
  D := 3;
  N := 2;
  BI := Big_Natural (D / Big_Positive (N));
  Put_Line ("BI: " & BI'Image);
end Show_Big_Positive_Natural;
```

---

**Code block metadata**


---

Chapter 7. Numerics
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.

### 7.5.5 Other operators for big integers

We can use the `mod` and `rem` operators with big integers:

```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;
begin
  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:

```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;
begin
  BI := Greatest_Common_Divisor (145, 25);
  Put_Line ("BI: " & BI'Image);
end Show_Big_Integer_Greatest_Common_Divisor;
```
In this example, we retrieve the greatest common divisor of 145 and 25 (i.e.: 5).

### 7.5.6 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 61: show_big_real_quotient_conversion.adb

```ada
pragma Ada_2022;

with Ada.Text_IO; use Ada.Text_IO;

with Ada.Numerics.Big_Numbers.Big_Reals;
use Ada.Numerics.Big_Numbers.Big_Reals;

procedure Show_Big_Real_Quotient_Conversion
is
    BR : Big_Real;
begin
    BR := 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;
```

Runtime output

```
BR: 0.666
Q: 2 / 3
```

(continues on next page)
In this example, we assign \( \frac{2}{3} \) to \( BR \) — we could have used the `From_Quotient_String` function as well. Also, we use the `To_Quotient_String` to get a string that represents the quotient. Finally, we use the `Numerator` and `Denominator` functions to retrieve the values, respectively, of the numerator and denominator of the quotient (as big integers) of the big real variable.

### 7.5.7 Range checks

Previously, we've talked about the `Big_Natural` and `Big_Positive` subtypes. In addition to those subtypes, we have the `In_Range` function for big numbers:

#### Listing 62: show_big_numbers_in_range.adb

```ada
pragma Ada_2022;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
with Ada.Numerics.Big_Numbers.Big_Reals;

procedure Show_Big_Numbers_In_Range is
  BI : Big_Integer := 1023;
  BR : Big_Real := 1023.9;
  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
  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");
  end if;

(continues on next page)
```
end if;
end Show_Big_Numbers_In_Range;

Code block metadata

MD5: 9c85e8374db1095142260f45c4c4e7e1

Runtime output

BI (1023) is in the 0 .. 1024 range
BR (1023.900) is in the 0.000 .. 1024.000 range

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.
Part II

Control Flow
8.1 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

8.1.1 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):

Listing 1: show_expression_elements.adb

```
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;
```

(continues on next page)

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 `M1` 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;
  - null;
  - a string literal;
  - an aggregate (page 167);
  - a name;
  - an allocator (like new Integer);
  - a parenthesized expression (page 323);
  - a conditional expression (page 326);
  - a quantified expression (page 328);
  - a declare expression (page 332).

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:

\[
\text{simple_expression ::= [unary_adding_operator] term \{binary_adding_operator term\}}
\]

\[
\text{term ::= factor \{multiplying_operator factor\}}
\]

\[
\text{factor ::= primary [** primary] | abs primary | not primary}
\]

\[
\text{primary ::= numeric_literal | null | string_literal | aggregate | name | allocator | (expression) |
\text{(conditional_expression) | (quantified_expression) |
\text{(declare_expression) }}
\]

Later on in this chapter, we discuss conditional expressions (page 326), quantified expressions (page 328) and declare expressions (page 332) in more details.

In the relation M2 \text{ in } \text{ Off | A} from the code example, Off | 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:

\[
\text{expression ::= relation \{and relation\} |
\text{relation \{and then relation\} |
\text{relation \{or relation\} |
\text{relation \{or else relation\} |
\text{relation \{xor relation\}}} (continues on next page)}
\]

8.1. Expressions: Definition
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\textsuperscript{118} of the Ada Reference Manual.

In the Ada Reference Manual

- 4.4 Expressions\textsuperscript{119}
- 4.5.2 Relational Operators and Membership Tests\textsuperscript{120}

8.1.2 Numeric expressions

The expressions we’ve seen so far had the \textbf{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:

\begin{verbatim}{language=ada}
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
  --        primary
  --        factor
  --        term
  --        numeric literal
  --        primary
  --        numeric literal
  --        primary
  --        factor
  --        term
  --        binary adding operator
  --        unary adding operator
  --        term
  --        simple expression
  --

\end{verbatim}

\textsuperscript{118} \url{http://www.ada-auth.org/standards/22rm/html/RM-4-4.html}
\textsuperscript{119} \url{http://www.ada-auth.org/standards/22rm/html/RM-4-4.html}
\textsuperscript{120} \url{http://www.ada-auth.org/standards/22rm/html/RM-4-5-2.html}
In the Ada Reference Manual

• 4.4 Expressions\textsuperscript{121}

\section*{8.1.3 Other expressions}

Expressions aren't limited to the \texttt{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 \texttt{section on expressions}\textsuperscript{122} of the Ada Reference Manual for further details.

\section*{8.1.4 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;
  Dummy := (2 + C1) * C2;
  -- ^^ name
  -- ^^ primary
  -- ^^ factor
  -- ^^ term
end Show_Parenthesized_Expressions;
```

\textsuperscript{121} http://www.ada-auth.org/standards/22rm/html/RM-4-4.html
\textsuperscript{122} http://www.ada-auth.org/standards/22rm/html/RM-4-4.html
In this example, we first start with the single expression \((2 + C1) \times C2\), 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:

```ada
procedure Show_Name_In_Expression is
  type Mode is (Off, A, B, C, D);
begin
  M1 : Mode;
  begin
    M1 := A;
    case M1 is
      when Off | D => null;
      when A | B | C => M1 := D;
```

(continues on next page)
Here, the case statement expects a selecting expression. In this case, M1 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 351).

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.)
8.2 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 \( \text{if} \ldots \text{then} \ldots \text{else} \) form) or a "case-expression" (in the \( \text{case} \ldots \text{is when} \Rightarrow \) form).

The Max function in the following code example is an expression function implemented with a conditional expression — an if-expression, to be more precise:

Listing 8: expr_func.ads

```ada
package Expr_Func is

function Max (A, B : Integer) return Integer is
  (if A >= 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:

Listing 9: expr_func.ads

```ada
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 elsif branch (and therefore be more complicated).

The code above corresponds to this more verbose version:

Listing 10: expr_func.ads

```ada
package Expr_Func is

type State is (Off, On, Waiting, Invalid);

function Toggled (S : State) return State;

end Expr_Func;
```
Listing 11: expr_func.adb

```ada
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;
```

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:

Listing 12: expr_func.ads

```ada
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;
```

Note that we use commas in case-expressions to separate the alternatives (the when expressions). The code above corresponds to this more verbose version:

Listing 13: expr_func.ads

```ada
package Expr_Func is

type State is (Off, On, Waiting, Invalid);

function Toggled (S : State) return State;
end Expr_Func;
```

8.2. Conditional Expressions

327
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;

8.3 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'Range => A (I) = 0, and
- "at least one value of array A must be zero" into for some I in A'Range => A (I) = 0.

In the quantified expression for all I in A'Range => 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;

8.3. Quantified Expressions
As you might have expected, we can rewrite a quantified expression as a loop in the for I in A'Range loop if ... return ... form. In the code below, we're implementing Is_Zero and Has_Zero using loops and conditions instead of quantified expressions:

Listing 18: int_arrays.ads

```ada
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;
```

Listing 19: int_arrays.adb

```ada
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 False;
    end Has_Zero;

end Int_Arrays;
```

(continues on next page)
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;

So far, we've seen quantified expressions using indices — e.g. for all I in A'Range => .... We could remove 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:

8.3. Quantified Expressions
Listing 21: int_arrays.ads

```ada
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;
```

In the Ada Reference Manual

- 4.5.8 Quantified Expressions

### 8.4 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

```ada
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;
```

Code block metadata

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;

  -- Expression function using declare expression:
  --
  function Avg (Arr : Integer_Array) return Float is
    (declare
      A : Integer_Array renames Arr;
      S : constant Float := Float (Sum (A));
      L : constant Float := Float (A'Length);
      begin
      S / L);

end Integer_Arrays;
```

### Listing 24: integer_arrays.adb

```ada
package body Integer_Arrays is

  function Sum (Arr : Integer_Array) return Integer is
  begin
    return Acc : Integer := 0 do
      for V of Arr loop
        Acc := Acc + V;
      end loop;
    end return;
  end Sum;

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;

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\(^\text{125}\)

8.4.1 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

pragmada2022;
package Integer_Arrays is
  type Integer_Array is
    array (Positive range <>) of Integer;

  type Integer_Sum is limited private;

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)
--
-- (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 : Integer := 0 do
        for V of Arr loop
            Acc := Acc + V;
        end loop;
    end return;
end Sum;

(continues on next page)
function Sum (Arr : Integer_Array) return Integer_Sum is
  (Integer_Sum (Integer'((Sum (Arr)))));
end Integer_Arrays;

8.5 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 \(\times\)) 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:

```ada
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];
end Show_Reduction_Expression;
```

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!)
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:

\[
I := 0; \\
\text{for } E \text{ of } A \text{ loop} \\
\quad I := I + E; \\
\text{end loop;}
\]

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;
```

In this example, we call A'Reduce (**, 1) to reduce the list. (Note that we use an
initial value of one because it is the identity element\textsuperscript{126} of a multiplication, so the complete operation is: $1 \times 2 \times 3 \times 4 = 24.$

In the Ada Reference Manual

- Reduction Expressions\textsuperscript{127}

### 8.5.1 Value sequences

In addition to arrays, we can apply reduction expression to value sequences, which consist of an iterated element association — for example, \texttt{[for I in 1..3 => 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 := [for I in 1..3 => I + 1]'Reduce ("+", 0);
    Put_Line ("I = 
               & I'Image);
    I := [for I in 1..3 => I + 1]'Reduce ("*", 1);
    Put_Line ("I = 
               & I'Image);
end Show_Reduction_Expression;
```

Code block metadata


MD5: e714f69700e3f03b7314ee0e531620c4

Runtime output

```
I =  9
I =  24
```

In this example, we create the value sequence \texttt{[for I in 1..3 => 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.)

\textsuperscript{126} https://en.wikipedia.org/wiki/Identity_element
\textsuperscript{127} http://www.ada-auth.org/standards/22rm/html/RM-4-5-10.html
8.5.2 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

Custom Reducer Procedure
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 `Integer` as the component type, while the accumulator type is `Long_Integer`. (For this kind of reducers, using `Long_Integer` instead of `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:

Listing 32: 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;

  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 `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'Reduce (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)
I := Accumulate (I, E);
end loop;

8.5.3 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:

8.5. Reduction Expressions
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**

```
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.
CHAPTER
NINE

STATEMENTS

9.1 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:

<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

9.2 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 1: 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

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 on 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

9.2.1 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 2: show_cleanup.adb

procedure Show_Cleanup is
  pragma Warnings (Off, "always false");
  Some_Error : Boolean;
begin
  Some_Error := False;
  if Some_Error then
    goto Cleanup;
  end if;
<<Cleanup>> null;
end Show_Cleanup;
```

Code block metadata

MD5: 0ce06582bbefae818d4da3b72d3436b

Here, we transfer the control to the cleanup statement as soon as an error is detected.

---

9.2.2 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 3: show_continue.adb

```
procedure Show_Continue is
  function Is_Further_Processing_Needed
    (Dummy : Integer)
  return Boolean
is
begin
  -- 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 Is_Further_Processing_Needed (E) then
      -- Do more stuff...
      null;
    end if;
  end loop;
end Show_Continue;
```

Code block metadata

MD5: 115eeaf08d5fb072d707d6325fe9cfd0

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:

Listing 4: show_continue.adb

```
procedure Show_Continue is
  function Is_Further_Processing_Needed
    (Dummy : Integer)
  return Boolean
is
begin
  -- 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 Is_Further_Processing_Needed (E) then
      goto continue;
    end if;
  end loop;
end Show_Continue;
```

(continues on next page)
if not Is_Further_Processing_Needed (E) then
goto Continue;
end if;
-- Do more stuff...
<<Continue>>
end loop;
end Show_Continue;

**Code block metadata**

MD5: 260b52ead782adf76e6e5cf3c4e8332b

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:

```ada
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:

```ada
loop
  -- Some statements...
  <<Continue>> null;
end loop;
```

### 9.2.3 Labels and compound statements

We can use labels with compound statements as well. For example, we can label a for loop:

```ada
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
```

(continues on next page)
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 6: show_statement_identifier.adb**

```ada
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;
```

**Code block metadata**

MD5: e52cb5e9a9427addf3cabe64dd73bc2d

**Runtime output**

Found: TRUE

Loop statement and block statement identifiers are generally preferred over labels. Later in this chapter, we discuss this topic in more detail.

9.2. Labels
9.3 Exit loop statement

We've introduced bare loops back in the Introduction to Ada course\(^\text{130}\). In this section, we'll briefly discuss loop names and exit loop statements.

A bare loop has this form:

```
loop
  exit when Some.Condition;
end loop;
```

We can name a loop by using a loop statement identifier:

```
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 7: 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;
  V  : constant Vector  :=  20 & 10 & 0 & 13;
  C  :    Cursor;
begin
  C := V.First;
  Put_Line ("Vector elements are: ");
  Show_Elements :
    loop
      exit Show_Elements when C = No_Element;
      Put_Line ("Element: 
       & Integer'Image (V (C)));
      C := Next (C);
    end loop Show_Elements;
  end Show_Vector_Cursor_Iteration;
```

**Code block metadata**

MD5: b77353f6ed98f8ddb32c73c47d249020

**Runtime output**

\(^{130}\) https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-bare-loops
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 8: show_inner_loop_exit.adb

```ada
procedure Show_Inner_Loop.Exit is
pragma Warnings (Off);
Cond : Boolean := True;
beg
Outer_Processing : loop
InInner_Processing : loop
exit Outer_Processing when Cond;
end loop Inner_Processing;
end loop Outer_Processing;
end Show_Inner_Loop.Exit;
```

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

9.4 If, case and loop statements

In the Introduction to Ada course, we talked about if statements, loop statements, and case statements. This is a very simple code example with these statements:

Listing 9: show_if_case_loop_statements.adb

```ada
procedure Show_If_Case_Loop_Statements is
pragma Warnings (Off);
Reset : Boolean := False;
```

https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-if-statement
https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-loop-statement
https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-case-statement
In this section, we'll look into a more advanced detail about the case statement.

In the Ada Reference Manual

- 5.3 If Statements
- 5.4 Case Statements
- 5.5 Loop Statements

9.4.1 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 additional to being a variable or a constant.

As we discussed earlier on (page 323), 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:

Listing 10: scales.ads

```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;
```

Listing 11: scales.adb

```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;
        when 9 .. 10 => Satisfaction := Very_Satisfied;
      end case;
    return Satisfaction;
  end To_Satisfaction_Scale;

end Scales;
```

Listing 12: show_case_statement_expression.adb

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

(continues on next page)
with Scales;  use Scales;

procedure Show_Case_Statement_Expression is
  Score : constant Scale := 0;
begin
  Put_Line ("Score: " & Scale'Image (Score)
 & Satisfaction_Scale'Image (To_Satisfaction_Scale (Score)));
end Show_Case_Statement_Expression;

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\(^{138}\), 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)` 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 11) Scale'Base'Range.

In other languages

In C, the switch-case statement requires parentheses for the choice expression:

\begin{verbatim}
#include <stdio.h>

int main(int argc, const char * argv[])
{
  int s = 0;
  switch (s)
  {
    case 0:
    case 1:
      printf("Value in the 0 -- 1 range\n");
    default:
      printf("Value > 1\n");
  }
}
\end{verbatim}

\(^{138}\)https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-case-statement
Runtime output

Value in the 0 -- 1 range
Value > 1

In Ada, parentheses aren't expected in the choice expression. Therefore, we shouldn't write `case (S) is` in a C-like fashion — unless, of course, we really want to evaluate an expression in the case statement.

### 9.5 Block Statements

We've introduced block statements back in the Introduction to Ada course[^139]. They have this simple form:

Listing 14: show_block_statement.adb

```ada
procedure Show_Block_Statement is
  pragmaWarnings (Off);
begin
  -- BLOCK STARTS HERE:
  declare
    I : Integer;
  begin
    I := 0;
  end;
end Show_Block_Statement;
```

Code block metadata

MD5: 61134b3899620c6d9ed68974fae33b5e

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 15: show_block_statement.adb

```ada
procedure Show_Block_Statement is
  pragmaWarnings (Off);
begin
  Simple_Block : declare
    I : Integer;
  begin
    I := 0;
  end Simple_Block;
```

[^139]: https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-block-statement
end Show_Block_Statement;

Code block metadata

MD5: b327b7675931d9b994637671c806c7c3

Note that we must write \texttt{end Simple Block}; when we use the \texttt{Simple Block} identifier. Block statement identifiers are useful:

\begin{itemize}
\item to indicate the begin and the end of a block — as some blocks might be long or nested in other blocks;
\item to indicate the purpose of the block (i.e. as code documentation).
\end{itemize}

In the Ada Reference Manual

- 5.6 Block Statements\textsuperscript{140}

9.6 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 16: 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

MD5: 16e85a8cb869802f912627c40a64c20

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:

\textsuperscript{140} \url{http://www.ada-auth.org/standards/22rm/html/RM-5-6.html}
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\(^{141}\)

---

9.6.1 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\(^{142}\).

While the extended_return_statement was added to the language specifically to support limited constructor functions (page 707), it comes in handy whenever you want a local name for the function result:

---

\(^{141}\) http://www.ada-auth.org/standards/22rm/html/RM-6-5.html

\(^{142}\) https://www.adacore.com/gems/ada-gem-10
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;
        end if;

        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;

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.
10.1 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\(^{143}\)
- 6.4.1 Parameter Associations\(^{144}\)

10.1.1 Formal Parameter Modes

We already discussed formal parameter modes in the *Introduction to Ada*\(^ {145} \) course:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>in</em></td>
<td>Parameter can only be read, not written</td>
</tr>
<tr>
<td><em>out</em></td>
<td>Parameter can be written to, then read</td>
</tr>
<tr>
<td><em>in out</em></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.

