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 2025-05

Gustavo A. Hoffmann
and Robert A. Duff

May 31, 2025

CONTENTS

IV Resource Management 591

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:

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

or this:

```
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: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

[1] http://creativecommons.org/licenses/by-sa/4.0

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

In order to use this code example, just follow these steps:

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

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.

[2] https://learn.adacore.com/zip/learning-ada_code.zip
[3] https://learn.adacore.com/courses/intro-to-ada/index.html#intro-ada-course-index

Part I

Data types

CHAPTER

ONE

TYPES

1.1 Names

In simple terms, a "name" can be an identifier, i.e. the *name* that we use to refer to an object or a subprogram, for example. This is what we call a *direct name*. However, in Ada, a name can also refer to other language constructs, as we discuss later on in this section.

In the Ada Reference Manual

- 4.1 Name[4]

1.1.1 Direct names

Direct names are the simplest form of names in Ada. They can be either identifiers or operator symbols.

Identifiers

An identifier — as the term implies — is a (direct) name that we use to *identify* an object, a subprogram, a type, and so on. When specifying an identifier, we aren't limited to ASCII[5] characters: we can use a subset of the Unicode[6] standard.

For further reading...

To be more precise, the Normalization Form KC of the Unicode standard is applied to identifiers. You can find more information about it in the Unicode Standard Annex #15[7].

For example:

Listing 1: show_identifiers.adb

```
procedure Show_Identifiers is
--        ^^^^^^^^^^^^^^^^
--           identifier

   type New_Integer is new
   --    ^^^^^^^^^^^
   --      identifier
     Integer;
```

(continues on next page)

[4] http://www.ada-auth.org/standards/22rm/html/RM-4-1.html
[5] https://en.wikipedia.org/wiki/ASCII
[6] https://en.wikipedia.org/wiki/Universal_Coded_Character_Set
[7] https://unicode.org/reports/tr15/

(continued from previous page)

```
   --  ^^^^^
   --   identifier

   Something_Important : New_Integer;
   --  ^^^^^^^^^^^^^^^
   --  identifier
   --                    ^^^^^^^^^^^
   --                     identifier
begin
  null;
end Show_Identifiers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Names.Identifiers
MD5: e427d3e5fe5f549df593b5e5941cf2ba
```

In this example, we see the following identifiers: `Show_Identifiers` (subprogram), `New_Integer` (type), **`Integer`** (type), and `Something_Important` (object).

Operator symbols

The set of operator symbols that we can use is restricted to the following symbols or reserved words specified in the Ada language:

Operator kind	Operators
Logical operators	**and**, **or**, **xor**
Relational operators	=, /=, <, <= >, >=
Binary adding operators	+, -, &
Unary adding operators	+, -
multiplying opertors	*, /, **mod**, **rem**
Highest precedence operators	**, **abs**, **not**

In the Ada Reference Manual

- 4.5 Operators and Expression Evaluation[8]

1.1.2 Other kinds of names

In addition to direct names, we have the following kinds of names: *explicit dereferences* (page 623), indexed components, slices, selected components, attribute references, *type conversions* (page 49), function calls, character literals, *qualified expressions* (page 68), *generalized references* (page 744), and target name.

Let's see an example of some of them:

Listing 2: show_other_names.adb

```
pragma Ada_2022;

procedure Show_Other_Names is

   type Integer_Access is
```

(continues on next page)

[8] http://www.ada-auth.org/standards/22rm/html/RM-4-5.html

(continued from previous page)

```
     access Integer;

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

   type New_Integer is new
     Integer;

   function Zero
     return New_Integer is
       (0);

   subtype Sub_Integer is
     Integer;

   type Rec is record
      Val : Integer := 0;
   end record;

   type ABC_Enum is
     ('A', 'B', 'C');

   IA  : Integer_Access := new Integer;
   Arr : Integer_Array (1 .. 5) :=
           (others => 0);
   R   : Rec;
   NI  : New_Integer;
   SI  : Sub_Integer;
   E   : ABC_Enum := 'A';
begin
   R.Val := IA.all;
   --        ^^^^^^
   --  explicit dereference

   R.Val := Arr (1);
   --        ^^^^^^^
   --    indexed component

   Arr (1 .. 2) := Arr (3 .. 4);
   --               ^^^^^^^^^^^^
   --                   slice

   Arr (1 .. 2) := (others => R.Val);
   --                          ^^^^^
   --               selected component

   R.Val := Arr'Size;
   --        ^^^^^^^^
   --  attribute reference

   NI := New_Integer (IA.all);
   --     ^^^^^^^^^^^^^^^^^^^^
   --       type conversion

   NI := Zero;
   --     ^^^^
   --  function call

   E  := 'A';
   --     ^^^
   --  character literal
```

(continues on next page)

(continued from previous page)

```

   IA.all := Sub_Integer (R.Val);
   --        ^^^^^^^^^^^^^^^^^^^
   --        qualified expression

   R.Val := @ + 1;
   --       ^
   --   target name
   --
   --   equivalent to:
   --      R.Val := R.Val + 1;

end Show_Other_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Names.Other_Names
MD5: 8063a4c9ff7a01ff7a69454fae096089
```

In this example, we see instances of the following kinds of names:

- explicit dereference: `IA.all`;
- indexed components: `Arr (1)`;
- slices: `Arr (1 .. 2)`, `Arr (3 .. 4)`;
- selected components: `R.Val`;
- attribute references: `Arr'Size`;
- type conversions: `New_Integer (IA.all)`;
- function calls: `Zero`;
- character literals: `'A'`;
- qualified expressions: `Sub_Integer (R.Val)`;
- target name: `@`.

In the Ada Reference Manual

- 4.1 Name[9]
- 4.1.1 Indexed Components[10]
- 4.1.2 Slices[11]
- 4.1.3 Selected Components[12]
- 4.1.4 Attributes[13]
- 4.1.5 User-Defined References[14]
- 4.6 Type Conversions[15]
- 4.7 Qualified Expressions[16]
- 5.2.1 Target Name Symbols[17]

1.2 Objects

The term *object* may be misleading for readers that have a strong background in object-oriented programming. Moreover, its meaning can vary depending on the context. Therefore, it's important to define what we mean by *objects* when focusing on Ada programming.

In computer science, the term object[18] can refer to a piece of data stored in memory — but it can also refer to a table or a form in a database. Also, even when we define the term *object* as data in memory, we can still classify programming languages as object-based[19] or object-oriented[20] languages.

> **Important**
>
> In object-oriented programming, an object belongs to a *class* of objects. In Ada, objects of this kind are called *tagged* objects. Note, however, that we can have objects that don't belong to a class of objects: those are called *untagged* objects.

In the context of Ada programming, an object is an "entity that contains a value, and is either a constant or a variable" — according to the Ada Reference Manual. In other words, any constants or variables that we declare in Ada source code are objects. In addition, there are other examples of objects that don't originate from object declarations:

Listing 3: show_objects.adb

```
procedure Show_Objects is

   type New_Integer is new
     Integer;

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

   procedure Dummy (Obj : Integer)
     is null;
   --               ^^^
   --              object

   task type TT is
      entry Start (Id : Integer);
      --           ^^
      --         object
   end TT;

   task body TT is
   begin
      accept Start (Id : Integer) do
         null;
```

(continues on next page)

9 http://www.ada-auth.org/standards/22rm/html/RM-4-1.html
10 http://www.ada-auth.org/standards/22rm/html/RM-4-1-1.html
11 http://www.ada-auth.org/standards/22rm/html/RM-4-1-2.html
12 http://www.ada-auth.org/standards/22rm/html/RM-4-1-3.html
13 http://www.ada-auth.org/standards/22rm/html/RM-4-1-4.html
14 http://www.ada-auth.org/standards/22rm/html/RM-4-1-5.html
15 http://www.ada-auth.org/standards/22rm/html/RM-4-6.html
16 http://www.ada-auth.org/standards/22rm/html/RM-4-7.html
17 http://www.ada-auth.org/standards/22rm/html/RM-5-2-1.html
18 https://en.wikipedia.org/wiki/Object_(computer_science)
19 https://en.wikipedia.org/wiki/Object-based_language
20 https://en.wikipedia.org/wiki/Object-oriented_programming

(continued from previous page)

```
      end Start;
   end TT;

   function Add_One (V : Integer)
   --                ^
   --    view of an object
                     return Integer is
   begin
      return V + 1;
      --     ^^^^^
      --     object
   end Add_One;

   Arr : Integer_Array (1 .. 10);
   --  ^^^^^^^^^^^^^^^
   --  object

   NI  : New_Integer;
begin
   Arr (1 .. 3) := (others => 1);
   --  ^^^^^^^^
   --  object
   --               ^^^^^^^^^^^^^
   --                  object

   NI := New_Integer (Arr (1));
   --    ^^^^^^^^^^^^^^^^^^^^^
   --      object

   for I in Arr'Range loop
   --  ^
   --  object

      Arr (I) := Add_One (Arr (I));
      --                  ^^^^^^^
      --                  object
   end loop;
end Show_Objects;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_Examples
MD5: edf9eab70ec0ecce90ef71591324ac94
```

As we can see in this code example a formal parameter of a subprogram or an entry is also an object — in addition, so are *value conversions* (page 50), the result returned by a function, the result of evaluating an *aggregate* (page 251), loop parameters, *arrays* (page 299), or the slices of arrays objects, or the components of composite objects.

Other examples of objects include:

- the object created via a *view conversion* (page 58);
- a *dereference* (page 623) of an *access-to-variable* (page 637) value;
- the return object of a function;
- a choice parameter of an exception handler[21].

[21] https://learn.adacore.com/courses/intro-to-ada/chapters/exceptions.html#intro-ada-handling-an-exception

In the Ada Reference Manual

- 3.3 Objects and Named Numbers[22]

1.2.1 Constant and variable objects

Objects can be classified as constant and variable objects. When declaring objects, the distinction is clear:

Listing 4: show_objects.adb

```
procedure Show_Objects is
   Const : constant Integer := 42;
   Var   :          Integer := 0;
begin
   null;
end Show_Objects;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_Declaration_Examples
MD5: 16b4d9546e9c05443ced05c7f6608cc9
```

In this example, `Const` is a constant object, while `Var` is a variable object.

In addition to this, constant objects include:

- the *discriminant component* (page 192) of a variable discriminant;
- a formal parameter or generic formal object of mode **in**.

On the other hand, variable objects include:

- the object created via a *view conversion* (page 58) of a variable;
- a *dereference* (page 623) of an *access-to-variable* (page 637) value.

For example:

Listing 5: show_objects.adb

```
procedure Show_Objects is

   type Device (Id : Positive) is
   record
      Value : Integer;
   end record;

   type Device_Access is
     access all Device;

   Dev : aliased Device (99);
   --                    ^^
   --  Discriminant `Id` is a
   --  constant object.
   --
   --  `Dev` is a variable object,
   --  though.