---

\(^{143}\) [http://www.ada-auth.org/standards/22rm/html/RM-6-2.html](http://www.ada-auth.org/standards/22rm/html/RM-6-2.html)

\(^{144}\) [http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html](http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html)

\(^{145}\) [https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#intro-ada-parameter-modes](https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#intro-ada-parameter-modes)
10.1.2 By-copy and by-reference

In the Introduction to Ada 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 1: check_param_passing.ads

```ada
with System;

procedure Check_Param_Passing
  (Formal : System.Address;
   Actual : System.Address);
```

Listing 2: 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 3: machine_x.ads

```ada
with System;

package Machine_X is

  procedure Update_Value
    (V     : in out Integer;
     AV    : System.Address);

end Machine_X;
```

Listing 4: machine_x.adb

```ada
with Check_Param_Passing;

package body Machine_X is

  procedure Update_Value
    (V     : in out Integer;
     AV    : System.Address) is
  (continues on next page)
```

---

146 https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#intro-ada-parameter-modes
begin
    V := V + 1;
    Check_Param_Passing (Formal => V'Address,
                         Actual => AV);
end Update_Value;
end Machine_X;

Listing 5: 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
    Update_Value (A, A'Address);
end Show_By_Copy_By_Ref_Params;

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 132)).

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;

10.1. Parameter Modes and Associations
• 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:

<table>
<thead>
<tr>
<th>Type category</th>
<th>Parameter passing</th>
<th>List of types</th>
</tr>
</thead>
<tbody>
<tr>
<td>By copy</td>
<td>By copy</td>
<td>• Elementary types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Descendant of a private type whose full type is a by-copy type</td>
</tr>
<tr>
<td>By reference</td>
<td>By reference</td>
<td>• Tagged types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Task and protected types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Explicitly limited record types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Composite types with at least one subcomponent of a by-reference type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Private types whose full type is a by-reference type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Any descendant of the types mentioned above</td>
</tr>
<tr>
<td>Unspecified</td>
<td>Either by copy or by reference</td>
<td>• Any type not mentioned above</td>
</tr>
</tbody>
</table>

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 716).

Let's see an example:

Listing 6: 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 Machine_X;
```

(continues on next page)
end record;

procedure Update_Value
(R : in out Rec;
 AR : System.Address);

procedure Update_Value
(RA : in out Rec_Array;
 ARA : System.Address);

procedure Update_Value
(R : in out Tagged_Rec;
 AR : System.Address);

end Machine_X;

Listing 7: machine_x.adb

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 8: show_by_copy_by_ref_params.adb

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)
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
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**

- **Project**: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_Associations.By_Copy_By_Ref_Params
- **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.)
10.1.3 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 9: 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 10: 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...");
  end Update_Value;
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

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 11: show_by_copy_by_ref_params.adb

with Ada.Text_IO; use Ada.Text_IO;
with Machine_X; use Machine_X;

procedure Show_By_Copy_By_Ref_Params is
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

MD5: 96be7054b7ff6a304705edf6b15f031

Runtime output

Calling Update_Value...
Formal parameter at 00007FC5B7D4A9C
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.
10.1.4 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 523) and aliased parameters (page 532) later on.)

Let's rewrite a previous code example that has a parameter of elementary type and change it to aliased:

**Listing 12: 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 13: machine_x.adb**
```ada
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 14: show_by_copy_by_ref_params.adb**
```ada
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: c066af3a7d081815d0a7598733f9e6aee

**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 15: 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'Address);
end Show_By_Copy_By_Ref_Params;
```

Again, we discuss more about aliased parameters (page 532) and aliased objects (page 525) later on in the context of access types.

### 10.1.5 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 16: operations.ads

```ada
package Operations is

  procedure Add (Left : in out Integer;
                 Right : Float := 1.0);

end Operations;
```

147 [http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html](http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html)
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:

```ada
package Operations is
  procedure Add (Left : in out Integer);
  procedure Add (Left : in out Integer;
                 Right : Float := 1.0);
end Operations;
```

```ada
package body Operations is
  procedure Add (Left : in out Integer) is
    begin
      Left := Left + 1;
    end Add;
  procedure Add (Left : in out Integer;
                 Right : Float := 1.0) is
    begin
      Left := Left + Integer (Right);
    end Add;
end Operations;
```

```ada
with Operations; use Operations;
procedure Show_Param_Association is
  A : Integer := 5;
begin
  Add (A);
  -- ERROR: cannot decide which
  -- procedure to take
end Show_Param_Association;
```

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 \texttt{Add} with one parameter

or

- the second instance of \texttt{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:

- we could just rename one of \texttt{Add} procedures (thereby eliminating the subprogram overloading);

- we could rename the first parameter of one of the \texttt{Add} procedures and use named parameter association in the call to the procedure;
  
  - For example, we could rename the parameter to \texttt{Value} and call \texttt{Add (Value \Rightarrow A)}.

- remove the default value from the second parameter of the second instance of \texttt{Add}.

### Overlapping actual parameters

When we have more than one \texttt{out} or \texttt{in out} parameters in a subprogram, we might run into the situation where the actual parameter overlaps with another parameter. For example:

```ada
package Machine_X is

  procedure Update_Value (V1 : in out Integer;
                          V2 :   out Integer);

end Machine_X;

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;

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;
```

### Code block metadata

10.1. Parameter Modes and Associations

369
In this case, we're using A for both output parameters in the call to `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.

### 10.2 Operators

Operators are commonly used for variables of scalar types such as `Integer` and `Float`. In these cases, they replace `usual` function calls. (To be more precise, operators are function calls, but written in a different format.) For example, we simply write

```ada
A := A + B + C;
```

when we want to add three integer variables. A hypothetical, non-intuitive version of this operation could be

```ada
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><code>and</code>, <code>or</code>, <code>xor</code></td>
</tr>
<tr>
<td>Relational</td>
<td><code>=</code>, <code>/=</code> <code>&lt;=</code>, <code>&gt;=</code></td>
</tr>
<tr>
<td>Unary adding</td>
<td><code>+</code>, <code>-</code></td>
</tr>
<tr>
<td>Binary adding</td>
<td><code>+</code>, <code>-</code>, <code>&amp;</code></td>
</tr>
<tr>
<td>Multiplying</td>
<td><code>*</code>, <code>/</code>, <code>mod</code>, <code>rem</code></td>
</tr>
<tr>
<td>Highest precedence</td>
<td><code>**</code>, <code>abs</code>, <code>not</code></td>
</tr>
</tbody>
</table>

In the Ada Reference Manual

- 4.5 Operators and Expression Evaluation

#### 10.2.1 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 (`=` and `/=`), which are defined for any type (be it scalar, record types, and array types).

For array types, the concatenation operator (`&`) is a primitive operator:

---

package Integer_Arrays is

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

end Integer_Arrays;

with Ada.Text_IO;   use Ada.Text_IO;
with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Concatenation is
   A, B : Integer_Array (1 .. 5);
   R : Integer_Array (1 .. 10);
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_Concatenation;

Code block metadata

MD5: 1899e66ec1d0b36b10d8b89fc2dfac0e

Runtime output

1 2 3 4 5 6 7 8 9 10

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:

package Integer_Arrays is

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

end Integer_Arrays;

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);

   (continues on next page)
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: d94f9791523359d390a7cafd900d1268

Build output

show_array_addition.adb:9:11: error: there is no applicable operator "+" for type "Integer_Array" defined at integer_arrays.ads:3
gprbuild: *** compilation phase failed

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:

\[
(\|) :: \text{Int} \to \text{Int} \to \text{Int}
\]

\[a \| b = a + a + b\]

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:

```
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;
```
Listing 30: integer_arrays.adb

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;
      return R;
    end "+";
end Integer_Arrays;

Listing 31: 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: 6f50fa47270d973fb50379b6275777d

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 328).)

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.)

10.2. Operators 373
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:

Listing 32: addresses.ads

```ada
package Addresses is

   type Person is private;

   function "+" (Name : String;
                    Address : String)
       return Person;

   function "+" (Left, Right : Person)
       return Person;

   procedure Display (P : Person);

private

   subtype Name_String is String (1 .. 40);       
   subtype Address_String is String (1 .. 100);   

   type Person is record
      Name : Name_String;
      Address : Address_String;
   end record;

end Addresses;
```

Listing 33: 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;
```

(continues on next page)
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:
8 "Jane" + "7 High Street";
9 begin
10 if Jane + John = John + Jane then
11 Put_Line ("It's commutative!");
12 else
13 Put_Line ("It's not commutative!");
14 end if;
15 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\(^{149}\)

### 10.3 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:

Listing 36: expr_func.ads

```
package Expr_Func is
  function Is_Zero (I : Integer) return Boolean is
    (I = 0);
end Expr_Func;
```

\(^{149}\) [http://www.ada-auth.org/standards/22rm/html/RM-6-1.html](http://www.ada-auth.org/standards/22rm/html/RM-6-1.html)
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:

**Listing 37: expr_func.ads**

```ada
package Expr_Func is

  function Is_Zero (I : Integer) return Boolean;

end Expr_Func;
```

**Listing 38: expr_func.adb**

```ada
package body Expr_Func is

  function Is_Zero (I : Integer) return Boolean is
    begin
      return I = 0;
    end Is_Zero;

end Expr_Func;
```

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:

**Listing 39: expr_func.ads**

```ada
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:

Listing 40: expr_func.ads

```ada
package Expr_Func is

   function Is_Zero (I : Integer) return Boolean;

end Expr_Func;
```

Listing 41: expr_func.adb

```ada
package body Expr_Func is

   function Is_Zero (I : Integer)
      return Boolean is
      (I = 0);

end Expr_Func;
```

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:

Listing 42: my_data.ads

```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;
```

Naturally, we could write the function implementation in the package body instead:
Listing 43: my_data.ads

```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;

end My_Data;
```

Listing 44: my_data.adb

```ada
package body My_Data is

  function Is_Valid (D : Data) return Boolean is
    (D.Valid);

end My_Data;
```

**Code block metadata**

- Private_Expression_Function_2
- MD5: 3c6e2a3c53c7c8e1a7b86efccdc3bf8d

**In the Ada Reference Manual**

- 6.8 Expression functions

## 10.4 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:


151 [https://www.adacore.com/gems/gem-50](https://www.adacore.com/gems/gem-50)
Listing 45: show_overloading.adb

```ada
procedure Show_Overloading is

   package Types is
      type Sequence is null record;
      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;
```

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 46: show_top_down_overloading.adb

```ada
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: 020c4f04285c80c1050d8edbabf2dbca
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 information.

If we overload things too heavily, we can cause ambiguities:

```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;
begnin
    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:

```ada
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 el) 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 48: 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 49: 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;
```

You can't tell by looking at the literal 10_000 what its type is. The type of formal parameter Count tells us that 10_000 is an Orange_Count, as opposed to some other type, such as Standard.Long_Integer.

Technically, the type of 10_000 is universal_integer, which is implicitly converted to Orange_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:
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:

```
Slice (Count => 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.

```
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 Grind procedures visible, so the type of the aggregate could be Complex or String, so it is ambiguous and therefore illegal. The compiler is not required to notice that there is only one type with components Re and Im, of some real type — in fact, the compiler is not allowed to notice that, for overloading purposes.

We can qualify as usual:

```
Grind (X => Complex'(Re => 1.0, Im => 1.0));
```
Only after resolving that the type of the aggregate is Complex can the compiler look inside and make sure Re and 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.

### 10.5 Operator Overloading

We've seen previously (page 370) that we can define custom operators for any type. We've also seen that subprograms can be overloaded (page 379). 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\(^\text{152}\)
- G.1.1 Complex Types\(^\text{153}\)

### 10.6 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:

\(^{152}\text{http://www.ada-auth.org/standards/22rm/html/RM-6-6.html}\)
\(^{153}\text{http://www.ada-auth.org/standards/22rm/html/RM-G-1-1.html}\)
### Listing 51: fixed_point.ads

```ada
package Fixed_Point is

  D : constant := 2.0 ** (-31);
  type TQ31 is delta D range -1.0 .. 1.0 - D;

end Fixed_Point;
```

### Listing 52: show_sat_op.adb

```ada
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 53: fixed_point.ads

```ada
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;
```

---

10.6. Operator Overriding 385
package body Fixed_Point is

function "+" (Left, Right : TQ31) return TQ31 is

  type TQ31_2 is
    delta TQ31'Delta
    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;

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;
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.

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).

### 10.7 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;

package body Servers is
    procedure Run_Server is
        begin
           pragma Warnings
            (Off,
                "implied return after this statement");
            while True loop
                null;
            end loop;
        end Run_Server;
end Servers;

with Servers; use Servers;
procedure Show_Endless_Loop is
    begin
        (continues on next page)
```
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:

Listing 59: loggers.ads

```ada
package Loggers is

  Logged_Failure : exception;
  procedure Log_And_Raise (Msg : String)
    with No_Return;
end Loggers;
```

Listing 60: 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);
      raise Logged_Failure;
    end Log_And_Raise;
end Loggers;
```

Listing 61: 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;
```

In this example, Run_Server doesn't exit from the while True loop, so it never returns to the Show_Endless_Loop procedure.
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:

```ada
package Loggers is
  type Logged_Failure is exception;
  procedure Log_And_Raise (Msg : String)
    with No_Return;
end Loggers;
```

```ada
package body Loggers is
  procedure Log_And_Raise (Msg : String) is
  begin
    Put_Line (Msg);
  end Log_And_Raise;
end Loggers;
```

```ada
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**


**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

- 6.5.1 Nonreturning Subprograms\(^{154}\)

10.8 Inline subprograms

Inlining\(^{155}\) 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 65: 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;
```

\(^{154}\) http://www.ada-auth.org/standards/22rm/html/RM-6-5-1.html

\(^{155}\) https://en.wikipedia.org/wiki/Inline_expansion
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;

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 = ");
        Put_Line ("Average = ");
        & Float'Image (Average_Value));
    end Compute_Average;

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.

10.8. Inline subprograms
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

- 6.3.2 Inline Expansion of Subprograms\textsuperscript{156}

### 10.9 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 \texttt{is null} in its declaration. For example:

```
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:

```
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 \texttt{null}. Therefore, the Do_Nothing procedure above is equivalent to this:

\textsuperscript{156} http://www.ada-auth.org/standards/22rm/html/RM-6-3-2.html
Listing 70: null_procs.ads

```ada
package Null_procs is
    procedure Do_Nothing (Msg : String);
end Null_procs;
```

Listing 71: null_procs.adb

```ada
package body Null_procs is
    procedure Do_Nothing (Msg : String) is
        begin
            null;
        end Do_Nothing;
end Null_procs;
```

**Code block metadata**

MD5: d0c9dc9265ebbaa9603681182dee1d92

### 10.9.1 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:

Listing 72: 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;
end Simple_Storage;
```

**Code block metadata**

MD5: 553e78bc15dcec1302be4b5f484ac21f

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:

Listing 73: show_null_proc.adb

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

procedure Show_Nul_Proc is
```

(continues on next page)
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:

Listing 74: 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 75: 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;
```
Listing 76: 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;
```

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 151) contains an extended example that makes use of null procedures.

In the Ada Reference Manual

- 6.7 Null Procedures

11.1 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:

Listing 1: show pragma assert.adb

```ada
procedure Show_Pragma_Assert is
  I : constant Integer := 10;
pragma Assert (I = 10); begin null; 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:

Listing 2: show pragma assert.adb

```ada
procedure Show_Pragma_Assert is
  I : constant Integer := 11;
pragma Assert (I = 10, "I is not 10"); begin null; end Show_Pragma_Assert;
```

**Build output**
show pragma assert.adb:4:19: warning: assertion will fail at run time [-gnatw.a]

Runtime output

raised ADAASSERTIONSASSERTION_ERROR: I is not 10

Similarly, we can use the procedural form of Assert. For example, the code above can implemented as follows:

Listing 3: show procedure assert.adb

```ada
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: cbab23645ff89d4adffcaaddae6f0e3

Runtime output

raised ADAASSERTIONSASSERTION_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 4: show assertion error.adb

```ada
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

MD5: 9c846acf998ca7adabd47c3b5a6ce39f

Runtime output

raised ADAASSERTIONSASSERTION_ERROR: I is not 10

In the Ada Reference Manual

398 Chapter 11. Exceptions
11.2 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:

Listing 5: show_pragma_assertion_policy.adb

```ada
procedure Show_Pragma_Assertion_Policy is
  I : constant Integer := 11;
pragma Assertion_Policy (Assert => Ignore);
beg
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:

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert</td>
<td>Check assertions</td>
</tr>
<tr>
<td>Static_Predicate</td>
<td>Check static predicates</td>
</tr>
<tr>
<td>Dynamic_Predicate</td>
<td>Check dynamic predicates</td>
</tr>
<tr>
<td>Pre</td>
<td>Check pre-conditions</td>
</tr>
<tr>
<td>Pre'Class</td>
<td>Check pre-condition of classes of tagged types</td>
</tr>
<tr>
<td>Post</td>
<td>Check post-conditions</td>
</tr>
<tr>
<td>Post'Class</td>
<td>Check post-condition of classes of tagged types</td>
</tr>
<tr>
<td>Type_Invariant</td>
<td>Check type invariants</td>
</tr>
<tr>
<td>Type_Invariant'Class</td>
<td>Check type invariant of classes of tagged types</td>
</tr>
</tbody>
</table>

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:

158 http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html
• 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\textsuperscript{159}.

You can specify multiple policies in a single call to Assertion_Policy. For example, you can activate all policies by writing:

Listing 6: 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;
```

\textbf{Code block metadata}

\begin{itemize}
  \item MD5: 3abbf97160b755b84cc4f7e652ca5551
\end{itemize}

\textsuperscript{159} https://gcc.gnu.org/onlinedocs/gnat_rm/Pragma-Assertion\_005fPolicy.html

400 Chapter 11. Exceptions
In the GNAT toolchain

With GNAT, policies can be specified in multiple ways. In addition to calls to Assert_Policy, you can use configuration pragmas files. You can use these files to specify all pragmas that are relevant to your application, including Assert_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 (Assert_Policy (Assert => Ignore)). For example:

Listing 7: 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 Assert_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;
```

Code block metadata

MD5: 7be3bab24d856081afeddabe40afc84f

Build output

show_assert_procedure_policy.adb:10:19: warning: assertion would fail at run time
..[-gnatw.a]

Runtime output

----- Pragma Assert -----
----- Procedure Assert ----

raised ADA.ASSERTIONS.ASSERTION_ERROR : a-assert.adb:42

Here, the `pragma Assert` is ignored due to the assertion policy. However, the call to Assert is not ignored.

In the Ada Reference Manual

- [11.4.2 Pragmas Assert and Assert_Policy](http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html)

---

160 [https://gcc.gnu.org/onlinedocs/gnat_ugn/The-Configuration-Pragmas-Files.html#The-Configuration-Pragmas-Files](https://gcc.gnu.org/onlinedocs/gnat_ugn/The-Configuration-Pragmas-Files.html#The-Configuration-Pragmas-Files)

11.3 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.

11.3.1 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:

Listing 8: show_access_check.adb

```ada
procedure Show_Access_Check is
    type Integer_Access is access Integer;
    AI : Integer_Access;
begin
    AI.all := 10;
end Show_Access_Check;
```

Code block metadata

MD5: 4db8b63efb23caa7da926d4ec9f204bf

Build output

show_access_check.adb:5:04: warning: variable "AI" is read but never assigned [-gnatwv]
show_access_check.adb:7:04: warning: null value not allowed here [enabled by default]
show_access_check.adb:7:04: warning: Constraint_Error will be raised at run time [enabled by default]

Runtime output

402 Chapter 11. Exceptions
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 554).) For example:

```
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;
```

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).
11.3.2 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 10: show_discriminant_check.adb

```ada
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;
```

Code block metadata

Discriminant_Check
MD5: 665ab37962f8f9c129acac543b1eb15d

Build output

show_discriminant_check.adb:14:09: warning: incorrect value for discriminant "Valid"
show_discriminant_check.adb:14:09: warning: Constraint_Error will be raised at run time

Runtime output

raised CONSTRAINT_ERROR : show_discriminant_check.adb:14 discriminant check failed

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:

Listing 11: show_discriminant_check.adb

```ada
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;
```

(continues on next page)
11.3.3 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:

```
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;
```

Here, R's discriminant (Valid) is False, so we cannot access the Counter component, for it only exists when the Valid discriminant is True.
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
I := Mod_Op (10, 0);
end Show_Division_Check;
```

**Code block metadata**


MD5: 6ec0856be947eea6610cffaa0e875d45

**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.

### 11.3.4 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:

```ada
procedure Show_Index_Check is
  type Integer_Array is
    array (Positive range <>) of Integer;

  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 : ops.ads:4 divide by zero
```

406 Chapter 11. Exceptions
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.

### 11.3.5 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 15: show_length_check.adb

```ada
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;

  begin
    Arr_1 : Integer_Array (1 .. 10);
    Arr_2 : Integer_Array (1 .. 9) :=
             (others => 1);
    Assign (Arr_1, Arr_2);
  end Show_Length_Check;
```

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.

### 11.3.6 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 16: show_overflow_check.adb

```ada
procedure Show_Overflow_Check is
  A, B : Integer;
  begin
    (continues on next page)
```
A := Integer'Last;
B := 1;
A := A + B;
end Show_Overflow_Check;

11.3.7 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 17: show_range_check.adb

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;

In this example, I already has the last possible value of the Integer'Base range, so increasing it by one causes an overflow error.
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.

### 11.3.8 Tag Check

The tag check ensures that the tag of a tagged object matches the expected tag in a dispatching operation. For example:

```
Listing 18: p.ads

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 19: show_tag_check.adb

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;
```

**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.

### 11.3. Checks and exceptions
Since the tags don't match, the tag check fails in the assignment of A1 to A2.

### 11.3.9 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 534).

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:

```
package P is
  type T is tagged null record;
  type T_Class is access all T'Class;
end P;
```

```
with P; use P;

procedure Show_Accessibility_Check is
  A1 : access T'Class := new T;
  A2 : T_Class;
begin
  A2 := T_Class (A1);
end Show_Accessibility_Check;
```

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:
### 11.3. Checks and exceptions

#### 11.3.10 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\(^\text{162}\):

```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;

   (continues on next page)
```

---

advanced journey with ada: a flight in progress

15 type T2_Ref is access T2;
16   procedure Free is new
17     Ada.UncheckedDeallocation (T2, T2_Ref);
18 end P;

Listing 24: p.adb

1 with Ada.Text_IO; use Ada.Text_IO;
2 3 package body P is
4 5 procedure Finalize (X : in out T1) is
6     X2 : T2_Ref := new T2;
7     begin
8       Put_Line ("Finalizing T1...");
9       Free (X2);
10      end Finalize;
11     procedure Finalize (X : in out T2) is
12      begin
13        Put_Line ("Finalizing T2...");
14      end Finalize;
15 end P;

Listing 25: show_allocation_check.adb

1 with P; use P;
2 3 procedure Show_Allocation_Check is
4     X2 : T2_Ref := new T2;
5     begin
6       Free (X2);
7     end Show_Allocation_Check;

Code block metadata

Allocation_Check
MD5: 915e8ab21e550c981503c014bcceade1

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.
11.3.11 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 AI-00064:\textsuperscript{163}:

```
Listing 26: p.ads

function P return Integer;
```

```
Listing 27: p.adb

function P return Integer is
begin
  return 1;
end P;
```

```
Listing 28: 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;
begin
  null;
end Show_Elaboration_Check;
```

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

\textsuperscript{163} http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00064.txt?rev=1.12&raw=N
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.

### 11.3.12 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 85):

```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
with Ada.Text_IO; use Ada.Text_IO;
with Custom_Types; use Custom_Types;
procedure Show_Storage_Check is
    RAV1, RAV2 : UInt_7_Reserved_Access;
begin
    Put_Line ("Allocating RAV1...");
    RAV1 := new Uint_7;
    Put_Line ("Allocating RAV2...");
    RAV2 := new Uint_7;
    New_Line;
end Show_Storage_Check;
```

On each allocation (new Uint_7), a storage check is performed. Because there isn't enough reserved storage space before the second allocation, the checks fails and raises a Storage_Error exception.
### 11.4 Ada.Exceptions package

**Note:** Parts of this section were originally published as Gem #142 : Exception-ally

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:

```ada
raise Constraint_Error with "some message";
```

**Historically**

Since Ada 2005, we can use the `raise Constraint_Error with "some message"` syntax. In Ada 95, you had to call the `Raise_Exception` procedure:

```ada
Ada.Exceptions.Raise_Exception -- Ada 95
  (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.

**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

### 11.4.1 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));
```

---

165 https://www.adacore.com/gems/gem-142-exceptions
166 http://www.ada-auth.org/standards/22rm/html/RM-11-4-1.html
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:

```ada
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 "We got a problem";
    end Nested;

  begin
    Nested;

  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;
```

11.4.2 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:

```ada
with Ada.Exceptions; use Ada.Exceptions;

package Exception Tests is
```

(continues on next page)
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;

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);
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;

(continues on next page)
begin
    Last_Occur := 0;
    Nested_1;
    Nested_2;
end Test_Exceptions;

end Exception_Tests;

Listing 34: 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

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 35: exception_occurrence_stream.adb

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;
    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;
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.

11.4.3 Debugging exceptions in the GNAT toolchain

Here is a typical exception handler that catches all unexpected exceptions in the application:

Listing 36: main.adb

```ada
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;
    exception
        when E : others =>
        Put_Line
        (Ada.Exceptions.Exception_Information (E));
    end Main;
```

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):

```bash
> rm *.o # Cleanup previous compilation
> gnatmake -g main.adb
> gdb ./main
(gdb) catch exception
(continues on next page)
```
(gdb) run
Catchpoint 1, CONSTRAINT_ERROR at 0x0000000000402860 in main.nested () at main.
- addr:8
                    raise Constraint_Error with "some message";
(gdb) bt
#0 <__gnat_debug_raise_exception> (e=0x62ec40 <constraint_error>) at s-excdeb.
- addr: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:

> addr2line -e main --functions --demangle 0x10b7e24d1 0x10b7e24ee 0x10b7e2472
/path/main.adb:8
    ada_main
/path/main.adb:12
    main
/path/b-main.adb:240

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

Exception name: CONSTRAINT_ERROR
Message: some message
Call stack traceback locations:
0x1000014d1 0x1000014ee 0x100001472
> atos -o main 0x10b7e24d1 0x10b7e24ee 0x10b7e2472
main__nested.2550 (in main) (main.adb:8)
    _ada_main (in main) (main.adb:12)
    main (in main) + 90

11.4. Ada.Exceptions package
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:

Listing 37: main.adb

```ada
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)');
    end Nested;

    begin
      Nested (3);
      exception
        when E : others =>
          Put_Line
          (Ada.Exceptions.Exception_Information (E));
    end Main;
```

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:

```
```

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.
11.5 Exception renaming

We can rename exceptions by using an exception renaming declaration in this form:

Renamed_Exception : exception renames Existing_Exception;  

For example:

Listing 38: show_exception_renaming.adb

```ada
procedure Show_Exception_Renaming is
  CE : exception renames Constraint_Error;
begin
  raise CE;
end Show_Exception_Renaming;
```

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:

Listing 39: internal_exceptions.ads

```ada
package Internal_Exceptions is
  Int_E : exception;
end Internal_Exceptions;
```

Listing 40: test_constraints.ads

```ada
with Internal_Exceptions;
package Test_Constraints is
  Ext_E : exception renames Internal_Exceptions.Int_E;
end Test_Constraints;
```

Listing 41: 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;
```
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 withing 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

11.6 Out and Uninitialized

Note: This section was originally written by Robert Dewar and published as Gem #150: Out and Uninitialized

Perhaps surprisingly, the Ada standard indicates cases where objects passed to out and in out parameters might not be updated when a procedure terminates due to an exception. Let's take an example:

Listing 42: 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;
```

168 https://www.adacore.com/gems/gem-150out-and-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 out and 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 out or 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:

```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;

  begin
    B : Integer := 0;
    Local (B);
  exception
    when others =>
      Put_Line ("Value for B is" & Integer'Image (B));
  end Show_Out_Uninitialized_Warnings;
```

Show that if we simplify the above code to:

```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;

  begin
    B : Integer := 0;
    Local (B);
  exception
    when others =>
      Put_Line ("Value for B is" & Integer'Image (B));
  end Show_Out_Uninitialized_Warnings;
```

In the GNAT toolchain

In general, any code that reads the actual object passed to an out or 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:

```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;

  begin
    B : Integer := 0;
    Local (B);
  exception
    when others =>
      Put_Line ("Value for B is" & Integer'Image (B));
  end Show_Out_Uninitialized_Warnings;
```

In the GNAT toolchain

In general, any code that reads the actual object passed to an out or 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:

```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;

  begin
    B : Integer := 0;
    Local (B);
  exception
    when others =>
      Put_Line ("Value for B is" & Integer'Image (B));
  end Show_Out_Uninitialized_Warnings;
```

In the GNAT toolchain

In general, any code that reads the actual object passed to an out or 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:

```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;

  begin
    B : Integer := 0;
    Local (B);
  exception
    when others =>
      Put_Line ("Value for B is" & Integer'Image (B));
  end Show_Out_Uninitialized_Warnings;
```
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 44: show_out_initialized_rec.adb

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

procedure Show_Out_Initialized_Rec is
  type Rec is tagged record
    Field : Integer;
  end record;

  procedure Local (A : in out Rec) is
  begin
    A.Field := 1;
    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); -- "1"
end Show_Out_Initialized_Rec;
```

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:
package Exported_Procedures is

procedure Local (A : in out Integer;
    Error : Boolean);
pragma Export_Procedure
    (Local,
    Mechanism => (A => Reference));
end Exported_Procedures;

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;

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;

Code block metadata

MD5: aed2788be2b3ceeeec19b28421c53fc66

Runtime output

Value for B is 1

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\(^{169}\)). For instance, the following code makes an incorrect assumption:

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

### 11.7 Suppressing checks

#### 11.7.1 pragma Suppress

**Note:** This section was originally written by Gary Dismukes and published as Gem #63: The Effect of Pragma 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:

```ada
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;
```

---

[^171]: https://www.adacore.com/gems/gem-63
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.

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.

### 11.7.2 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:

```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);
  A_2 : Half_Integer_Array := (others => 0);
  begin
    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);
  A_2 : Half_Integer_Array := (others => 0);
```

(continues on next page)
begin
  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

MD5: 0585b78fd57913d3172c7ablea6f4864

Runtime output

raised CONSTRAINT_ERROR : show_index_check.adb:39 index check failed

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

- 11.5 Suppressing Checks\textsuperscript{172}

\textsuperscript{172} [http://www.ada-auth.org/standards/22rm/html/RM-11-5.html]
Part III

Modular programming
12.1 Package renaming

We’ve seen in the Introduction to Ada course that we can rename packages\textsuperscript{173}.

In the Ada Reference Manual

• 10.1.1 Compilation Units - Library Units\textsuperscript{174}

12.1.1 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

```
package Driver_M1 is
  end Driver_M1;
```

Listing 2: driver_m2.ads

```
package Driver_M2 is
  end Driver_M2;
```

Listing 3: drivers.ads

```
package Drivers
  with Pure is
  end Drivers;
```

Listing 4: drivers-m1.ads

```
with Driver_M1;
package Drivers_M1 renames Driver_M1;
```

\textsuperscript{173} https://learn.adacore.com/courses/intro-to-ada/chapters/modular_programming.html#intro-ada-package-renaming
\textsuperscript{174} 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_Prog.Packages.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_Prog.Packages.Package_Renaming.Package_Renaming_1 Refer To Parent
MD5: d174746d8151d9a2cd48ad44e853850

**Build output**

```
driver_m1.ads:3:27: error: "Drivers" is undefined
```

As expected, compilation fails here because Drivers.Counter isn't visible in Driver_M1, even though the renaming (Drivers.M1) creates a virtual hierarchy.
12.1.2 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

```ada
-- A package called Drivers.M1.Ext is automatically available!
with Drivers.M1.Ext;
procedure Dummy is
  begin
    null;
  end Dummy;
```

Listing 10: dummy.adb

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.)

12.1.3 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.

12.2 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\textsuperscript{175}, 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

\textsuperscript{175} https://learn.adacore.com/courses/intro-to-ada/chapters/privacy.html#intro-ada-course-privacy
private part of a package to distinguish between the partial and full view (page 35) 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.

### 12.2.1 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;
```

(continues on next page)
end Data_Processing;

Listing 19: data_processing-calculations.ads

private package Data_Processing.Calculations is
  procedure Calculate (D : in out Data);
end Data_Processing.Calculations;

Listing 20: data_processing.adb

with Data_Processing.Calculations;
use Data_Processing.Calculations;

package body Data_Processing is
  procedure Process (D : in out Data) is
    begin
      Calculate (D);
    end Process;
end Data_Processing;

Listing 21: data_processing-calculations.adb

package body Data_Processing.Calculations is
  procedure Calculate (D : in out Data) is
    begin
      -- Dummy implementation...
      null;
    end Calculate;
end Data_Processing.Calculations;

Listing 22: 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

Package
MD5: 3edd5f73938e809994347b5876014d0d

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.
12.2.2 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
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: 32fc76ae13f1eecd854a029793034d8

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: 27fd3b063a1led7797cc44fae8349

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;
end Data_Processing.Advanced_Calculations;

Listing 31: data_processing-advanced_calculations.adb

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;

12.2.3 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

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;

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.

12.2. Private packages
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:

Listing 33: data_processing-calculations-child.ads
1 package Data_Processing.Calculations.Child is
2 procedure Process (D : in out Data);
3 end Data_Processing.Calculations.Child;

Listing 34: data_processing-calculations-child.adb
1 package body Data_Processing.Calculations.Child is
2 procedure Process (D : in out Data) is
3 begin
4 Calculate (D);
5 end Process;
6 end Data_Processing.Calculations.Child;

Listing 35: test_data_processing.adb
1 with Data_Processing; use Data_Processing;
3 procedure Test_Data_Processing is
4 D : Data;
5 begin
6 Calculate (D);
7 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 454).