   Dev_Acc : Device_Access := Dev'Access;

```

(continues on next page)

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

(continued from previous page)

```
   procedure Process (D : Device) is
     null;
   --                  ^
   --         constant object
begin
   Dev.Value := 0;
   --  ^^^^^
   --  variable object

   Dev_Acc.all.Value := 1;
   --  ^^^^^^^
   --  variable object
end Show_Objects;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_Examples
MD5: c0d1386a37e5ed31f0d3163fadbb1b30
```

In this example, we see that `Dev` is a variable object, while its `Id` discriminant is a constant object. In addition, the `Dev_Acc.all` dereference is a variable object. Finally, the **in** parameter of procedure `Process` is a constant object.

In the Ada Reference Manual

- 3.3 Objects and Named Numbers[23]
- 3.3.1 Object Declarations[24]

1.2.2 View of an object

As we've just seen, an object can be either constant or variable. In addition, the *view* of an object is classified as constant or variable as well.

Before we start, note that the classification of an object as constant or variable doesn't directly imply how its view is classified. You may, for example, expect that a constant object has a constant view, but this is not necessarily the case, as we discuss in this section. (In fact, a constant object only has a constant view if it doesn't have a part that has a variable view.)

A part of an object has a variable view if it is of *immutably limited type* (page 805), *controlled type* (page 838), *private type* (page 43), or private extension. In that sense, if any of those parts with variable view exist in a constant object, then we say that the *whole object* has a variable view. Only if a constant object doesn't have *any* parts with variable view, then this object has a constant view.

In contrast, variable objects always have a variable view.

Let's see an example:

Listing 6: devices.ads

```
package Devices is

   type Device_Settings is
   record
      Started : Boolean;
```

(continues on next page)

[23] http://www.ada-auth.org/standards/22rm/html/RM-3-3.html
[24] http://www.ada-auth.org/standards/22rm/html/RM-3-3-1.html

(continued from previous page)

```
   end record;

   type Device (Id : Positive) is
     private;

   function Init (Id : Positive)
                  return Device;

private

   type Device (Id : Positive) is
     null record;

   function Init (Id : Positive)
                  return Device is
      (Device'(Id => Id));

end Devices;
```

Listing 7: show_object_view.adb

```
with Devices; use Devices;

procedure Show_Object_View is
   Dev      : constant Device := Init (5);
   --  Constant object with
   --  variable view.

   Default  : constant Device_Settings
                := (Started => False);
   --  Constant object with
   --  constant view.

   Settings : Device_Settings;

begin
   Settings := (Started => True);
end Show_Object_View;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_View
MD5: b9a56ee937e71c728bac116f21d98742
```

In this example, both `Default_S` and `Dev` are constant objects. However, they have different views: while `Default_S` has a constant view because it doesn't have any parts with variable view, `Dev` has a variable view because it's a private type. Finally, as expected, `Settings` has a variable view because it's a variable object.

1.2.3 Named numbers

In addition to objects, we can have named numbers. Those aren't objects, but rather *names* (page 5) that we assign to numeric values. For example:

Listing 8: show_named_number.adb

```
procedure Show_Named_Number is

   Pi : constant := 3.1415926535;

```

(continues on next page)

(continued from previous page)

```
begin
   null;
end Show_Named_Number;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Named_Number
MD5: ee6808bb7ecb7fef687831f53a8b6668
```

In this example, `Pi` is a named number.

A named number is always known at compilation time. Also, it doesn't have a type associated with it. In fact, its type is called universal real or universal integer — depending on the number being a real or integer number. (In this specific case, `Pi` is a universal real number.) We talk about *universal types* (page 32) later on in this chapter and about *universal real and integer types* (page 366) in another chapter.

In the Ada Reference Manual

- 3.3.2 Number Declarations[25]

1.3 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:

Category	Discrete	Numeric
Enumeration	Yes	No
Integer	Yes	Yes
Real	No	Yes

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:

Category	Attribute	Returned value
Ranges	`First`	First value of the discrete subtype's range.
	`Last`	Last value of the discrete subtype's range.
	`Range`	Range of the discrete subtype (corresponds to `Subtype'First .. Subtype'Last`).
Iterators	`Pred`	Predecessor of the input value.
	`Succ`	Successor of the input value.
Comparison	`Min`	Minimum of two values.
	`Max`	Maximum of two values.
String conversion	`Image`	String representation of the input value.
	`Value`	Value of a subtype based on input string.

We already discussed some of these attributes in the Introduction to Ada course (in the

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

sections about range and related attributes[26] and image attribute[27]). 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[28]

1.3.1 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:

Listing 9: show_first_last_real.adb

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

procedure Show_First_Last_Real is
   subtype Norm is Float range 0.0 .. 1.0;
begin
   Put_Line ("Float'First: " & Float'First'Image);
   Put_Line ("Float'Last:  " & Float'Last'Image);
   Put_Line ("Norm'First:  " & Norm'First'Image);
   Put_Line ("Norm'Last:   " & Norm'Last'Image);
end Show_First_Last_Real;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Ranges_Real_Types
MD5: 89745a94fbdc41a2880ba14e50401acb
```

Runtime output

```
Float'First: -3.40282E+38
Float'Last:   3.40282E+38
Norm'First:   0.00000E+00
Norm'Last:    1.00000E+00
```

This program displays the first and last values of both the **Float** type and the Norm subtype. In the case of the **Float** type, we see the full range, while for the Norm subtype, we get the values we used in the declaration of the subtype (i.e. 0.0 and 1.0).