Note that subprograms can also be declared private. We'll see this in another section (page 473).
Important

We’ve discussed package renaming in a previous section (page 435). 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;
```

Code block metadata

MD5: c03584dc26abb108c9c04074234b9637

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;
```

Code block metadata

MD5: 55415978604lcea4eaaeb02df13cd2f4

Build output

```
test_driver.adb:2:06: error: unit in with clause is private child unit
test_driver.adb:2:06: error: current unit must also have parent "Drivers"
gprbuild: *** compilation phase failed
```

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

12.2. Private packages 445
12.3 Private with clauses

12.3.1 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`:

```ada
package P is
  type T is null record;
end P;
```

```ada
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;
```

```ada
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:

private with P;

package Data is

-- ERROR: information from P
-- isn't visible here

type T2 is new P.T;

end Data;

package P is

    type T is null record;

    procedure Process (A : T) is null;

end P;

Listing 43: data.ads

Listing 44: p.ads

Listing 45: data.ads

Listing 46: data.adb

---

**Code block metadata**

**Project:** Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Simple_Private_With_Clause

**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:

```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;

package body Data is

    procedure Process (A : T2) is

(continues on next page)

12.3. Private with clauses
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;

12.3.2 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;
end P.Private_Child;

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;
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:

```
with P.Public_Child; procedure Test_Parent_Child is
   A : T2;
begin
   null;
end Test_Parent_Child;
```

In the Ada Reference Manual

- 10.1.2 Context Clauses - With Clauses\(^\text{177}\)

\(^{177}\) http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html
12.4 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

```ada
with B; use B;

package A is
  type T1 is record
    Value : T2;
  end record;
end A;
```

Listing 54: b.ads

```ada
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 149) 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 34). For the specific case of the previous source-code example, this would be the limited visibility to package B from package A's perspective:

```ada
package B is
  -- Incomplete type
  type T2;
```

(continues on next page)
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 149).

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:

```ada
limited with B;

package A is

  type T1 is record
    Ref : access B.T2;
  end record;

end A;
```

```ada
with A; use A;

package B is

  type T2 is record
    Value : T1;
  end record;

end B;
```

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:

```ada
limited with B;

package A is

  type T1 is record
    Ref : access B.T2;
  end record;

end A;
```

**12.4. Limited Visibility**
Now, both packages \( A \) and \( B \) have limited visibility to each other.

In the Ada Reference Manual

- 10.1.2 Context Clauses - With Clauses

12.4.1 Limited visibility and private with clauses

We can limit the visibility and use \textit{private with clauses} (page 446) at the same time. For a package \( P \), we do this by simply writing \textit{limited private with} \( P \).

Let's reuse the previous source-code example and convert types \( T1 \) and \( T2 \) to private types:

178 \[\text{http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html}\]
type T2 is private;
private
  use A;
  -- Here, we have full visibility
  -- of package A
  type T2 is record
    Value : T1;
  end record;
end B;

12.4.2 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 A;

Code block metadata
MD5: 61ecb5dc2617ecac62a05d7d2c6c0df

Build output
a.ads:10:26: error: "Zero_Const" not declared in "Info"
a.ads:11:26: error: "Zero_Func" not declared in "Info"
gprbuild: *** compilation phase failed

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.)

12.5 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.

12.5.1 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.

### 12.5.2 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.

12.5.3 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 \texttt{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.
12.5.4 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.

12.6 Use type clause

Back in the Introduction to Ada course, 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

```ada
with Ada.Text_IO;

procedure Display_Message is
begin
    (continues on next page)
```

179 https://learn.adacore.com/courses/intro-to-ada/chapters/modular_programming.html#intro-ada-use-clause
Ada.Text_IO.Put_Line ("Hello World!");

```ada
end Display_Message;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.No_Use_Clause
MD5: 4c6ff19809c13ebd2fdda482914e5f8

**Runtime output**

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;
```

**Code block metadata**

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Clause
MD5: b105a777a1afd79008f8580cda432cfe

**Runtime output**

Hello World!

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

---

### 12.6.1 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;
end Points;
```


---

### 12.6. Use type clause

---
private

    type Point is record
        X, Y : Integer;
    end record;

    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_Prog.Packages.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_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: f5d44dd1fee8cf4da7e738f9a764cc

Here, we have a use clause, so we have direct visibility to the content of Points’s visible part.

12.6.2 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
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).

12.6.3 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:

```ada
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.

12.6.4 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:

```ada
with Points;

procedure Show_Point is
  use all type Points.Point;
  P : Points.Point;
begin
  P := Init;
  P := P + 1;
end Show_Point;
```
Now, we've removed the prefix from all operations on the P variable.

## 12.7 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

### 12.7.1 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:

```ada
with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is new
      Ada.Numerics.Generic_Complex_Types (<>);
procedure Show_Any_Complex
   (Msg : String;
    Val : Complex_Types.Complex);
```

```ada
with Ada.Text_IO;
with Ada.Text_IO.Complex_IO;
procedure Show_Any_Complex
   (Msg : String;
    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;
```

---

Then, we implement a test procedure where we declare the `Complex_Float_Types` package as an instance of the `Generic_Complex_Types` package:

```
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;

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:** cc2a612c9884539f33154680854a4c82

**Runtime output**

```
C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)
```

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.

### 12.7.2 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:

```
with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Generic_Complex_Types;
with Show_Any_Complex;
```

(continues on next page)
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

show_use.adb:21:14: error: "Complex" is not visible
show_use.adb:21:14: error: multiple use clauses cause hiding
show_use.adb:21:14: error: hidden declaration at a-ngcoty.ads:42, instance at line_13
show_use.adb:21:14: error: hidden declaration at a-ngcoty.ads:42, instance at line_8

This example doesn't compile because we have direct visibility to both Complex_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.

12.7.3 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

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;
  -- ^ SOLVED: package is now specified.
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;

Another possibility is to write a use clause in the form use all type:

Listing 75: 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 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;

(continues on next page)
procedure Show_Complex_Float is new
    Show_Any_Complex (Complex_Float_Types);
begin
    C, D, X : Complex_Float_Types.Complex;
    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: 90333ff41e25afb1399f7f94ff7e2b566

Runtime output

C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)

For the sake of completeness, let’s declare and use variables of both Complex types:

Listing 76: 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 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);
    procedure Show_Complex_Long_Float is new
        Show_Any_Complex (Complex_Long_Float_Types);
begin
    C, D, X : Complex_Float_Types.Complex;
    E, F, Y : Complex_Long_Float_Types.Complex;
    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;
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;

As expected, the code compiles correctly.
CHAPTER
THIRTEEN

SUBPROGRAMS AND MODULARITY

13.1 Private subprograms

We've seen previously (page 438) 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 1: test.ads

```
private procedure Test;
```

Listing 2: 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 3: 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\textsuperscript{182}
- 10.1.2 Context Clauses - With Clauses\textsuperscript{183}

### 13.1.1 Private subprograms of a package

A more useful example is to declare private subprograms of a package. For example:

Listing 4: 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 5: 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 6: data_processing-calculate.ads

```ada
private
procedure Data_Processing.Calculate
  (D : in out Data);
```

Listing 7: data_processing-calculate.adb

```ada
procedure Data_Processing.Calculate
  (D : in out Data)
is
  begin
    -- Dummy implementation...
```

\textsuperscript{182} http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html
\textsuperscript{183} http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html
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.

### 13.1.2 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 9: 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 10: private_data_processing.adb**

```ada
package body Private_Data_Processing is
  procedure Process (D : in out Data) is
  begin
    (continues on next page)
  end
```

### 13.1. Private subprograms
In this code example, we have the private \texttt{Private Data Processing} package. In order to test it, we implement the private procedure \texttt{Test_Private_Data_Processing}. The fact that this procedure is private allows us to use the \texttt{Private Data Processing} package as if it was a non-private package. We then use the private \texttt{Test_Private_Data_Processing} procedure as our main application, so we can run it to test application the private package.

\textbf{Child subprograms of private packages}

We could also implement the Test subprogram that we use to test a private package \texttt{P} as a child subprogram of that package. In other words, we could write a procedure \texttt{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 \texttt{P} in the test procedure.

Let's rewrite the Test \texttt{Private Data Processing} procedure from the previous example as the child procedure \texttt{Private Data Processing.Test}:
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.)
Part IV

Resource Management
We discussed access types back in the Introduction to Ada course\(^\text{184}\). 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.

### 14.1 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.

#### 14.1.1 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:

```
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 566)).

\(^{184}\) https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-types-overview
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 595) later on.

### 14.1.2 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.
14.1.3 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:

```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;
```

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 512).)

In the Ada Reference Manual
- 3.10 Access Types\(^{185}\)

14.2 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:

```ada
procedure Show_Simple_Allocation is
   -- Declaring access type:
   type Integer_Access is access Integer;
(continues on next page)
```

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 486) 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 525) later on.)

**For further reading...**

Note that it is possible to use general access types to allocate objects on the heap:

```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;
```

Here, we’re using a general access type Integer_Access, but allocating an integer object on the heap.

**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](http://www.ada-auth.org/standards/22rm/html/RM-3-10.html)
14.2.1 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:\(^{187}\):
  - an access type can be unconstrained, but the actual object allocation must be constrained;
  - we can use a qualified expression (page 61) 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 84).

- When running out of memory while allocating via new, we get a Storage_Error exception because of the storage check (page 414).

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
(continues on next page)
```

\(^{187}\) https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-type-allocation-constraints
with Storage_Size => 128;
-- 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;

14.2.2 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);

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 components of the Arr array by simply writing Arr := (others => new Integer). This array aggregate (page 182) is syntactic sugar for a loop over Arr that allocates each component. (Note 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;
end Workers;
type Worker_Array is
   array (Positive range <>) of Worker_Access;
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;
      end Start;
      Put_Line ("Started Worker ",
                     & Id'Image);

      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

Chapter 14. Access Types
Runtime output

Started Worker # 1
Started Worker # 4
Started Worker # 2
Started Worker # 3
Started Worker # 5
Started Worker # 6
Started Worker # 7
Started Worker # 8
Started Worker # 9
Started Worker # 10
Started Worker # 11
Started Worker # 12
Started Worker # 13
Started Worker # 14
Started Worker # 15
Started Worker # 16
Started Worker # 17
Started Worker # 18
Started Worker # 19
Started Worker # 20
Some processing...
Stopped Worker # 1
Stopped Worker # 18
Stopped Worker # 17
Stopped Worker # 10
Stopped Worker # 15
Stopped Worker # 4
Stopped Worker # 16
Stopped Worker # 14
Stopped Worker # 6
Stopped Worker # 5
Stopped Worker # 2
Stopped Worker # 13
Stopped Worker # 12
Stopped Worker # 3
Stopped Worker # 9
Stopped Worker # 7
Stopped Worker # 8
Stopped Worker # 19
Stopped Worker # 11
Stopped Worker # 20

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).
14.3 Discriminants as Access Values

We can use access types when declaring discriminants. Let's see an example:

Listing 13: custom_recs.ads

```ada
package Custom_Recs is

   -- Declaring an access type:
   type Integer_Access is access Integer;

   -- Declaring a discriminant with this
   -- access type:
   type Rec (IA : Integer_Access) is record
       I : Integer := IA.all;
   end record;

   procedure Show (R : Rec);

end Custom_Recs;
```

Listing 14: 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 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:
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
-- -- R.IA := new Integer;
end Show_Discriminants_As_Access_Values;
```

### Code block metadata

**Project:** Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_Access_Values

**MD5:** c7850acefd8e5227f4be654faed13055

### Runtime output

```
R.IA = 10
R.I = 10
R.IA = 20
R.I = 30
```

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
    -- ^^^^^^^^^^^^^^^
    -- default value
  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);
```

(continues on next page)
Show (R2);
end Show_Discriminants_As_Access_Values;

Code block metadata

MD5: 968cb88ed7e9e6958ab86fb6f5a7ce2d

Runtime output

R.IA = 0
R.I = 0
R.IA = 20
R.I = 20

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
- 3.7.1 Discriminant Constraints

14.3.1 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;
```

188 http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
189 http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html
Advanced Journey With Ada: A Flight In Progress

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;
```

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 24), 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;

(continues on next page)
```

---

14.3. Discriminants as Access Values 495
Advanced Journey With Ada: A Flight In Progress

(end Persons;)

Code block metadata

MD5: 4144852aaaf95da62bc4781b1e8dc2717

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 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
```

(continues on next page)
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 615), which provide some advantages in terms of accessibility rules (page 534).

### 14.3.2 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:

```
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";
```

(continues on next page)
22
Show (P);
23
P := (Name => S2, Age => 40);
24
-- ^^ ERROR: we changed the
discriminant!
25
-- ^ ERROR: we changed the
discriminant!
26
Show (P);
end Show_Person;

Code block metadata
MD5: 96f474236eb6a07c377a5dec28b5767

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.

14.4 Parameters as Access Values

In addition to using discriminants as access values (page 492), 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
package Names is
  type Name is access String;
  procedure Show (N : Name);
end Names;

Code block metadata
MD5: 82ce94987dce9026aed54a0deb3cc548

This is the complete code example:

Listing 27: names.ads
package Names is
  type Name is access String;

(continues on next page)
procedure Show (N : Name);
end Names;

Listing 28: names.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Names is
   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:

Listing 30: names.ads

package Names is
   type Name is access String;
   procedure Show (N : String);
end Names;

Listing 31: names.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Names is
   procedure Show (N : String) is
      begin
         (continues on next page)
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 \texttt{N} parameter of the \texttt{Show} procedure. However, the difference between these two cases is that:

- \texttt{N : Name} is a reference to an object (because it's an access value) that is passed by value, and
- \texttt{N : String} is an object passed by reference.

### 14.4.1 Changing the referenced object

Since the \texttt{Name} type gives us access to an object in the \texttt{Show} procedure, we could actually change this object inside the procedure. To illustrate this, let's change the \texttt{Show} procedure to lower each character of the string before displaying it (and rename the procedure to \texttt{Lower_And_Show}):
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

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: 063a507284f5e7ffa669db2c8fdd3d6f

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

package Names is

    type Name is access String;

    procedure Lower_And_Show (N : in out String);

end Names;

Listing 37: names.adb

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 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;

(continues on next page)
With `Names; use Names;` we can use the `Names` package.

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;
```

**Code block metadata**

MD5: 783ea8c45ed8ad3e0007524c11b6b4c4

**Runtime output**

Name: john

### 14.4.2 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:

Listing 39: names.ads

```ada
package Names is
   type Name is access String;
   procedure Lower_And_Show (N : in out Name);
end Names;
```

Listing 40: names.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
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 41: 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: a4abfe6fdb1e5029e8eea17641cd960b

**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 546), as explained later on.

### 14.4.3 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 526), 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.

---

14.4. Parameters as Access Values 503
package Names is

  type Name is access String;
  type Constant_Name is access constant String;
  procedure Show (N : Constant_Name);

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;

end Names;

with Names; use Names;

procedure Show_Names is
  N : Name := new String'("John");
begin
  Show (Constant_Name (N));
end Show_Names;

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:
Advanced Journey With Ada: A Flight In Progress

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;
```

Listing 46: names.adb

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

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 \texttt{in String}, the information in the specification of \texttt{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 \texttt{N.all} in the \texttt{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 \texttt{'0ld} to refer to the original object before the sub-
program call. Unfortunately, we cannot use this attribute when dealing with *limited private types* (page 677) — 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) return Boolean;

   procedure Show (N : Name)
      with Post => Equal (N'Old = N);

private

   type Name is access String;

   function Init (S : String) return Name is
      (new String (S));

   function Equal (N1, N2 : Name)
      return Boolean is
      (N1.all = N2.all);

end Names;
```

Listing 49: names.adb

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

package body Names is

   procedure Show (N : Name) is
      begin
         for I in N'Range loop
            -- N (I) := To_Lower (N (I));
            Put_Line ("Name: " & N.all);
      end Show;

end Names;
```
Listing 50: show_names.adb

```ada
with Names; use Names;

procedure Show_Names is
    N : Name := Init ("John");
begin
    Show (N);
end Show_Names;
```

**Code block metadata**


MD5: 39691394d7a934869dc569eb72d1bf3a

**Build output**

names.ads:11:26: error: attribute "Old" cannot apply to limited objects
gprbuild: *** compilation phase failed

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:

Listing 51: names.ads

```ada
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));
```

(continues on next page)
function To_Constant_Name
(N : Name)
  return Constant_Name is
  (Constant_Name (N));
end Names;

Listing 52: names.adb

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 53: show_names.adb

with Names; use Names;
procedure Show_Names is
  N : Name := Init ("John");
begin
  Show (To_Constant_Name (N));
end Show_Names;

In this version of the source code, the Show procedure doesn't have any side-effects, as we cannot modify 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
14.5 Self-reference

As we've discussed in the section about incomplete types <Adv_Ada_Incomplete_Types>, we can use incomplete types to create a recursive, self-referencing type. Let's revisit a code example from that section:

Listing 54: linked_list_example.ads

```ada
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:
(continues on next page)```
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

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 := Last.Next;
        end loop;
        Last.Next := New_Last;
      end;
    end if;
  end Append_Rear;

  procedure Show (L : List) is
    Curr : List := L;
  begin
    if L = null then
      Put_Line ("[ "]);
    else
      Put ("[");
      loop
        Put (Curr.Value'Image);
        Curr := Curr.Next;
    end loop;
      Put ("]");
    end if;
  end Show;

end Linked_Lists;

(continues on next page)
Put (" ");
exit when Curr.Next = null;
Curr := Curr.Next;
end loop;
Put_Line ("]");
end if;
end Show;
end Linked_Lists;

Listing 57: test_linked_list.adb

with Linked_Lists;

procedure Test_Linked_List is
  package Integer_Lists is new
    Linked_Lists (? => Integer);
  use Integer_Lists;
  L : List;
begin
  Append_Front (L, 3);
  Append_Rear (L, 4);
  Append_Rear (L, 5);
  Append_Front (L, 2);
  Append_Front (L, 1);
  Append_Rear (L, 6);
  Append_Rear (L, 7);
  Show (L);
end Test_Linked_List;

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 498) for the Append_Front, Append_Rear and Show procedures.

In the Ada Reference Manual

• 3.10.1 Incomplete Type Declarations190

190 http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html
14.6 Mutually dependent types using access types

In the section on mutually dependent types (page 149), 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;
```

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.

14.7 Dereferencing

In the Introduction to Ada course¹⁹¹, 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:
```

¹⁹¹ https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-dereferencing
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:

```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);

  for I in Arr'Range loop
    Put_Line ("Arr (: " & Integer'Image (I) & ": ",
              & Integer'Image (Arr.all (I)));
  end loop;

end Show_Dereferencing;
```

In this example, we dereference the access value by writing `Arr.all`. We then assign an array aggregate to it — this becomes `Arr.all := (..., ...)`. Similarly, in the loop, we write `Arr.all (I)` to access the I component of the array.
### 14.7.1 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;
```

---

192 http://www.ada-auth.org/standards/22rm/html/RM-4-1.html
Runtime output

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>: 1</td>
<td>1</td>
</tr>
<tr>
<td>: 2</td>
<td>2</td>
</tr>
<tr>
<td>: 3</td>
<td>3</td>
</tr>
<tr>
<td>: 4</td>
<td>5</td>
</tr>
<tr>
<td>: 5</td>
<td>8</td>
</tr>
<tr>
<td>: 6</td>
<td>13</td>
</tr>
</tbody>
</table>

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:

```ada
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);
  end Show_Array_Assignments;
```

(continues on next page)
-- 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;

---

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 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;

---

Here, Arr_2 := 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 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;

---

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 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;

---

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 cases where there's no risk of ambiguities or oversights.
Again, we can replace \texttt{R.I} by \texttt{R.I}, as record components are implicitly dereferenced. Also, we could use implicit dereference when assigning to record components individually:

\begin{verbatim}
R.I := 1;
R.F := 5.0;
\end{verbatim}

However, we have to write \texttt{R.all} when assigning to the whole record \texttt{R}.

\section*{Attributes}

Finally, let's see an example of implicit dereference when using attributes:

\noindent \textbf{Listing 64: show_dereferencing.adb}

\begin{verbatim}
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
  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;
\end{verbatim}

\section*{Code block metadata}

\begin{small}
\begin{tabular}{ll}
MD5: & 5730e18c8d2ed5e26a4d7d325a46a7e9
\end{tabular}
\end{small}
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>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>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>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
- 4.1.1 Indexed Components
- 4.1.2 Slices

http://www.ada-auth.org/standards/22rm/html/RM-4-1.html
http://www.ada-auth.org/standards/22rm/html/RM-4-1-1.html
http://www.ada-auth.org/standards/22rm/html/RM-4-1-2.html
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.

### 14.8.1 Uniform multidimensional arrays

Consider an algorithm that processes data based on coefficients that depends 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:

```ada
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

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
```

```ada
package Data_Processing is

  type Quality_Level is
    (Simplified, Better, Best);

  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;
```

Code block metadata

Uniform_Table
MD5: 4c8602f5edcede0ac1231838c0a0a54b7

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:

```
Listing 67: data_processing-operations.ads
```

```ada
package Data_Processing.Operations is

  procedure Process (D : in out Data;
                     Q : Quality_Level);

end Data_Processing.Operations;
```

Listing 68: data_processing-operations.adb

```ada
package body Data_Processing.Operations is

(continues on next page)
```

520 Chapter 14. Access Types
procedure Process (D : in out Data;
        Q : Quality_Level) is
begin
    for I in D'Range loop
        for J in 1 .. Calc_Table (Q).Last loop
            null;
        end loop;
        -- D (I) := ...
        null;
    end loop;
end Process;
end Data_Processing.Operations;

**Code block metadata**

Uniform_Table
MD5: 2b0d2cee265509e64e507cfa6289bdcc

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)’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 coef-
ficients (or 40 percent of the table) aren't being used. Naturally, this is wasted memory
space. We can improve this by using ragged arrays.

**14.8.2 Non-uniform multidimensional array**

Ragged arrays are declared by using an access type to an array. By doing that, each ar-
ray 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,
```

(continues on next page)
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`:

```ada
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;
```

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 526) in more details later on.

This is the adapted Process procedure:
### Listing 71: data_processing-operations.ads

```ada
package Data_Processing.Operations is

  procedure Process (D : in out Data;
    Q : Quality_Level);

end Data_Processing.Operations;
```

### Listing 72: data_processing-operations.adb

```ada
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;

end Data_Processing.Operations;
```

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.

### 14.9 Aliasing

The term *aliasing*[^1] 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));
    Put_Line ("A2: " & Integer'Image (A2.all));
    A2.all := 24;
    Put_Line ("A1: " & Integer'Image (A1.all));
    Put_Line ("A2: " & Integer'Image (A2.all));
  end Show_Aliasing;
```

[^1]: https://en.wikipedia.org/wiki/Aliasing_(computing)

---

14.9. Aliasing
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**

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

14.9.1 Aliased objects

As we discussed previously (page 483), 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;
  I_Var : aliased Integer;
  A1 : Integer_Access;
begin
  A1 := I_Var'Access;
  A1.all := 22;
  Put_Line ('A1: ' & Integer'Image (A1.all));
end Show_Aliased_Obj;
```

Code block metadata

MD5: 98c8e47d7c2b5df8075918b239a8d476

Runtime output

A1: 22

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`:

Listing 76: 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;
  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));
  Put_Line ('A2: ' & Integer'Image (A2.all));
  A2.all := 24;
  Put_Line ('A1: ' & Integer'Image (A1.all));
  Put_Line ('A2: ' & Integer'Image (A2.all));
end Show_Aliased_Obj;
```

14.9. Aliasing 525
In this example, both A1 and A2 refer to the I_Var variable.

Note that these examples make use of these two features:

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><strong>type</strong> T_Acc <strong>is</strong> access all <strong>T</strong></td>
</tr>
<tr>
<td>Access-to-constant</td>
<td><strong>type</strong> T_Acc <strong>is</strong> access constant <strong>T</strong></td>
</tr>
</tbody>
</table>

Let's look at an example:

Listing 77: integer_access_types.ads

```
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;
```

---

200 http://www.ada-auth.org/standards/22rm/html/RM-3-3-1.html
201 http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
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_Cst.

**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:

```ada
package Integer_Access_Types is
  type Integer_Access is
    access Integer;

  type Integer_Access_All is
    access all Integer;

  type Integer_Access_Cst is
    access constant Integer;

  procedure Show;
end Integer_Access_Types;
```

```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_Cst := I_Var’Access;
  Fact_Ptr : constant Integer_Access_Cst := 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));
end Show;
```

(continues on next page)
& 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

with Integer_Access_Types;

procedure Show_Access_Modifiers is
begin
  Integer_Access_Types.Show;
end Show_Access_Modifiers;

Code block metadata

MD5: c9036f060859207ea14354b26dc8b981

Runtime output

Dyn_Ptr: 30
I_Var_Ptr: 0
I_Var_C_Ptr: 0
Fact_Ptr: 42

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\(^\text{202}\)

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;
  Al : Integer_Access;
begin
(continues on next page)

\(^\text{202}\) http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html
A1 := I_Var'Access;
A1.all := 22;
Put_Line ("A1: " & Integer'Image (A1.all));
end Show_Access_Error;

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 519). For example, we can rewrite a previous program using aliased constant objects:

```ada
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;
```

14.9. Aliasing

---

**Code block metadata**

MD5: 2a9904062eea96ae6dc209493d6f20d4

**Build output**

show_access_error.adb:8:10: error: prefix of "Access" attribute must be aliased
gprbuild: *** compilation phase failed
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:

```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.

#### 14.9.2 Aliased components

Components of an array or a record can be aliased. This allows us to get access to those components:

```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 : Integer_Access;
    A : aliased Integer_Access;
  end Rec;
  R : Rec;

begin
  R.I := 22;
  R.A := R.I'Access;
  Put_Line ("R.I: " & Integer'Image (R.I));
  Put_Line ("R.A: " & Integer'Image (R.A.all));
end Show_Aliased_Components;
```
I_Var_1 : Integer;
I_Var_2 : aliased Integer;
end record;

type Integer_Array is array (Positive range <>) of aliased Integer;
R : Rec := (22, 24);
Arr : Integer_Array (1 .. 3) := (others => 42);
A : Integer_Access;

begin

-- A := R.I_Var_1'Access;
-- ^ ERROR: cannot access
-- non-aliased
-- component

A := R.I_Var_2'Access;
Put_Line ("A: " & Integer'Image (A.all));

A := Arr (2)'Access;
Put_Line ("A: " & Integer'Image (A.all));
end Show_Aliased_Components;

In this example, we get access to the I_Var_2 component of record R. (Note that trying to access the 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 Arr.

Declaring components with the 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

- 3.6 Array Types

---

14.9.3 Aliased parameters

In addition to aliased objects and components, we can declare *aliased parameters* (page 365), 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:

<table>
<thead>
<tr>
<th>Parameter mode</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>aliased in</td>
<td>Access-to-constant</td>
</tr>
<tr>
<td>aliased out</td>
<td>Access-to-variable</td>
</tr>
<tr>
<td>aliased in out</td>
<td>Access-to-variable</td>
</tr>
</tbody>
</table>

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
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada

procedure Add_One (A : Integer_Access) is
begin
   A.all := A.all + 1;
end Add_One;

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;
```

**Code block metadata**


MD5: 076238603036a851cafcc013f3b8c8f3

**Runtime output**

| Value : I 22 |
| Value : I 23 |

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
- 6.2 Formal Parameter Modes
- 6.4.1 Parameter Associations

204 http://www.ada-auth.org/standards/22rm/html/RM-6-1.html
205 http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
206 http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
14.10 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

14.10.1 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:

```
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

207 http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html
208 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.)

### 14.10.2 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.

### 14.10.3 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 541), which we discuss later. Also, they are based on the accessibility levels (page 535) 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)
-- 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: b3bed7e6b2a8d4c78a2e7a7d2ce99f736

Build output

14.10. Accessibility Levels and Rules: An Introduction
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 \( L_1\_IA := L_1\_Var\}'Access \) are legal because we're operating at the same accessibility level. Also, \( L_1\_IA := L_0\_Var\}'Access \) is legal because \( L_1\_IA \) is at a deeper level than \( L_0\_Var\}'Access \).

However, many operations in the code example are illegal. For instance, \( L_0\_IA := L_1\_Var\}'Access \) and \( L_0\_IA_2 := L_1\_Var\}'Access \) 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 \( L_1\_Var\}'Access \) 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 \( L_0\_Integer\_Access \rightarrow L_1\_IA \) 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 \( L_1\_Integer\_Access \) type to the \( L_1\_B\_Integer\_Access \), which are both at the same level.
14.10.4 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.

### 14.10.5 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 549).

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 410). 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
  (continues on next page)
```

14.10. Accessibility Levels and Rules: An Introduction 541
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
-- I_Var_2 only exists in Inner_Block
--
I_Var_2 : aliased Integer := 42;
--
-- A2 only exists in Inner_Block
--
A2 : Integer_Access;

begin
  A2 := I_Var_1'Access;
  Put_Line ("A2.all: 
            & Integer'Image (A2.all));

  A1 := I_Var_2'Access;
  -- PROBLEM: A1 and Integer_Access type
  -- have longer lifetime than
  -- I_Var_2
  Put_Line ("A1.all: 
            & Integer'Image (A1.all));

  A2 := I_Var_2'Access;
  -- PROBLEM: A2 has the same lifetime as
  -- I_Var_2, but Integer_Access
  -- type has a longer lifetime.
  Put_Line ("A2.all: 
            & Integer'Image (A2.all));
  end Inner_Block;

Put_Line ("Inner Block has ended!");
Put_Line ("A1.all: 
          & Integer'Image (A1.all));
end Show_Dangling_Reference;
```

Code block metadata

MD5: 98e597f3f6a12075c474612bb42f4cb7

Build output

```
show_dangling_reference.adb:33:13: error: non-local pointer cannot point to local
  object
show_dangling_reference.adb:41:13: error: non-local pointer cannot point to local
  object
gprbuild: *** compilation phase failed
```

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
scope and its lifetime has ended — A1 would still be pointing to an object that does not longer exist.

2. In the \texttt{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.
   \begin{itemize}
   \item However, as mentioned in the previous section, Ada also cares about the lifetime of access types. In fact, since the \texttt{Integer\_Access} type is declared outside of the \texttt{Inner\_Block}, it has a longer lifetime than A2 and I\_Var\_2.
   \item 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.
   \end{itemize}

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:

\begin{verbatim}
A2 := I\_Var\_2'Access;
A1 := A2;
\end{verbatim}

\begin{verbatim}
-- 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!
\end{verbatim}

Here, we're introducing the \texttt{A1 := A2} assignment. The problem with this is that I\_Var\_2's lifetime ends when the \texttt{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 \texttt{A2 := I\_Var\_2'Access} assignment to be safe, as we're not using the \texttt{A1 := A2} assignment there. Since we're confident that no error could ever occur in the \texttt{Inner\_Block} due to the assignment to A2, we could replace it with \texttt{A2 := I\_Var\_2'Unchecked\_Access}, so that the compiler accepts it. We discuss more about the unchecked access attribute \textit{later in this chapter} (page 544).

Alternatively, we could have solved the compilation issue that we see in the \texttt{A2 := I\_Var\_2'Access} assignment by declaring another access type locally in the \texttt{Inner\_Block}:

\begin{verbatim}
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;
\end{verbatim}

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.

(Note that in the \texttt{Inner\_Block}, we could have simply named the local access type \texttt{Integer\_Access} instead of \texttt{Integer\_Local\_Access}, thereby masking the \texttt{Integer\_Access}... \textit{later in this chapter} (page 544).)
We discuss the effects of dereferencing dangling references later in this chapter (page 551).

### 14.11 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 534) that the accessibility levels verify the lifetime of access types. Let's see a simplified version of a code example from that section:

```plaintext
package Integers is
  type Integer_Access is access all Integer;
end Integers;
```

```plaintext
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 `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:

```ada
package Integers is
  type Integer_Access is access all Integer;
end Integers;
```

```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;
```

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 arise 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\(^{209}\)

---

14.12 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
```

(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;
```
Advanced Journey With Ada: A Flight In Progress

Listing 105: show_unchecked_deallocation.adb

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

procedure Show_Unchecked_Deallocation is

   I : Integer_Access;

begin
   Put ("We haven't called new yet... ");
   Show_Is_Null (I);

   Put ("Calling new... ");
   I := new Integer;
   Show_Is_Null (I);

   Put ("Calling Free... ");
   Free (I);
   Show_Is_Null (I);
end Show_Unchecked_Deallocation;
```

Code block metadata

MD5: a9f2df04e2fe0d5ee8c17249b4ae315a

Runtime output

We haven't called new yet... access value is null.
Calling new... access value is NOT null.
Calling Free... access value is null.

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:

Listing 106: show_unchecked_deallocation.adb

```ada
with Integer_Types; use Integer_Types;

procedure Show_Unchecked_Deallocation is

   I : Integer_Access;

begin
   I := new Integer;
   Free (I);
   Free (I);
   Free (I);
end Show_Unchecked_Deallocation;
```

Code block metadata
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 \((\text{I})\), 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

14.12.1 Unchecked Deallocation and Dangling References

We've discussed dangling references (page 541) 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 ShowUncheckedDeallocation 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);
```

(continues on next page)

---

\(^{210}\) [http://www.ada-auth.org/standards/22rm/html/RM-4-8.html]


14.12. Unchecked Deallocation
-- A call to Free (I_2) is
-- NOT OK:
Free (I_2);
end Show_Unchecked_Deallocation;

Code block metadata

MD5: ee5c20209a113a6c1bc7895b8ebdb174

Runtime output

free(): double free detected in tcache 2
raised PROGRAM_ERROR : unhandled signal

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:

Listing 108: show_unchecked_deallocation.adb

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 558) later on.

### 14.12.2 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:

```ada
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;
```

14.12. Unchecked Deallocation 551
Runtime output

```
I_1.all = 42
I_2.all = 42
Freeing I_1
I_2.all = 7443
```

In this example, we allocate an object for `I_1` and make `I_2` point to the same object. Then, we call `Free (I)`, which has the following consequences:

- The call to `Free (I_1)` will try to reclaim the storage for the original object (`I_1.all`), so it may be reused for other allocations.
- `I_1 = null` after the call to `Free (I_1)`.
- `I_2` becomes a dangling reference by the call to `Free (I_1)`.
  - In other words, `I_2` is still non-null, and what it points to is now undefined.

In principle, we could check for `null` before trying to dereference the access value. (Remember that when deallocating an object via a call to Free, the corresponding access value is set to `null`.) In fact, this strategy works fine for `I_1`, but it doesn't work for `I_2` because the access value is not `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!

In the Ada Reference Manual

- 13.9.1 Data Validity\(^{212}\)
- 13.11.2 Unchecked Storage Deallocation\(^{213}\)

14.12.3 Restrictions for Ada.Unchecked_Deallocation

There are two unsurprising restrictions for 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 `new`.)

Let's see an example of these restrictions:

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;

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.
14.13 Null & Not Null Access

Note: This section was originally written by Robert A. Duff and published as Gem #23: Null Considered Harmful\(^ {214} \) and Gem #24\(^ {215} \).

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
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:

```
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:

\(^ {214} \) https://www.adacore.com/gems/ada-gem-23
\(^ {215} \) 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)
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 an 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\(^\text{216}\) 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;
    P;
  end P;

  procedure Q (X : not null Ref_Element) is
  begin
    for I in 1 .. 1000 loop
      P (X);
    end loop;
    Q;
  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;
```

If X (Index) occurs inside Process_Array, there is no need to check that Index is in range, because the check is pushed to the caller.

14.14 Design strategies for access types

Previously, we learned about dangling references (page 541) and discussed the effects of dereferencing them (page 551). Also, we've seen the relationship between unchecked deallocation and dangling references (page 549). 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.

14.14.1 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 677).)

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;

end Access_Type_Abstraction;
```

(continues on next page)
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);

procedure Destroy (Obj : in out Info);

private

    type Info is access String;

end Access_Type_Abstraction;

Listing 121: access_type_abstraction.adb

with Ada.Unchecked_Deallocation;

package body Access_Type_Abstraction is

    function To_Info (S : String) return Info is
        (new String'(S));

    function To_String (Obj : Info)
        return String is
        (if Obj /= null then Obj.all else "");

    function Copy (Obj : Info) return Info is
        (To_Info (Obj.all));

    procedure Copy (To : in out Info;
                    From : Info) is
        begin
            Destroy (To);
            To := To_Info (From.all);
        end Copy;

    procedure Append (Obj : in out Info;
                     S : String) is
        New_Info : constant Info :=
            To_Info (To_String (Obj) & S);
        begin
            Destroy (Obj);
            Obj := New_Info;
        end Append;

    procedure Reset (Obj : in out Info) is
        begin
            Destroy (Obj);
        end Reset;

    procedure Destroy (Obj : in out Info) is
        procedure Free is
            new Ada.Unchecked_Deallocation
                (Object => String,
                Name => Info);

end Access_Type_Abstraction;
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
begin
    Free (Obj);
end Destroy;

end Access_Type_Abstraction;
```

Listing 122: main.adb

```ada
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 (""----------"" );

(continues on next page)
```
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.