1.3.2 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:

Listing 10: show_succ_pred_discrete.adb

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

procedure Show_Succ_Pred_Discrete is
   type State is (Idle, Started,
```

(continues on next page)

[26] https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-range-attribute
[27] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-image-attribute
[28] http://www.ada-auth.org/standards/22rm/html/RM-3-5.html

(continued from previous page)

```
                  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);
   Put_Line ("----------");

   Put_Line ("I                 : "
             & I'Image);
   Put_Line ("Integer'Pred (I): "
             & Integer'Pred (I)'Image);
   Put_Line ("Integer'Succ (I): "
             & Integer'Succ (I)'Image);
end Show_Succ_Pred_Discrete;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Show_Succ_Pred_Discrete
MD5: e11d0f50105864fdc1594b3bb72d927e
```

Runtime output

```
State                     : STARTED
State'Pred (Machine_State): IDLE
State'Succ (Machine_State): PROCESSING
----------
I                :  2
Integer'Pred (I):  1
Integer'Succ (I):  3
```

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 11: 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
```

(continues on next page)

(continued from previous page)

```
     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);

      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'First;
   end loop;
   Put_Line ("==================");

   Put_Line ("---- MY_FLOAT ----");
   Put_Line ("----- Succ -------");
   N := My_Float'First;
   for I in 1 .. 5 loop
      Put_Line (N'Image);
      N := My_Float'Succ (N);
   end loop;
   Put_Line ("----- Pred -------");

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

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Show_Succ_Pred_Real
MD5: f426d6539c3ce863101f1e6afb21c08f
```

Runtime output

```
---- DECIMAL -----
Small:  1.00000000000000000E-01
----- Succ -------
 0.0
 0.1
 0.2
 0.3
 0.4
----- Pred -------
 0.5
 0.4
 0.3
 0.2
```

(continues on next page)

(continued from previous page)

```
 0.1
==================
---- MY_FLOAT ----
----- Succ -------
 0.00000E+00
 1.40130E-45
 2.80260E-45
 4.20390E-45
 5.60519E-45
----- Pred -------
 7.00649E-45
 5.60519E-45
 4.20390E-45
 2.80260E-45
 1.40130E-45
```

As the output of the program indicates, the smallest value (see `Decimal'Small` in the example) is used to calculate the previous and next values of `Decimal` type.

In the case of the `My_Float` type, the difference between the current and the previous or next values is 1.40130E-45 (or 2^{-149}) on a standard PC.

1.3.3 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:

Listing 12: show_image_value_attr.adb

```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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Image_Value_Attr
MD5: 9daa13b1f05511fac7e108eb9b8eefa7
```

Runtime output

```
 42
```

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:

Conversion type	Attribute	String type
Conversion to string	`Image`	`String`
	`Wide_Image`	`Wide_String`
	`Wide_Wide_Image`	`Wide_Wide_String`
Conversion to subtype	`Value`	`String`
	`Wide_Value`	`Wide_String`
	`Wide_Wide_Value`	`Wide_Wide_String`

We discuss more about `Wide_String` and `Wide_Wide_String` in *another section* (page 318).

1.3.4 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 13: show_width_attr.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Width_Attr
MD5: 82cff0cf4fecfdecce3020135cf98fd2
```

Runtime output

```
Value:          42
String Length:  3
----
Value:          103
String Length:  4
```

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.

1.3.5 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 14: my_integers.ads

```
package My_Integers is

   subtype One_To_Ten is Integer
     range 1 .. 10;

end My_Integers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Base_Attr
MD5: e3f8310ed742e61a65728fecb6caa557
```

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 + 1 .. 2**15 - 1. In modern 64-bit systems — where wider types such as **Long_Integer** are defined — the range is at least -2**31 + 1 .. 2**31 - 1. Therefore, we could think of the **Integer** type as having the following declaration:

```
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,

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

is really as if we said this:

```
subtype Integer is
  Some_Hardware_Type_With_Sufficient_Range
  range -2 ** 31 .. 2 ** 31 - 1;
```

Since the Some_Hardware_Type_With_Sufficient_Range type is anonymous and we therefore cannot refer to it in the code, we just say that **Integer** is a type rather than a subtype.

Let's focus on signed integers — as the other numerics work the same way. When we declare a signed integer type, we have to specify the required range, statically. If the compiler cannot find a hardware-defined or supported signed integer type with at least the range requested, the compilation is rejected. For example, in current architectures, the code below most likely won't compile:

Listing 15: int_def.ads

```
package Int_Def is

   type Too_Big_To_Fail is
     range -2 ** 255 .. 2 ** 255 - 1;

end Int_Def;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Very_Big_Range
MD5: 29f54776dc814dc8a5d245105b527992
```

Build output

```
int_def.ads:4:06: error: integer type definition bounds out of range
gprbuild: *** compilation phase failed
```

Otherwise, the compiler maps the named Ada type to the hardware "type", presumably choosing the smallest one that supports the requested range. (That's why the range has to be static in the source code, unlike for explicit subtypes.)

The following example shows how the Base attribute affects the bounds of a variable:

Listing 16: show_base.adb

```
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: "
             & One_To_Ten'Last'Image);
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("One_To_Ten'Base'Last: "
             & One_To_Ten'Base'Last'Image);
end Show_Base;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Base_Attr
MD5: ce3e9fb3ff1619e835e9108ae0a787e7
```

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_To_Ten'Base...
One_To_Ten'Last:  10
One_To_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_To_Ten'Succ (One_To_Ten'Last)`. As expected, since the last value of the range doesn't have a successor, a constraint exception is raised.

In the `Using_Base` block, we're declaring a variable `V` of `One_To_Ten'Base` subtype. In this case, the next value exists — because the condition `One_To_Ten'Last + 1 <= One_To_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 constraint errors:

Listing 17: my_integers.ads

```ada
package My_Integers is

   subtype One_To_Ten is Integer range 1 .. 10;

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

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

end My_Integers;
```

Listing 18: my_integers.adb

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

package body My_Integers is

   function Saturate (V : One_To_Ten'Base)
                      return One_To_Ten is
   begin
      --  Put_Line ("SATURATE " & V'Image);
```

(continues on next page)

(continued from previous page)

```

      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 19: show_base.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Base_Attr_Sat
MD5: e9b31345c2efc056bdb71824072852d0
```

Runtime output

```
SATURATE ADD ( 1,  1) =  2
SATURATE ADD ( 10,  8) =  10
SATURATE SUB ( 1,  8) =  1
```

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.

1.4 Enumerations

We've introduced enumerations back in the Introduction to Ada course[29]. 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[30]

1.4.1 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:

Listing 20: days.ads