### 14.14.2 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
end Access_Type_Abstraction;
```

(continues on next page)
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;

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;

-- 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

(continues on next page)
procedure Initialize (Obj : in out Info) is
begin
  -- 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 (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;
Listing 125: main.adb

```ada
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;

Code block metadata

Access_Type_Limited_Controlled_Abstraction
MD5: e98659ad1b87be56fb173fa407ab7e82

Runtime output

TO_INFO / COPY
Obj_1 : hello
Obj_2 : hello
----------
RESET / APPEND
Obj_1 :
Obj_2 : hello world
----------
COPY
Obj_1 : hello world
Obj_2 : hello world
----------
RESET

(continues on next page)
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.)

14.15 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.

14.15.1 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:

Listing 126: p.ads

```ada
procedure P (I : in out Integer);
```

Listing 127: p.adb

```ada
procedure P (I : in out Integer) is
begin
  null;
end P;
```

Listing 128: proc.adb

```ada
with P;

procedure Proc is
I : Integer := 0;
begin
```

(continues on next page)
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.)

### 14.15.2 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);
```

---

14.15. Access to subprograms
end Access_To_Subprogram_Params;

Listing 131: access_to_subprogram_params.adb

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;

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;

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:

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
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

14.15.3 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;
```

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
```


14.15. Access to subprograms
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.

### 14.15.4 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;

end Access_To_Subprogram_Types;
```
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
code
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;
```

Runtime output

```
14.15. 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 index of the component, e.g. PA (2). Once we specify the procedure we want to use, we simply pass the parameters, e.g.: PA (2) (Some_Int).

Now, let's use the Rec_Access_To_Procedure record type in a test application:

```
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
   RA : Rec_Access_To_Procedure;
   Some_Int : Integer := 0;
begin
   Put_Line ("Some_Int: " & Some_Int'Image);
   RA := (AP => Add_Ten'Access);
   RA.AP (Some_Int);
   Put_Line ("Some_Int: " & Some_Int'Image);
   RA := (AP => Add_Twenty'Access);
   RA.AP (Some_Int);
   Put_Line ("Some_Int: " & Some_Int'Image);
end Show_Access_To_Subprograms;
```

Here, we declare two record aggregates where we specify the AP component, e.g.: (AP => Add_Ten'Access), which indicates the access-to-subprogram we want to use. We can call the subprogram by simply accessing the AP component, i.e.: RA.AP.
14.15.5 Access-to-subprogram as discriminant types

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 492) 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:

Listing 141: custom_processing.ads

```ada
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

```ada
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;
```

(continues on next page)
In this example, we declare the access-to-subprogram type \texttt{Integer Processing}, which we use as the IP discriminant of the \texttt{Rec} type. In the \texttt{Process} procedure, we call the IP procedure that we specified as the record's discriminant (\texttt{R.IP (R.I)}).

Before we look at a test application for this package, let's implement another small procedure:

```ada
procedure Mult_Two (I : in out Integer);
```

```ada
procedure Mult_Two (I : in out Integer) is
  begin
    I := I * 2;
  end Mult_Two;
```

Now, let's look at the test application:

```ada
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);
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

25 Put_Line ("---- R_Add_Ten ----");
26 Show (R_Add_Ten);
27
28 Put_Line ("Calling Process procedure...");
29 Process (R_Add_Ten);
30 Show (R_Add_Ten);
31
32 Put_Line ("---- R_Mult_Two ----");
33 Show (R_Mult_Two);
34
35 Put_Line ("Calling Process procedure...");
36 Process (R_Mult_Two);
37 Show (R_Mult_Two);
38 end Show_Access_To_Subprogram_Discriminants;

Code block metadata

MD5: 544c224f8bc8e6ba2db4914c2a3dcff4

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.

14.15.6 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

generic
1 type T is private;
2
3 --
4 -- Declaring formal access-to-subprogram
5 -- type:
6 --
7 type T_Processing is
8 access procedure (Element : in out T);
9
10 --
11 -- Declaring formal access-to-subprogram
12 -- parameter:
13 --
14 Proc : T_Processing;
15

(continues on next page)
with function Image_T (Element : T) return String;
package Gen_Custom_Processing is
  type Rec is private;
  procedure Init (R : in out Rec; Value : T);
  procedure Process (R : in out Rec);
  procedure Show (R : Rec);
private
  type Rec is record
    Comp : T;
  end record;
end Gen_Custom_Processing;

Listing 147: gen_custom_processing.adb

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:
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,
     -- ^^^^^^^^^^^^^^^^^^
     -- access-to-subprogram type
     Proc => Add_Ten'Access,
     -- ^^^^^^^^^^^^^^^^^^
     -- access-to-subprogram
     Image_T => Integer'Image);

  use Custom_Processing;

  R_Add_Ten : Rec;

  begin
    Init (R_Add_Ten, 1);
    Put_Line ("---- R_Add_Ten ----");
    Show (R_Add_Ten);
    Put_Line ("Calling Process procedure...");
    Process (R_Add_Ten);
    Show (R_Add_Ten);
  end Show_Access_To_Subprogram_As_Formal_Parameter;

Here, we instantiate the Gen_Custom_Processing package as Custom_Processing and specify the access-to-subprogram type and the access-to-subprogram.
14.15.7 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:

Listing 149: data_processing.ads

```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;
```

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;
```

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
  (continues on next page)
```

Code block metadata

MD5: 59399e0809deb476f608faab7e4398bd
Finally, we implement a test application that selects each of the logging procedures that we've just implemented:

Listing 153: show_access_to_subprograms.adb

```ada
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;
```

**Runtime output**

```
==== Log_Element_Per_Line =====
Elements:
```

14.15. 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.

### 14.15.8 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**
```
function Init_Zero return Float;
```

**Listing 157: init_one.ads**
```
function Init_One return Float;
```

**Listing 158: init_zero.adb**
```
function Init_Zero return Float is
begin
  return 0.0;
end Init_Zero;
```

**Listing 159: init_one.adb**
```
function Init_One return Float is
begin
  return 1.0;
end Init_One;
```

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);
```

(continues on next page)
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;

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\(^{218}\).

Here’s another example, first with `null`:

Listing 162: show_null_procedure.ads

```ada
package Show_Null_Procedure is
  type Element is limited null record;
  -- Not implemented yet
```

\(^{218}\) [https://www.adacore.com/gems/ada-gem-24](https://www.adacore.com/gems/ada-gem-24)
type Ref_Element is access all Element;

type Table is limited null record;
  -- Not implemented yet

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.

end Show_Null_Procedure;

Code block metadata
  ↓Subprograms.Null_Procedure
MD5: ac21dd76ed9fb7f26839c24210cf4425

and without null:

Listing 163: show_null_procedure.ads

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;

Code block metadata
  ↓Subprograms.Null_Procedure
MD5: 7341d8f23cd4efe45698481be452a9e8

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:
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
        (T : Table;
         Action : not null Iterate_Action
             := Do_Nothing'Access);

private

    type Element is limited
        record
            Component : Integer;
        end record;

    type Table is limited null record;

end Example;

package body Example is

    An_Element : aliased Element;

    procedure Iterate
        (T : Table;
         Action : Iterate_Action := null)
        is
        begin
            if Action /= null then
                Action (An_Element'Access);
                -- In a real program, this would do
                -- something more sensible.
            end if;
        end Iterate;

    procedure Iterate_2
        (T : Table;
         Action : not null Iterate_Action
             := Do_Nothing'Access)
        is
        begin
            Action (An_Element'Access);
            -- In a real program, this would do
            -- something more sensible.
        end Iterate_2;

(continues on next page)
end Example;

Listing 166: show_example.adb

with Example; use Example;

procedure Show_Example is
  T : Table;
begin
  Iterate_2 (T);
end Show_Example;

**14.15.9 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
      null;
    end Do_Something;
end Obj;
```

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.
Here, we declare the Access_Proc type as an access type to protected procedures. Then, we declare the variable Acc and assign 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

package Work_Registry is

  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;
```

Listing 170: work_registry.adb

```
(continues on next page)
```
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;

end Work_Registry;

Code block metadata

MD5: 5dfa8ab9b90000ab4f6b7575e1cde5e53

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
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:

Listing 171: integer_storage_system.ads

```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;

type Integer_Storage_Access is access Integer_Storage;

type Integer_Storage_Array is array (Positive range <>) of
    Integer_Storage_Access;

end Integer_Storage_System;
```

Listing 172: integer_storage_system.adb

```ada
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;
```

Code block metadata

MD5: a388d792bc85709785d324c914d9d236
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 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 : 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 ProcessAll 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:

Listing 174: show_access_to_protected_subprograms.adb

```ada
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) := new Integer_Storage;
    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

590 Chapter 14. Access Types
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 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 Int_Stor (2)).

Also, even though we have registered the same procedure (Show) of the same type (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 Float_Storage (based on the code that we used for the Integer_Storage type) and register some objects of Float_Storage type into the system (with a couple of additional calls to Register). If we then call WHR.Process_All, we'd see that the system is able to cope with objects of both Integer_Storage and Float_Storage types. In fact, the system implemented with the 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 Valid_Work_Handler type.

### 14.16 Accessibility Rules and Access-To-Subprograms

In general, the accessibility rules that we discussed previously for access-to-objects (page 534) 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 535) and the accessibility rules (page 536) that are based on those levels. The same accessibility rules apply to access-to-subprograms. As we said previously (page 539), operations targeting objects at a 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;
```
Listing 176: 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;
  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
      -- Access_To_Function type is less deep
      -- than Add_One.
    end;
    Put_Line ("Value: " & Value'Image);
    Value := Func (Value);
  end Show_Access_To_Subprogram_Error;
```

Code block metadata

Access_To_Subprograms.Access_To_Subprogram_Accessibility_Error_Less_Deep
MD5: 2a068732606a1fee156e82515febe9c4

Build output

show access to subprogram_error.adb:16:15: error: subprogram must not be deeper
than access type
gprbuild: *** compilation phase failed

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

```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 178: add_one.ads

```ada
function Add_One (I : Integer) return Integer;
```
Advanced Journey With Ada: A Flight In Progress

Listing 179: add_one.adb

```ada
function Add_One (I: Integer) return Integer is
begin
    return I + 1;
end Add_One;
```

Listing 180: 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;
with Add_One;

procedure Show_Access_To_Subprogram_Error is
    Func: Access_To_Function;
    Value: Integer := 0;
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.

14.16.1 Unchecked Access

Previously, we discussed about the Unchecked Access attribute (page 544), 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;
```

14.16. Accessibility Rules and Access-To-Subprograms
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);
  Value : Integer := 0;
begin
  Func := Add_One'Access;
  Put_Line ("Value: " & Value'Image);
  Value := Func (Value);
  Put_Line ("Value: " & Value'Image);
end Show_Access_To_Subprogram_Error;
```

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);
```

(continues on next page)
type Access_To_Function is
    access function (I : Integer) return Integer;
end Access_To_Subprogram_Types;

Listing 184: show_access_to_subprogram_error.adb

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.

14.17 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 131). 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.
14.17.1 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:

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;
```

Runtime output

I’Address : 00007FFC0FE22A04
AI.all’Address : 00007FFC0FE22A04
To_Address (AI) : 00007FFC0FE22A04

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 sub-type 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 402) 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:

```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: 5c6fc19ca1aa227f6ba97ea610dd921b*
Advanced Journey With Ada: A Flight In Progress

Runtime output

<table>
<thead>
<tr>
<th>Address</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00007FFF08F33EEC</td>
<td>00007FFF08F33EEC</td>
</tr>
</tbody>
</table>

\(\text{AI}_1 = \text{AI}_2\)

Here, we convert the address back to an access value by calling \(\text{To\_Pointer}(A)\). (When running this object, we see that \(\text{AI}_1\) and \(\text{AI}_2\) 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 \(\text{Object}\) parameter of the generic \(\text{Address\_To\_Access\_Conversions}\) package — is unbounded, the result of a call to \(\text{To\_Pointer}\) may not have bounds.

Let's adapt the previous code example and replace the \texttt{Integer} type by the (unbounded) \texttt{String} type:

Listing 187: show_address_conversion.adb

```
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;
```

(continues on next page)
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
Address_Conversion
MD5: b1adcaa1f2cb4dfbd157aeaf7893bd72

Build output
show_address_conversion.adb:9:04: warning: in instantiation at s-atacco.ads:43,
             [enabled by default]
show_address_conversion.adb:9:04: warning: Object is unconstrained array type,
             [enabled by default]
show_address_conversion.adb:9:04: warning: To_Pointer results may not have bounds,
             [enabled by default]

Runtime output
AI_1.all'Address : 00007FFED8849C8
AI_2.all'Address : 00007FFED8849C8
AI_1 = AI_2
AI_1: Hello
AI_2: Hello

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

221 http://www.ada-auth.org/standards/22rm/html/RM-13-7-2.html

14.17. Access and Address 599
ANONYMOUS ACCESS TYPES

15.1 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 625) and it's specifying an anonymous access-to-object type (page 605). Another flavor of anonymous access types are anonymous access-to-subprograms (page 648). 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;
begin
  A := I_Var'Access;
  A.all := 22;
  Put_Line ("A.all: ", Integer'Image (A.all));
end Show_Anonymous_Access_Types;
```

Listing 1: show_anonymous_access_types.adb
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

15.1.1 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 608).)

15.1.2 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'Access attribute always works.

Let's see an example:

Listing 2: 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;
```

http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
Listing 3: 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;
```

Code block metadata

MD5: 2822ca0bd6ac251dccc1ced60747fbel

Runtime output

A.all: 22
B.all: 22

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.

(Nota 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:

Listing 4: aux.ads

```ada
package Aux is
  type Integer_Access is access all Integer;
  procedure Show (Name : String; V : Integer_Access);
end Aux;
```

Listing 5: aux.adb

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

package body Aux is
  procedure Show (Name : String; V : Integer_Access) is
  begin
    (continues on next page)
```

15.1. Named and Anonymous Access Types
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 adaptations 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 indicated 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 7: show_anonymous_access_subtype_error.adb

```ada
procedure ShowAnonymousAccessSubtypeError 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 ShowAnonymousAccessSubtypeError;
```

Code block metadata

  Anonymous_Access_Types.Anonymous_Access_Subtype_Error
MD5: cecfe703ea8b42bad61c45f33cbb67b

Build output

show_anonymous_access_subtype_error.adb:10:09: error: target designated subtype not compatible with type "Standard.Integer"
show_anonymous_access_subtype_error.adb:13:09: error: object subtype must statically match designated subtype
gprbuild: *** compilation phase failed

Even though Integer_1_10 is a subtype of Integer, we cannot assign A to B because the subtype that their access type declarations refer to — 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 605) and anonymous access-to-subprograms (page 648) cover more specific details on anonymous access types.

### 15.2 Anonymous Access-To-Object Types

In the previous chapter (page 481), we introduced named access-to-object types and used those types throughout the chapter. Also, in the previous section (page 601), 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 615) and anonymous access parameters (page 625) later on.) Let’s see a code example that includes all these cases:
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

    type Rec (AI : access Integer) is record
    -- ^ Anonymous access type
        Internal_AI : access Integer;
    -- ^ Anonymous access type
    end record;

end All_Anonymous_Access_To_Object_Types;

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
15.2.1 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 9: all_anonymous_access_to_object_types.ads

```ada
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
    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;
```

(continues on next page)
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:

```ada
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.

### 15.2.2 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` 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:

```ada
with Ada.Unchecked_Deallocation;

package Missing_Features is
```

(continues on next page)
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:

```ada
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:

```ada
procedure Free is
  new Ada.Unchecked_Deallocation
  (Object => Integer, Name => Integer_Access);
-- Specifying a limited storage size for the access type:
```

```ada
type Integer_Access_Store_128 is
  access Integer
  with Storage_Size => 128;
-- Limiting the storage size for the access type to zero:
```

```ada
type Integer_Access_Store_0 is
  access Integer
  with Storage_Size => 0;
```

end Missing_Features;
```

In the Missing_Features package, we see some of the features that we cannot use for the anonymous \texttt{access Integer} type, but that are available for equivalent named access types:

- There's no specific memory pool associated with the access object \texttt{IA}. In contrast, named types — such as \texttt{String Access} and \texttt{Integer Access} — have an associated pool, and we can use the \texttt{Storage_Pool} aspect and the \texttt{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:

```ada
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:
  Free (Integer_Access (IA));
end Show_Dangerous_Deallocation;
```

**Code block metadata**

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:

Listing 12: show_dangerous_allocation.adb

```ada
pragma Restrictions (No_Anonymous_Allocators);

procedure Show_Dangerous_Allocation is
  IA : access Integer;
begin
  IA := new Integer;
  IA.all := 30;
end Show_Dangerous_Allocation;
```

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:

Listing 13: show_successful_deallocation.adb

```ada
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;
```
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
-- Deallocation of the access object that has
-- an associated type:
Free (Typed_IA);
end Show_Successful_Deallocation;
```

### Code block metadata

**Project:** Courses.Advanced_Ada.Resource_Management.Anonymous_Access_TYPES.Anonymous_Access_To_Object_Types.Deallocation_Anonymous_Access_To_Object_1

**MD5:** eff8b54adfc8ccee10920dc3620ff1b9

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 541) 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:

#### Listing 14: hidden_anonymous_allocation.ads

```ada
generic
  type T is private;
package Hidden_Anonymous_Allocation is
  function New_T return not null access T;
  procedure Free (Obj : access T);
end Hidden_Anonymous_Allocation;
```

#### Listing 15: hidden_anonymous_allocation.adb

```ada
with Ada.Unchecked_Deallocation;
package body Hidden_Anonymous_Allocation is
  type T_Access is access all T;
  procedure T_Access_Free is
    new Ada.Unchecked_Deallocation
      (Object => T, Name => T_Access);
  function New_T return not null access T is
    begin
      return T_Access'(new T);
      -- Using allocation of the T_Access type:
```

(continues on next page)
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. For example:

```
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;
```

15.2. Anonymous Access-To-Object Types
Advanced Journey With Ada: A Flight In Progress

Listing 17: 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:

Listing 18: show_automatic_deallocation.adb

```ada
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;
end Show_Automatic_Deallocation;
```

Code block metadata

MD5: 4381db8ba87717978a9629b1e6a5f1fc

In this case, we create the I object on the stack by simply declaring it. Then, we get access to it and assign it to the IA access object.