```
package Days is

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

   --  Essentially, we're declaring
   --  these functions:
   --
   --  function Mon return Day;
   --  function Tue return Day;
   --  function Wed return Day;
   --  function Thu return Day;
   --  function Fri return Day;
   --  function Sat return Day;
```

(continues on next page)

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

[30] http://www.ada-auth.org/standards/22rm/html/RM-3-5-1.html

(continued from previous page)

```
   --  function Sun return Day;

end Days;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_As_Function
MD5: fa3e58b58edffa5a3e04b060a7f8cb8b
```

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:

Listing 21: enumeration_example.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: e2e12bb3bfcb0b6e94769ced9a4b80f9
```

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

Listing 22: show_renaming.adb

```
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");
```

(continues on next page)

(continued from previous page)

```
      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 23: enumeration_example.ads

```
package Enumeration_Example is

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

end Enumeration_Example;
```

Listing 24: enumeration_renaming.ads

```
with Enumeration_Example;

package Enumeration_Renaming is

   subtype Day is Enumeration_Example.Day;

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

end Enumeration_Renaming;
```

Listing 25: show_renaming.adb

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

with Enumeration_Renaming;
use  Enumeration_Renaming;

procedure Show_Renaming is
   D1 : constant Day := Monday;
begin
   Put_Line (Day'Image (D1));
end Show_Renaming;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: 87fe75026f0fc118921eaee45fe55a8a
```

Runtime output

```
MON
```

Note that the call to Put_Line still display Mon instead of Monday.

1.4.2 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:

Listing 26: colors.ads

```
package Colors is

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

   type Primary_Color is
     (Red,
      Green,
      Blue);

end Colors;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Overloading
MD5: b808f90d9164f044b6b7a8931863726f
```

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:

Listing 27: red_colors.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Overloading
MD5: dd590eab88164773e974e748d77a51af
```

Runtime output

```
C1 = Red
C2 = Red
```

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:

Listing 28: colors-shades.ads

```
package Colors.Shades is

   subtype Blue_Shades is
     Colors range Blue .. Darkblue;

end Colors.Shades;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Overloading
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 29: colors.ads

```
package Colors is

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

   type Primary_Color is
     (Red,
      Green,
      Blue);

end Colors;
```

Listing 30: color_loop.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Ambiguities
MD5: 82d0d3f28f1faf6b296a4f44db71f41b
```

Build output

```
color_loop.adb:6:17: error: ambiguous bounds in range of iteration
color_loop.adb:6:17: error: possible interpretations:
color_loop.adb:6:17: error: type "Primary_Color" defined at colors.ads:16
color_loop.adb:6:17: error: type "Color" defined at colors.ads:3
color_loop.adb:6:17: error: ambiguous bounds in discrete range
color_loop.adb:9:30: error: expected type "Color" defined at colors.ads:3
color_loop.adb:9:30: error: found type "Primary_Color" defined at colors.ads:16
gprbuild: *** compilation phase failed
```

Here, it's not clear whether the range in the loop is of `Color` type or of `Primary_Color` type. Therefore, we get a compilation error for this code example. The next line in the code example — the one with the call to `Put_Line` — gives us a hint about the developer's intention to refer to the `Color` type. In this case, we can use qualification — for example, `Color'(Red)` — to resolve the ambiguity:

Listing 31: color_loop.adb

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

procedure Color_Loop is
begin
   for C in Color'(Red) .. Color'(Blue) loop
      Put_Line (Color'Image (C));
   end loop;
end Color_Loop;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Ambiguities
MD5: c3e946d330bb6aed258bcd005a540794
```

Runtime output

```
RED
DARKRED
LIME
FORESTGREEN
GREEN
DARKGREEN
BLUE
```

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

Listing 32: color_loop.adb

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

procedure Color_Loop is
begin
   for C in Color range Red .. Blue loop
      Put_Line (Color'Image (C));
   end loop;
end Color_Loop;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Ambiguities
MD5: 23f8db4fcb5710f7bda6b511234e0448
```

Runtime output

```
RED
DARKRED
LIME
FORESTGREEN
GREEN
DARKGREEN
BLUE
```

Alternatively, `Color range Red .. Blue` could be used in a subtype declaration, so we could rewrite the example above using a subtype (such as Red_To_Blue) in the loop:

Listing 33: color_loop.adb

```
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;
```

1.4.3 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:

Listing 34: days.ads

```
package Days is

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

end Days;
```

Listing 35: show_days.adb

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

```

(continues on next page)

(continued from previous page)

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

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Values
MD5: d6c5cb99b9770893b7277c470f40e805
```

Runtime output

```
MON position      =  0
MON internal code =  0
TUE position      =  1
TUE internal code =  1
WED position      =  2
WED internal code =  2
THU position      =  3
THU internal code =  3
FRI position      =  4
FRI internal code =  4
SAT position      =  5
SAT internal code =  5
SUN position      =  6
SUN internal code =  6
```

Note that this application also displays the internal code, which, in this case, is equivalent to the position value for all enumerations.

We may, however, change the internal code of an enumeration using a representation clause. We discuss this topic *in another section* (page 83).

1.5 Universal and Root Types

Previously, in the section about *scalar types* (page 14), we said that scalar types are the most basic types that we can get. However, Ada has the concept of universal and root types, which could be considered *more basic* than scalar types. In fact, universal and root types are underlying scalar types used by the language designers to define the language semantics. In this section, we briefly introduce this topic.

1.5.1 Universal Types

The Ada standard defines four universal types:

1. universal integer types
2. universal real types
3. universal fixed types
4. universal access types

The first three are numeric types, and we discuss them in detail later on *in another chapter* (page 366). The last one is used for *anonymous access types* (page 711).

Universal types aren't types we can use directly, but rather via specific languages constructs. In this sense, we cannot derive from universal types, but only make use of them indirectly.

For instance, if we declare *named numbers* (page 13) using a real value, we're indirectly using a universal real type. If we declare another named number using an expression, the computation is performed based on the universal types of the elements of that expression:

Listing 36: show_universal_real_integer.ads

```
package Show_Universal_Real_Integer is

   Pi     : constant := 3.1415926535;
   --                   ^^^^^^^^^^^^
   --              universal real type

   Two_Pi : constant := Pi * 2.0;
   --                   ^^^^^^^^
   --                  operation on
   --             universal real type

   N      : constant := 10;
   --                   ^^
   --           universal integer type

   N_10   : constant := N * 10;
   --                   ^^^^^^
   --                 operation on
   --           universal integer type

end Show_Universal_Real_Integer;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Universal_
 ↪Real_Integer
MD5: c9f002461d8ee7f11f2c42a33691f30d
```

In this example, the expression `Pi * 2.0` is computed using universal real types, while the expression `N * 10` is computed using universal integer types.

Similarly, for anonymous access types, the equality operator uses universal access types for the comparison:

Listing 37: show_universal_access.adb

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

procedure Show_Universal_Access is
   I : aliased Integer;
   A : access Integer := I'Access;
   B : access Integer := I'Access;
begin
   if A = B then
      Put_Line ("A = B");
   else
      Put_Line ("A /= B");
   end if;
end Show_Universal_Access;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Universal_
↪Access
MD5: e6a37de980cc3b2c19e36baa3a51c329
```

Runtime output

```
A = B
```

In this example, both A and B are variables of anonymous access types. Because the type isn't a known named type, the equality operation = uses the universal access type for the comparison.

In the Ada Reference Manual

- 3.3.2 Number Declarations[31]
- 4.5.2 Relational Operators and Membership Tests[32]

1.5.2 Root Types

The root types can be found on a level above the universal types. In this category, we can find the same numeric types that we have for universal types, namely the root real, root integer and root fixed types.

The term *root* is used in the context of type derivation. In fact, the root type is the first type that we derive all other types from. In other words, if we declare an integer range as a new type, that type is derived from the root integer type. Similarly, if we declare a new floating-point type, that type is derived from the root real type. For example:

Listing 38: show_root_integer_real.ads

```ada
package Show_Root_Integer_Real is

   type Score is range 0 .. 10;
   --  Type Score is derived from
   --  the root integer type.

   type Real_Score is
     digits 10 range 0.0 .. 10.0;
   --  Type Real_Score is derived from
   --  the root real type.

end Show_Root_Integer_Real;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Root_
↪Integer_Real
MD5: 619ecddeab6fd751cc1af9daa8794a25
```

Here, Score and Real_Score are derived from the root integer and real types, respectively. Note that the derivation is always implicit, as we cannot write something like `type Score is new Root_Integer range 0 .. 10` or `type Real_Score is new Root_Real digits 10 range 0.0 .. 10.0`.

In contrast, if we derive from an existing floating-point or integer type defined by the Ada standard, we're not deriving directly from the root types:

[31] http://www.ada-auth.org/standards/22rm/html/RM-3-3-2.html
[32] http://www.ada-auth.org/standards/22rm/html/RM-4-5-2.html

Listing 39: show_standard_derivation.ads

```
package Show_Standard_Derivation is

   type Score is new Integer
     range 0 .. 10;
   --  Type Score is derived from
   --  the Integer type.

   type Real_Score is new Float
     range 0.0 .. 10.0;
   --  Type Real_Score is derived from
   --  the Float type.

end Show_Standard_Derivation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Standard_
↪Integer_Float_Derivation
MD5: d32261966e1f1ae9626336f57ab16d89
```

In this case, we're explicitly deriving from the standard Ada types **Integer** and **Float**, which, on their turn, are derived from the root integer and root real types, respectively.

For further reading...

You might remember our discussion about the *Base attribute* (page 20) and the fact that it indicates the underlying subtype of a type. We said, for example, that **Integer**'Base gives us the base type of **Integer**, i.e. the the underlying hardware type representing the **Integer** type.

Although the concept of the base type *sounds* similar to the concept of the root type, the focus of each one is different: while the base type refers to the constraints of a type, the root type refers to the derivation tree of a type.

1.6 Definite and Indefinite Subtypes

Indefinite types were mentioned back in the Introduction to Ada course[33]. In this section, we'll recapitulate and extend on both definite and indefinite types.

Definite types are the basic kind of types we commonly use when programming applications. For example, we can only declare variables of definite types; otherwise, we get a compilation error. Interestingly, however, to be able to explain what definite types are, we need to first discuss indefinite types.

Indefinite types include:

- unconstrained arrays;
- record types with unconstrained discriminants without defaults.

Let's see some examples of indefinite types:

Listing 40: unconstrained_types.ads

```
package Unconstrained_Types is

```

(continues on next page)

[33] https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-indefinite-subtype

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
 ↪Indefinite_Types
MD5: e569dc73150b834c9315b14d46c0ac79
```

In this example, both `Integer_Array` and `Simple_Record` are indefinite types.

As we've just mentioned, we cannot declare variable of indefinite types:

Listing 41: using_unconstrained_type.adb

```
with Unconstrained_Types; use Unconstrained_Types;

procedure Using_Unconstrained_Type is

   A : Integer_Array;

   R : Simple_Record;

begin
   null;
end Using_Unconstrained_Type;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
 ↪Indefinite_Types
MD5: 806d4ec64b911a9978ad30fa45a6df10
```

Build output