With this approach, we're indirectly allocating an object for an anonymous access type by creating it on the stack. Also, because we know that the I is automatically deallocated when it gets out of scope, we don't have to worry about explicitly deallocating the object referred by IA.
When to use anonymous access-to-objects types

In summary, anonymous access-to-object types have many drawbacks that often outweigh their benefits (page 602). 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 628). 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.

15.3 Access discriminants

Previously, we’ve discussed discriminants as access values (page 492). 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 19: custom_recs.ads

```
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 20: custom_recs.adb

```
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 21: 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

```
R.IA = 10
R.I = 10
R.IA = 20
R.I = 30
```

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:

Listing 22: persons.ads

```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;
```

Listing 23: 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;
```

(continues on next page)
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.

### 15.3.1 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:
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:

Listing 26: custom_recs.ads

```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 limited
  record
    I : Integer := IA.all;
  end record;

  procedure Show (R : Rec);
end Custom_Recs;
```

Code block metadata

MD5: 9e8683c7a27e9097fd2003ad91bac269

Build output

custom_recs.ads:6:21: warning: coextension will not be deallocated when its associated owner is deallocated [enabled by default]

Note that, if we don’t provide a value for the access discriminant when declaring an object R, the default value is allocated (via `new`) during R’s creation.

Listing 27: show_access_discriminants.adb

```ada
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);
end Show_Access_Discriminants;
```

(continues on next page)
12 -- R gets out of scope here, and the object
13 -- allocated via new hasn't been deallocated.
14 end Show_Access_Discriminants;

15.3.2 Benefits of Access Discriminants

Access discriminants have the same benefits that we've already seen earlier while dis-
cussing discriminants as access values (page 492). 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:

Listing 28: persons.ads

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;

Listing 29: persons.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Persons is

  procedure Show (P : Person) is
    begin
      Put_Line ("Name = ", P.Name.all);

end Persons;
### Listing 30: show_person.adb

```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:
  --
  -- S : Access_String := new String'("John");
  -- P : Person (S);
begin
  P.Age := 30;
  Show (P);
end Show_Person;
```

#### Code block metadata

- **MD5:** e918db3790c7ffeeb7c0f54ced9f48b9

#### Build output

```
gprbuild: *** compilation phase failed
```

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 : Access_String := new String`). However, if we use an access discriminant in the declaration of Person, the code compiles fine:

### Listing 31: persons.ads

```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;
```
**Advanced Journey With Ada: A Flight In Progress**

### Listing 32: show_person.adb

```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: 6516fb4e0cbbac9cfe07a56e48ea9ff3

**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.

### 15.3.3 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:

```ada
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);
      -- ^^^^^^^^^^^^^^^^^^^^^
      -- ERROR: dangling reference!
    end Local_Init;

  P : Person := Local_Init;

begin
  Show (P);
end Show_Person;
```

**Code block metadata**

15.3. Access discriminants 621
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.

### 15.4 Self-reference

Previously, we've seen that we can declare self-references (page 509) using named access types. We can do the same with anonymous access types. Let's revisit the code example that implements linked lists:

```ada
package Linked_Lists is

generic
type T is private;

procedure Append_Front (L : in out List; E : T);

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 declaration 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 35: 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,
               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);
        -- ^^^^ type conversion:
        -- "access Component" to
        -- "List"
        Last.Next := New_Last;
      end loop;
      Last.Next := New_Last;
    end;
  end if;
end Append_Rear;

procedure Show (L : List) is
  Curr : List := L;
begin
  if L = null then
  Put_Line ("[ ]");
  else
  Put ("[");
    loop
      Put (Curr.Value'Image);
      Put (" ");
      exit when Curr.Next = null;
      Curr := Curr.Next;
    end loop;
  Put_Line ("]");
  end if;
end Show;

(continues on next page)
```

15.4. Self-reference 623
end Linked_Lists;

Listing 36: test_linked_list.adb

with Linked_Lists;

procedure Test_Linked_List is
package Integer_Lists is new
    Linked_Lists (T => Integer);
use Integer_Lists;

L : List;
begin
    Append_Front (L, 3);
    Append_Rear (L, 4);
    Append_Rear (L, 5);
    Append_Front (L, 2);
    Append_Front (L, 1);
    Append_Rear (L, 6);
    Append_Rear (L, 7);
    Show (L);
end Test_Linked_List;

15.5 Mutually dependent types using anonymous access types

In the section on mutually dependent types using access types (page 512), we've seen a code example that was using named access types. We could now rewrite it using anonymous access types:
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.

### 15.6 Access parameters

In the previous chapter, we talked about *parameters as access values* (page 498). 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):

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:
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;

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;

with Names; use Names;

procedure Show_Names is
  N : access String := Init ("Lily", "Ann");
begin
  Show (N);
end Show_Names;

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 498) 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 42: 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 43: 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 44: show_names.adb

```ada
with Names; use Names;

procedure Show_NAMES is
  N : String := Init ("Lily", "Ann");
  begin
    Show (N);
  end Show_NAMES;
```

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 647). 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 628). Another time when access parameters are vital is for inherited primitive operations for tagged types. We discuss this later on (page 631).

In the Ada Reference Manual

- 3.10 Access Types\(^{226}\)

15.6.1 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 45: operations_c.h

```ada
void add_one(int *p_i);
```

Listing 46: operations_c.c

```ada
void add_one(int *p_i)
{
    *p_i = *p_i + 1;
}
```

Code block metadata


MD5: 3270f3b2415266a203a6f4c605c3831b

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:

Listing 47: operations.ads

```ada
package Operations is

    procedure Add_One (IA : access Integer)
        with Import, Convention => C;

end Operations;
```

\(^{226}\) http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
Listing 48: show_operations.adb

```ada
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;
```

Code block metadata
MD5: 0219acdbd2dad69962875199ffdd930e

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:

Listing 49: operations.ads

```ada
package Operations is
   procedure Add_One (IA : aliased in out Integer)
      with Import, Convention => C;
end Operations;
```

Listing 50: show_operations.adb

```ada
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);
   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.

15.6. Access parameters
Advanced Journey With Ada: A Flight In Progress

Listing 51: operations_c.h

```c
int * alloc_integer();

void dealloc_integer(int *p_i);

void add_one(int *p_i);
```

Listing 52: 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: ec6dea12d0a948489ccee21b0cc0a1ad2

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:

Listing 53: operations.ads

```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;
```

Code block metadata

MD5: bcbc8a87037b64fc6469e67b928e6172

Note that we're still using an aliased integer type for the Add_One procedure, while we're using access types for the other two subprograms.
Finally, as expected, we can use this specification in a test application:

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

procedure Show_Operations is
  I : access Integer := Alloc_Integer;
begin
  I.all := 42;
  Put_Line (I.all'Image);
  Add_One (I.all);
  Put_Line (I.all'Image);
  Dealloc_Integer (I);
end Show_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\(^\text{227}\)

15.6.2 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:

```
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;
```

\(^{227}\) http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
private

  type T is tagged null record;
  type T_Child is new T with null record;
end Inherited_Primitives;

Listing 56: inherited_primitives.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Inherited_Primitives is
  procedure Proc (N : T_Access) is null;
end Inherited_Primitives;

Listing 57: show_inherited_primitives.adb

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: 8235b21ca9f1f105f533d74d891adfe

Build output

show_inherited_primitives.adb:9:10: error: expected type "T_Access" defined at...
show_inherited_primitives.adb:9:10: error: found type "T_Child_Access" defined at...
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 58: inherited_primitives.ads

package Inherited_Primitives is

(continues on next page)
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\(^{228}\)

15.7 User-Defined References

Implicit dereferencing (page 514) 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 615); and
- indicate the name of the reference discriminant when specifying the Implicit_Dereference aspect.

Let’s see a simple example:

Listing 61: 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:
  --
  -- type Id_Ref (Ref : access Id_Number) is
  --   ^ reference discriminant
  --
  null record
  with Implicit_Dereference => Ref;
  -- ^^^
  -- name of the reference discriminant

  -- Access value:
  --
  I : constant access Id_Number :=
  --   new Id_Number'(Id => 42);

  -- Reference object:
  --
  R : Id_Ref (I);

begin
  Put_Line ("ID: "
              & Positive'Image (R.Id));
  -- ^ Equivalent to:
  --   R REFER.Id
  --   or:
  --   R REFER.all.Id
end Show_User_Defined_Reference;
```

Code block metadata
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;

-- Reference type:
type Id_Ref (Ref : Id_Number_Access) is
  ^ ERROR: it must be
  -- an access
  -- discriminant!
  null record
  with Implicit_Dereference => Ref;
```

However, we could use other forms — such as not null access — in the reference discriminant:

```ada
-- 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

---

229 http://www.ada-auth.org/standards/22rm/html/RM-4-1-5.html
15.7.1 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 62: info.ads

```ada
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 63: info.adb

```ada
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));
  end Show;
end Info;
```

Then, let's declare a reference type and a reference object in the test application:

Listing 64: show_user_defined_reference.adb

```ada
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;
```

(continues on next page)
Advanced Journey With Ada: A Flight In Progress

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.

15.7.2 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:

Listing 65: national_date_info.ads

```ada
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;
  type National_Date_Access is access National_Date;
  procedure Show (Nat_Date : National_Date);
end National_Date_Info;
```

Listing 66: national_date_info.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body National_Date_Info is
  procedure Show (Nat_Date : National_Date) is
    begin
      Put_Line ('Country: ' & Nat_Date.Country);
      Put_Line ('Year: ' & Integer'Image (Nat_Date.Date.Year));
    end Show;
```

15.7. User-Defined References
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:

Listing 67: national_date_containers.ads

```ada
with National_Date_Info; use National_Date_Info;

package National_Date_Containers is

  -- 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;
```

Listing 68: national_date_containers.adb

```ada
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));
        -- ^^^^^^^^^^^^^^^^^^^^^^^^^
        -- Returning reference object with a
        -- reference to the national day we
        -- found.
      end if;
    end loop;
  return
```

(continues on next page)
20 National_Date_Reference'(Ref => null);
   -- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
21 -- Returning reference object with a null  
22 -- reference in case the country wasn't  
23 -- found. This will trigger an exception  
24 -- if we try to dereference it.  
25 end Find;
26 end National_Date_Containers;

Code block metadata

 Defined_References.National_Dates
MD5: ec37ae93a7052c4bc731b2a7be9763ab

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 69: show_national_dates.adb

```ada
with National_Date_Info;
use National_Date_Info;

with National_Date_Containers;
use National_Date_Containers;

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
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:

```ada
1 generic
2   type T is private;
3   type T_Access is access T;
4   type T_Cmp is private;
5   with function Matches (E : T_Access;
6       Elem : T_Cmp)
7       return Boolean;
8 package Generic_Containers is
9       type Ref_Type (Ref : access T) is
10          tagged limited null record
11             with Implicit_Dereference => Ref;
12       type Container is
13          array (Positive range <>) of
14             T_Access;
15       function Find (Cont : Container;
16             Elem : T_Cmp)
17          return Ref_Type;
18 end Generic_Containers;
```

```ada
1 package body Generic_Containers is
2       function Find (Cont : Container;
3             Elem : T_Cmp)
4          return Ref_Type is
5       begin
6          for I in Cont'Range loop
7            if Matches (Cont (I), Elem) then
8               return Ref_Type'(Ref => Cont (I));
9          end if;
10       end loop;
11       return Ref_Type'(Ref => null);
12 end Find;
```

(continues on next page)
When comparing the `Generic_Containers` package to the `National_DateContainers` 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_DateContainers` package as an instance of the `Generic_Containers` package:

```
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_DateContainers is new Generic_Containers (T => National_Date, T_Access => National_Date_Access, T_Cmp => Country_Code, Matches => Matches_Country);

use National_DateContainers;

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;
```
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:

```ada
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;

-- OMITTED

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;

-- OMITTED

function Reference
(Container : aliased in out Vector;
Position : in Cursor)
```
Advanced Journey With Ada: A Flight In Progress

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;

-- OMITTED
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

15.8 Anonymous Access Types and Accessibility Rules

In general, the accessibility rules (page 536) 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 536) to make use of anonymous access types:

Listing 73: library_level.ads

package Library_Level is
    L0_A0 : access Integer;
    L0_Var : aliased Integer;
end Library_Level;

with Library_Level; use Library_Level;

procedure Show_Library_Level is
  L1_Var : aliased Integer;
  L1_AO  : access 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
  L0_AO := L1_AO;  -- legal!!
  -- ^^^^^  -- LEGAL: L1 access object to
  -- L0 access object
  -- ILLEGAL: L1 object
  -- (L1_AO = L1_Var’Access)
  -- to
  -- L0 access object
  --
  -- This is actually OK at compile time,
  -- but the accessibility check fails at
  -- runtime.
end Show_Library_Level;

Chapter 15. Anonymous Access Types
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 \( L_0\_A0 := L_1\_Var'\text{Access} \) is illegal because we're trying to assign to an access object of less deep level.

However, assignment such as \( L_0\_A0 := L_1\_A0 \) 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: \( L_1\_A0 \) 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 \( L_0\_A0 := L_1\_A0 \) fails at runtime because the corresponding access value (\( L_1\_Var'\text{Access} \)) is of a deeper level than \( L_0\_A0 \), which is illegal. (If you comment out the \( L_1\_A0 := L_1\_Var'\text{Access} \) assignment prior to the \( L_0\_A0 := L_1\_A0 \) assignment, this accessibility check doesn't fail anymore.)

### 15.8.1 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 536) and add anonymous access types to it:

#### Listing 75: library_level.ads
```ada
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;
```

#### Listing 76: show_library_level.adb
```ada
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;
-- ^^^^^
-- 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
-- to
-- L0 access object (named type)
L0_IA := L0_Integer_Access (L1_AO);

(continues on next page)
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 target object in the assignment is less deep. This is the case in the $L_0\_AO := L_1\_IA$ and the $L_0\_IA := L_0\_Integer\_Access\,(L_1\_AO)$ assignments. Otherwise, mixing those access objects doesn't impose additional hurdles.

### 15.8.2 Accessibility rules on access parameters

In the previous chapter, we saw that the accessibility rules also apply to access values as subprogram parameters (page 540). 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 77: names.ads

```ada
package Names is
  procedure Show (N : access constant String);
end Show Library Level;
```
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.

### 15.9 Anonymous Access-To-Subprograms

In the previous chapter, we talked about named access-to-subprogram types (page 566). 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:
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 82: add_ten.ads

```ada
procedure Add_Ten (I : in out Integer);
```

Listing 83: add_ten.adb

```ada
procedure Add_Ten (I : in out Integer) is
begin
    I := I + 10;
end Add_Ten;
```

Listing 84: add_twenty.ads

```ada
procedure Add_Twenty (I : in out Integer);
```

Listing 85: add_twenty.adb

```ada
procedure Add_Twenty (I : in out Integer) is
begin
    I := I + 20;
end Add_Twenty;
```
Finally, this is our test application:

```ada
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;
```

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 568). 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 658).

In the Ada Reference Manual

- 3.10 Access Types

15.9.1 Examples of anonymous access-to-subprogram usage

In the section about named access-to-subprogram types (page 566), 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:

```ada
package All_Anonymous_Access_To_Subprogram is

  procedure Proc
    (P : access procedure (I : in out Integer));

```

http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
Advanced Journey With Ada: A Flight In Progress

(continued from previous page)

```ada
-- 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;

private
  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;

-- Anonymous access-to-subprogram as
-- formal parameter:

Proc_T : access procedure
  (Element : in out T);

procedure Gen_Process (Element : in out T);
end All_Anonymous_Access_To_Subprogram;
```

Listing 88: 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))

end All_Anonymous_Access_To_Subprogram;
```

15.9. Anonymous Access-To-Subprograms 651
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
   is
     begin
     I := I * 10;
   end Mult_Ten;

   procedure Mult_Twenty
   is
     begin
     I := I * 20;
   end Mult_Twenty;

end Protected_Integer;

end All_Anonymous_Access_To_Subprogram;

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:

Listing 89: show_anonymous_access_to_subprograms.adb

```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:
  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:
  -- Object (APP) of anonymous
procedure Process_Integer is new
  Gen_Process (T => Integer,
               Proc_T => Add_Twenty'Access);
```

(continues on next page)
-- access-to-protected-subprogram:
PI : Protected Integer;
APP : constant access protected procedure :=
  PI.Mult_Ten'Access;

Some_Int : Integer := 0;
beg
  Put_Line ("Some_Int: " & Some_Int'Image);
  --
  -- Using object of
  -- anonymous access-to-subprogram type:
  --
  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);
  --
  -- Using array with component of
  -- anonymous access-to-subprogram type:
  --
  Put_Line ("Calling procedure from PA array...");
  for I in PA'Range loop
    PA (I) (Some_Int);
    Put_Line ("Some_Int: " & Some_Int'Image);
  end loop;
  Put_Line ("Finished.");
  Put_Line ("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:
  --
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 566), 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.

15.9.2 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 605), which have many drawbacks (page 608) — 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 658). 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.)

15.9.3 Readability

Note that readability suffers if you use a cascade of anonymous access-to-subprograms. For example:

Listing 90: readability_issue.ads

```ada
package Readability_Issue is

  function F
    return access
      function (A : Integer)
        return access
          function (B : Float)
            return Integer;

end Readability_Issue;
```

Listing 91: readability_issue-functions.ads

```ada
package Readability_Issue.Functions is

  function To_Integer (V : Float) return Integer is
    (Integer (V));

  function Select_Conversion
      (A : Integer)
        return access
          function (B : Float)
            return Integer is
              (To_Integer'Access);

end Readability_Issue.Functions;
```
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 93: 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
        Select_Conversion'Access;
  F2 : access function (B : Float)
    return Integer
      := F1 (2);
  I : Integer := F2 (0.1);
begin
  I := F1 (2) (0.1);
end Show_Readability_Issue;
```

Therefore, our recommendation is to avoid this kind of access cascading by carefully designing your application. In general, you won't need that.
15.10 Accessibility Rules and Anonymous Access-To-Subprograms

In principle, the accessibility rules for anonymous access types (page 643) 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 591) 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\(^\text{232}\)

15.10.1 Named vs. anonymous access-to-subprograms

Let’s see an example of a named access-to-subprogram type:

```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;
    begin
        P := Add_One'Access;
    end;
    Show_Access_To_Subprogram_Error;
end Show_Access_To_Subprogram_Error;
```

In this example, we get a compilation error because the lifetime of the Add_One procedure is shorter than the access type PI.

\(^{232}\) http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
In contrast, using an anonymous access-to-subprogram type eliminates the compilation error, i.e. the assignment \( P := \text{Add}_\text{One}' \text{Access} \) becomes legal:

Listing 95: show_access_to_subprogram_error.adb

```ada
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;
```

In this case, the compiler introduces an accessibility check, which fails at runtime because the lifetime of \( \text{Add}_\text{One} \) is shorter than the lifetime of the access object \( P \).

### 15.10.2 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 96: 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 97: access_to_subprogram_types.adb

```ada
package body Access_To_Subprogram_Types is

  procedure Process
  (Arr : in out Integer_Array;
   P : Process_Procedure)
  begin
    P (Arr);
  end Process;

end Access_To_Subprogram_Types;
```

Listing 98: 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;

Code block metadata

  Accessibility_Rules_Anonymous_Access_To_Subprograms.Access_To_Subprogram_Parameter_Named
MD5: 76b70b52a0374fe0fd398024fe869876

Build output

show_access_to_subprogram_error.adb:29:18: error: subprogram must not be deeper
  than access type
show_access_to_subprogram_error.adb:32:18: error: subprogram must not be deeper
  than access type
show_access_to_subprogram_error.adb:34:18: error: subprogram must not be deeper
  than access type
gprbuild: *** compilation phase failed

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 99: access_to_subprogram_types.ads

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;
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;

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;

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;
Arr ( 1): 1
Arr ( 2): 2
Arr ( 3): 3
Add_One...
Arr ( 1): 2
Arr ( 2): 3
Arr ( 3): 4

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.

15.10.3 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 102: data_processing.ads

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 103: data_processing.adb

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
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_Named
MD5: e48e8200e571b62d027753ee96c47fcb

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 104: float_data_processing.ads

with Data_Processing;

package Float_Data_Processing is 
new Data_Processing (Element => Float);
Listing 105: show_access_to_subprograms.adb

```ada
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: 64ee435aac5f2817b7d9cecf538a1e4c

**Build output**

show_access_to_subprograms.adb:19:17: error: subprogram must not be deeper than

<access type>

gprbuild: *** compilation phase failed

Using Display'Access in the call to Iterate triggers a compilation error because its lifetime 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 106: 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));
```

(continues on next page)
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 107: data_processing.adb

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 : 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: fa56595ef1734f2f07ad719c36d6fd8b5

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 108: float_data_processing.ads

```ada
with Data_Processing;

package Float_Data_Processing is
    new Data_Processing (Element => Float);
```

Listing 109: show_access_to_subprograms.adb

```ada
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;
```

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.
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.

16.1 Assignment and equality

Limited types have the following restrictions, which we discussed in the Introduction to Ada 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 1: 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
```

---

236 https://learn.adacore.com/courses/intro-to-ada/chapters/privacy.html#intro-ada-limited-types
With Ada.Text_IO; use Ada.Text_IO;

package body Nonlimited_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 Set (E : Simple_Rec; I : Integer) is
    begin
      E.V.all := I;
    end Set;

  procedure Show (E : Simple_Rec; E_Name : String) is
    begin
      Put_Line (E_Name
        & ".V.all = "
        & Integer'image (E.V.all));
    end Show;
  end 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);");
end Show_Wrong_Assignment_Equality;

Listing 2: nonlimited_types.adb

Listing 3: show_wrong_assignmentEquality.adb
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

### 16.1.1 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

(continues on next page)
```

is
begin
  return E : Simple_Rec do
    E.V := new Integer'(I);
  end return;
end Init;
end Limited_Types;

Listing 6: 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: 019c16f7feca896fd8c37d40d0522dc8

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 677)).

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 7: 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;

16.1. Assignment and equality
Advanced Journey With Ada: A Flight In Progress

Listing 8: limited_types.adb

```ada
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
         To.V.all := From.V.all;
      end Copy;

end Limited_Types;
```

Listing 9: show_limited_assignment.adb

```ada
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;
```

**Code block metadata**

...Equality.Assignment
MD5: 2c017c3592c93be8c19fe247e9241fcb

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 case, 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.

### 16.1.2 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;
```

(continues on next page)
private

    type Simple_Rec is limited record
        V : Integer_Access;
    end record;
end Limited_Types;

Listing 11: 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 12: 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

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 13: limited_types.ads

package Limited_Types is

    type Integer_Access is access Integer;

(continues on next 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 14: 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) return Boolean is
    begin
      -- Comparing record components
      return Left.V.all = Right.V.all;
    end "=";

end Limited_Types;

Listing 15: 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

676 Chapter 16. Limited Types
A /= B

Here, the = 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 re-
placements 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;
```

16.2 Limited private types

As we've seen in code examples from the previous section, we can apply information hiding
(page 35) to limited types. In other words, we can declare a type as limited private
instead of just limited. For example:

```ada
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.

### 16.2.1 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 17: 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 38) 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 18: simple_recs.ads

```ada
package Simple_Recs is

  type Limited_Enumeration is limited private;
  type Limited_Integer is
```

(continues on next page)

---

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.

### 16.2.2 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;
    private
        type Rec is record
            I : Integer;
        end record;
    end Simple_Recs;
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:

```ada
16.2. Limited private types
```
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:

```ada
with Simple_Recs; use Simple_Recs;

procedure Use_Rec_In_Subprogram is
begin
  R1, R2 : Rec;
  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:

```ada
package Simple_Recs is
  type Rec is private;
private
```

(continues on next page)
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 23: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is tagged limited private;
  private
  type Rec is tagged record
    I : Integer;
  end record;
end Simple_Recs;
```

Build output

```
use_rec_in_subprogram.adb:6:06: error: no selector "I" for private type "Rec"
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.
16.2.3 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 24: 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 25: simple_recs.adb

```ada
package body Simple_Recs is

  procedure Copy
    (From : Rec_Limited_Full;
     To : in out Rec_Limited_Full)
  is
    begin
      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
    begin
      To := From;
      -- OK: assignment is allowed because
      -- Rec_Nonlimited_Full is
      -- nonlimited in its full view.
    end Copy;

end Simple_Recs;
```

Code block metadata
Build output

```
simple_recs.adb:8:07: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

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.

### 16.2.4 Limited private component

Another example mentioned by the Ada Reference Manual (7.3.1\(^{239}\), 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:

```
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;

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;
```

239 [http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html](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\(^{240}\)

\(^{240}\) http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html
16.2.5 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 29: 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;
```

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.

16.3 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 30: simple_recs.ads

```ada
package Simple_Recs is
  type Rec is limited record
  I : Integer;
  end record;
end Simple_Recs;
```

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.
The Rec type is also explicitly limited when it's declared limited in the private type's completion (in the package's private part):

```
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 679), 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:

```
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 685) are always explicitly limited types — because, as we've learned before, they cannot have a nonlimited type declaration in its full view.
16.4 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;
```
16.5 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 35: 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 36: 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;
```

As expected, compilations fails because Limited_Integer_Array_2 is a limited (sub)type.
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 37: 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

16.5.1 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 38: simple_recs.ads

```ada
package Simple_Recs is

  type Rec is limited private;

private

  type Rec is limited null record;

end Simple_Recs;
```

**16.5. Deriving from limited types**
Advanced Journey With Ada: A Flight In Progress

Listing 39: 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 40: 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

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.

16.5.2 Deriving from non-explicitly limited private types

Up to this point, we have discussed explicitly limited types (page 685). 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 41: 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

Chapter 16. Limited Types
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:

```
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;
```

```
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;
```

```
-- We copied the code to the
-- Test_Child_Limitedness procedure (in the
-- body of the Simple_Recs.Ext package) and
```

(continues on next 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;
```

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 682), 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 685).

Let's look at an example:

**Listing 45: simple_recs.ads**

```
package Simple_Recs is

  type Tagged_Rec is tagged limited private;

private

  type Tagged_Rec is tagged limited null record;

end Simple_Recs;
```

**Listing 46: simple_recs-ext.ads**

```
package Simple_Recs.Ext is

  type Rec_Derived is new Tagged_Rec with private;
```

(continues on next page)
private

    type Rec_Derived is new
    Tagged_Rec with null record;

end Simple_Recs.Ext;

Listing 47: 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: 81c8a010f093d8823b84bb6e69c4114e

Build output

test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

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 48: 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

16.5. Deriving from limited types
(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:

```ada
package Simple_Recs is
  type Limited_IF is limited interface;
end Simple_Recs;
```

```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;
```

```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;
```

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:

```ada
package Simple_Recs.Ext is
  type Rec_Derived is limited new Limited_IF with private;
```

(continues on next page)
private

    type Rec_Derived is limited new
    Limited_IF with null record;

end Simple_Recs.Ext;

Listing 53: 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: abb295cbfd5ade5f351991c2fbaf519c

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.

16.6 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 679), 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 615) 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 Derived_Immutable is new
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 55: 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;

In the Show_Immutably_Limited_Types package above, we see multiple instances of immutable limited types. (The comments in the source code indicate each type.)

In the Ada Reference Manual

- 7.5 Limited Types246

246 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
16.6.1 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 694), can be nonlimited, so it's not immutably limited.

In the Ada Reference Manual

- 7.3.1 Private Operations
- 7.5 Limited Types

16.7 Limited Types and Discriminants

16.8 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 56: 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;
```

Code block metadata

MD5: 71badd1e38cc4ff37f16d99dd203614b

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 57: simple_recs.ads

```ada
package Simple_Recs is
  type Int_Rec is limited record

(continues on next page)
```

---

247 http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html
248 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
Advanced Journey With Ada: A Flight In Progress

Listing 58: 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\(^{249}\)

### 16.9 Limited types and aggregates

**Note:** This section was originally written by Robert A. Duff and published as Gem #1: Limited Types in Ada 2005\(^{250}\) and Gem #2\(^{251}\).

In this section, we focus on using aggregates to initialize limited types.

Historically


\(^{250}\) [https://www.adacore.com/gems/gem-1](https://www.adacore.com/gems/gem-1)

\(^{251}\) [https://www.adacore.com/gems/gem-2](https://www.adacore.com/gems/gem-2)
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:

Listing 59: 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 60: 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;
```

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.

### 16.9.1 Full coverage rules for limited types

Previously, we discussed full coverage rules for aggregates (page 180). 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:
Listing 61: persons.ads

```ada
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

package Persons is

  type Limited_Person is limited record
    Self : Limited_Person_Access := Limited_Person'Unchecked_Access;
    Name : Unbounded_String;
    Age : Natural;
    Shoe_Size : Positive;
  end record;

end Persons;
```

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:

Listing 62: show_aggregate_box_init.adb

```ada
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
the type, which in this case is \( X \). Thus, we are setting \( X\_\text{Self} \) to be \( \text{X}'\text{Unchecked\_Access} \).

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:

```ada
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;
```

It seems uncomfortable to set the value of \( \text{Self} \) to the wrong value (\( \text{null} \)) and then correct it. It also seems annoying that we have a (correct) default value for \( \text{Self} \), 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 \( \text{Self} => \text{null} \) by \( \text{Self} => <> \).

**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 \texttt{String}, like this:

```ada
procedure Show_String_Box_Init is
  Uninitialized_Const_Str : constant String :=
    (1 .. 10 => <>);
begin
  null;
end Show_String_Box_Init;
```

**Code block metadata**

- MD5: 793ee000fd777d0aa5c15e16132ec411

---

702 Chapter 16. Limited Types
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:

Listing 65: show_dangerous_string.adb

```
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.

### 16.10 Constructor functions for limited types

**Note:** This section was originally written by Robert A. Duff and published as Gem #3²⁵².

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:

²⁵² [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:

with P; use P;

procedure Show_Constructor_FUNCTION is
  My_T : T := Make_T
    (Name => "Bartholomew Cubbins");
begin
null;
end Show_Constructor_Function;

Code block metadata
MD5: 52801fa6bf5af8f6e6e27b

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 69: show_rumplestiltskin_constructor.adb

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;

Code block metadata
MD5: d8d9e9f2e2a0f2f034057fe97f75eacfe

It might help to understand the implementation model: In this case, Rumplestiltskin_Is_My_Name is allocated in the usual way (on the stack, presuming it is declared local to some subprogram). Its address is passed as an extra implicit parameter to Make_Rumplestiltskin, which then passes that same address on to Make_T, which then builds the aggregate in place at that address. Limited objects must never be copied! In this case, Make_T will initialize the Name component, and create the My_Task and My_Prot components, all directly in Rumplestiltskin_Is_My_Name.

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.

16.10. Constructor functions for limited types
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:

```
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;
```

```
package body Aux is
  function Gen_Singleton_Set (Element : OS.Element_Type)
    return OS.Set
  is
    begin
    return S := OS.Empty_Set do
      S.Insert (Element);
      end return;
    end Gen_Singleton_Set;
end Aux;
```

Since Ada 2005, we can say:

```
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);
```

begin
  null;
end Show_Set_Decl;

**Code block metadata**

MD5: e5b6c0e148cfdb1987ab3002ec1f53bd

whether or not Set is limited. This_Set : Set := Empty_Set; seems clearer than:

Listing 73: show_set_decl.adb

```ada
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: e5b6c0e148cfdb1987ab3002ec1f53bd

which might mean "default-initialize to the empty set" or might mean "leave it uninitialized, and we'll initialize it in later".

---

### 16.11 Return objects

#### 16.11.1 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*[^1].

Previously, we discussed extended return statements (page 354). 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 74: task_construct_error.ads

```ada
package Task_Construct_Error is
  task type Task_Type (Discriminant : Integer);
```

[^1]: https://www.adacore.com/gems/ada-gem-10
function Make_Task (Val : Integer)
    return Task_Type;
end Task_Construct_Error;

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;

Listing 76: 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 77: 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
        -- some statements...
    end Make_Task;

(continues on next page)
return Result : Task_Type
    (Discriminant => Val * 3)
    do
        -- some statements...
        null;
    end return;
end Make_Task;
end Task_Construct;

Code block metadata
MD5: c91a24f09a76ae1c25d1a55bcbee910

If we call it like this:

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.

16.11.2 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:

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 80: 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 81: 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));
      end; 
      Put_Line ("----");
      declare
         B : Simple_Rec := Init (0);
      begin
         Put_Line ("B'Address (local): 
            & System.Address_Image (B'Address));
      end; 
      end Show_Limited_Init;
When running this code example and comparing the address of the object \texttt{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 \texttt{E} (of the \texttt{Init} function) and the local object in the \texttt{Show\_Limited\_Init} procedure are the same object.

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:

```ada
package Non_Limited_Types is
  type Integer_Access is access Integer;
  type Simple_Rec is private;
  function Init (I : Integer) return Simple_Rec is private
    type Simple_Rec is record
      V : Integer_Access;
    end record;
end Non_Limited_Types;
```

```ada
with Ada.Text_IO; use Ada.Text_IO; with System; with System.Address_Image;
package body Non_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 do;
end Non_Limited_Types;
```

(continues on next page)
```
end return;
end Init;
end Non_Limited_Types;
```

Listing 84: 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;
```

Code block metadata

Extended_Return_Statements_Limited_Types.Initialization_Return_Copy
MD5: 6e224b64b90dabdf5064c70364fa80cb

Runtime output

```
E'Address (Init): 00007FFDD4ED57B0
A'Address (local): 00007FFDD4ED58D0
----
E'Address (Init): 00007FFDD4ED57B0
B'Address (local): 00007FFDD4ED58D0
```

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.
16.12 Building objects from constructors

Note: This section was originally written by Robert A. Duff and published as Gem #11: Limited Types in Ada 2005\(^{254}\).

We’ve earlier seen examples of constructor functions for limited types similar to this:

```
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;
```

```
package P.Aux is
  function Make_Rumplestiltskin return T;
end P.Aux;
```

\(^{254}\) https://www.adacore.com/gems/ada-gem-11
**Advanced Journey With Ada: A Flight In Progress**

---

**Listing 88: p-aux.adb**

```ada
package body P.Aux is

function Make_Rumplestiltskin return T is
begin
    return Make_T (Name => "Rumplestiltskin");
end Make_Rumplestiltskin;
end P.Aux;
```

---

**Code block metadata**


*Project: From_Constructors.Building_Obs_From_Constructors*

*MD5: 1956721292a82899d244afcd10ff63ed*

It is useful to consider the various contexts in which these functions may be called. We've already seen things like:

**Listing 89: show_rumplestiltskin_constructor.adb**

```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**


*Project: From_Constructors.Building_Obs_From_Constructors*

*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:

**Listing 90: show_parameter_constructor.adb**

```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**


*Project: From_Constructors.Building_Obs_From_Constructors*

*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.
We can allocate initialized objects on the heap:

```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:

```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;
```

16.12. Building objects from constructors
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 93: 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: f7b0c78e9f3e2e104b82dfff25ac3e3a

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 94: 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;
```

16.13 Limited types as parameter

Previously, we saw that parameters can be passed by copy or by reference (page 357). Also, we discussed the concept of by-copy and by-reference types. Explicitly limited types (page 685) 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:

```
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 R s refer to the same object or not:

```
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;
```

```
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;
```

```
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;

Code block metadata

- MD5: d4fe2bb47d2223ef013d22aa305403e5

Runtime output

- R'Address (Proc): 00007FFCAAACE6C
- R'Address (Test): 00007FFCAAACE6C
- R was passed by reference.

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\textsuperscript{255}

\textsuperscript{255} http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
• 6.4.1 Parameter Associations\textsuperscript{256}
• 7.5 Limited Types\textsuperscript{257}

\textsuperscript{256} http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
\textsuperscript{257} http://www.ada-auth.org/standards/22rm/html/RM-7-5.html