```
using_unconstrained_type.adb:5:08: error: unconstrained subtype not allowed (need␣
 ↪initialization)
using_unconstrained_type.adb:5:08: error: provide initial value or explicit array␣
 ↪bounds
using_unconstrained_type.adb:7:08: error: unconstrained subtype not allowed (need␣
 ↪initialization)
using_unconstrained_type.adb:7:08: error: provide initial value or explicit␣
 ↪discriminant values
using_unconstrained_type.adb:7:08: error: or give default discriminant values for␣
 ↪type "Simple_Record"
gprbuild: *** compilation phase failed
```

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 42: using_unconstrained_type.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↪Indefinite_Types
MD5: f8e192537f42eea0ebc7873bdaa898f1
```

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 43: show_integer_array.ads

```
with Unconstrained_Types; use Unconstrained_Types;

procedure Show_Integer_Array (A : Integer_Array);
```

Listing 44: show_integer_array.adb

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

procedure Show_Integer_Array (A : Integer_Array)
is
begin
```

(continues on next page)

(continued from previous page)

```
   for I in A'Range loop
      Put_Line (Positive'Image (I)
                & ": "
                & Integer'Image (A (I)));
   end loop;
   Put_Line ("--------");
end Show_Integer_Array;
```

Listing 45: using_unconstrained_type.adb

```
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
 6:  99
 7:  99
 8:  99
 9:  99
 10:  99
--------
```

In this particular example, the compiler doesn't know a priori which range is used for the A parameter of Show_Integer_Array. It could be a range from 1 to 5 as used for variable A_5 of the Using_Unconstrained_Type procedure, or it could be a range from 1 to 10 as used for variable A_10, or it could be anything else. Although the parameter A of Show_Integer_Array is unconstrained, both calls to Show_Integer_Array — in 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 46: show_integer_array_header.ads

```
with Unconstrained_Types; use Unconstrained_Types;

procedure Show_Integer_Array_Header
  (AA : Integer_Array;
   HH : String);
```

Listing 47: show_integer_array_header.adb

```
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 48: using_unconstrained_type.adb

```
with Unconstrained_Types; use Unconstrained_Types;

with Show_Integer_Array_Header;

procedure Using_Unconstrained_Type is
   A_5  : constant Integer_Array (1 .. 5)  :=
            (1, 2, 3, 4, 5);
   A_10 : constant Integer_Array (1 .. 10) :=
            (1, 2, 3, 4, 5, others => 99);
begin
   Show_Integer_Array_Header (A_5,
                              "First example");
   Show_Integer_Array_Header (A_10,
                              "Second example");
end Using_Unconstrained_Type;
```

Code block metadata

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

(continues on next page)

(continued from previous page)

```
 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 49: show_integer_array_plus.ads

```
with Unconstrained_Types; use Unconstrained_Types;

procedure Show_Integer_Array_Plus
  (A : Integer_Array;
   V : Integer);
```

Listing 50: show_integer_array_plus.adb

```
with Show_Integer_Array;

procedure Show_Integer_Array_Plus
  (A : Integer_Array;
   V : Integer)
is
   A_Plus : Integer_Array (A'Range);
begin
   for I in A_Plus'Range loop
      A_Plus (I) := A (I) + V;
   end loop;
   Show_Integer_Array (A_Plus);
end Show_Integer_Array_Plus;
```

Listing 51: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↪Indefinite_Types
MD5: e58ae62272ff0b27c5f6e171c88a6880
```

Runtime output

```
 1:  6
 2:  7
 3:  8
 4:  9
 5:  10
--------
```

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[34]

1.7 Incomplete types

Incomplete types — as the name suggests — are types that have missing information in their declaration. This is a simple example:

```
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:

Listing 52: incomplete_type_example.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Incomplete_Types.Incomplete_Types
MD5: 5ca250595f2b0cc101df286ab319982f
```

[34] http://www.ada-auth.org/standards/22rm/html/RM-3-3.html

The first declaration of type R is incomplete. However, in the second declaration of R, we specify that R is a record. By providing this missing information, we're completing the type declaration of R.

It's also possible to declare an incomplete type in the private part of a package specification and its complete form in the package body. Let's rewrite the example above accordingly:

Listing 53: incomplete_type_example.ads

```
package Incomplete_Type_Example is

private

   type R;
   --  Incomplete type declaration!

end Incomplete_Type_Example;
```

Listing 54: incomplete_type_example.adb

```
package body Incomplete_Type_Example is

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

end Incomplete_Type_Example;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Incomplete_Types.Incomplete_Types_2
MD5: fd2f0301b4a63887add1cb2093692ddb
```

A typical application of incomplete types is to create linked lists using *access types* (page 593) based on those incomplete types. This kind of type is called a recursive type. For example:

Listing 55: linked_list_example.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Incomplete_Types.Linked_List_Example
MD5: b2d3a048473d498bbe691bc6e38ca1e9
```

Here, the N component of `Integer_List` is essentially giving us access to the next element of `Integer_List` type. Because the `Next` type is both referring to the `Integer_List` type and being used in the declaration of the `Integer_List` type, we need to start with an incomplete declaration of the `Integer_List` type and then complete it after the declaration of Next.

Incomplete types are useful to declare *mutually dependent types* (page 181), as we'll see later on. Also, we can also have formal incomplete types, as we'll discuss later.

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations[35]

1.8 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:

Listing 56: full_view.ads

```
package Full_View is

   --  Full view of the R type:
   type R is record
      I : Integer;
   end record;

end Full_View;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Full_View
MD5: d37792287d08f9aa3d32499e233516df
```

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 57: partial_full_views.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Partial_Full_View
MD5: b0cf748e43b23ea6c845e283c4266ff3
```

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

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 58: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Partial_Full_View
MD5: 05bc9a75406d0a46f6d009d97885d010
```

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:

Listing 59: limited_private_example.ads

```ada
package Limited_Private_Example is

   --  Partial view must be limited,
   --  since the full view is limited.
   type R is limited private;

private

   type R is limited record
      I : Integer;
   end record;

end Limited_Private_Example;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Limited_Private
MD5: 23d01b93fe052a500c8ca6ff76a2fd51
```

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:

Listing 60: tagged_full_view_example.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Tagged_Full_View
MD5: 0ff9142b1ee086695b98b72a9d0f50ac
```

In this case, from a user's perspective, the R type is non-tagged, so that users cannot use any object-oriented programming features for this type. In the package body of Tagged_Full_View_Example, however, this type is tagged, so that all object-oriented programming features are available for subprograms of the package body that make use of this type. Again, the partial view of the private type contains the most important information for users that want to declare objects of this type.

In the Ada Reference Manual

- 7.3 Private Types and Private Extensions[36]

1.8.1 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 61: 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;
```

Code block metadata

[36] http://www.ada-auth.org/standards/22rm/html/RM-7-3.html

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: f1fcbed95e0f66a6f67d1bfd9ba9df1c
```

This code compiles as expected, but isn't very useful. We can improve it by adding operators to it, for example:

Listing 62: 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 63: private_integers.adb

```
package body Private_Integers is

   function "+" (Left, Right : Private_Integer)
                 return Private_Integer
   is
      Res : constant Integer :=
              Integer (Left) + Integer (Right);
      --  Note that we're converting Left
      --  and Right to Integer, which calls
      --  the "+" operator of the Integer
      --  type. Writing "Left + Right" would
      --  have called the "+" operator of
      --  Private_Integer, which leads to
      --  recursive calls, as this is the
      --  operator we're currently in.
   begin
      return Private_Integer (Res);
   end "+";

end Private_Integers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: ac161cb5debfde16465c45949cf682d7
```

Now, let's use the new operator in a test application:

Listing 64: 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 [-
↪gnatwv]
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 65: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Array
MD5: b873c2d381c159532b429101e4533c05
```

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 66: private_arrays.ads

```
package Private_Arrays is

   type Private_Constrained_Array
     (L : Positive) is private;

private

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

   type Private_Constrained_Array
     (L : Positive) is
   record
      Arr : Integer_Array  (1 .. 2);
   end record;

end Private_Arrays;
```

Listing 67: 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
```

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 68: private_arrays.ads

```
package Private_Arrays is

   type Private_Constrained_Array (<>) is
     private;

   function Init
     (L : Positive)
      return Private_Constrained_Array;

private

```

(continues on next page)

(continued from previous page)

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

end Private_Arrays;
```

Listing 69: private_arrays.adb

```
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;
```

Listing 70: declare_private_array.adb

```
with Private_Arrays; use Private_Arrays;

procedure Declare_Private_Array is
  Arr : Private_Constrained_Array := Init (5);
begin
  null;
end Declare_Private_Array;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Array
MD5: cd170a1e44fffb93314776a68f1cb413
```

Build output

```
private_arrays.adb:7:07: warning: variable "PCA" is read but never assigned [-
↪gnatwv]
```

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.

1.9 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[37]. 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 427);
- in a view conversion, the operand is evaluated as a name.

[37] https://learn.adacore.com/courses/intro-to-ada/chapters/strongly_typed_language.html#intro-ada-type-conversion

In other words, we cannot use expressions such as `2 * A` in a view conversion, but only A. In a value conversion, we could use both forms.

In the Ada Reference Manual

- 4.6 Type Conversions[38]

1.9.1 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 71: custom_integers.ads

```
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 72: show_conversion.adb

```
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 :     "
```

(continues on next page)

38 http://www.ada-auth.org/standards/22rm/html/RM-4-6.html

(continued from previous page)

```
              & Int_Var'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Root_Derived_Type_
↪Conversion
MD5: 7cd324f308edc34de3bc4bccce63f1ee
```

Runtime output

```
Integer_Var :  1
Integer_Var :  2
Int_Var :      4
```

In the Show_Conversion procedure from this example, we first convert from Int to **Integer**. Then, we do the same conversion while providing the resulting value as an actual parameter for the Double function. Finally, we convert the Integer_Var * 2 expression from **Integer** to Int.

Note that the converted value must conform to any constraints that the target type might have. In the example above, Int has a predicate that dictates that its value cannot be zero. This (dynamic) predicate is checked at runtime, so an exception is raised if it fails:

Listing 73: show_conversion.adb

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

procedure Show_Conversion is
   Int_Var     : Int;
   Integer_Var : Integer;
begin
   Integer_Var := 0;
   Int_Var     := Int (Integer_Var);

   Put_Line ("Int_Var : "
             & Int_Var'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Root_Derived_Type_
↪Conversion
MD5: 4150cdffd4c1fed39fa1728a77fa599f
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_
↪conversion.adb:9
```

In this case, the conversion from **Integer** to Int fails because, while zero is a valid integer value, it doesn't obey Int's predicate.

Numeric type conversion

A typical conversion is the one between integer and floating-point values. For example:

Listing 74: show_conversion.adb

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

procedure Show_Conversion is
   F : Float   := 1.0;
   I : Integer := 2;
begin
   I := Integer (F);

   Put_Line ("I : "
             & I'Image);

   I := 4;
   F := Float (I);

   Put_Line ("F :    "
             & F'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Numeric_Type_
↪Conversion
MD5: f64649c786377617b0bc9ff49475ba55
```

Runtime output

```
I :  1
F :    4.00000E+00
```

Also, we can convert between fixed-point types and floating-point or integer types:

Listing 75: fixed_point_defs.ads

```
package Fixed_Point_Defs is
   S     : constant := 32;
   Exp   : constant := 15;
   D     : constant := 2.0 ** (-S + Exp + 1);

   type TQ15_31 is delta D
     range -1.0 * 2.0 ** Exp ..
            1.0 * 2.0 ** Exp - D;

   pragma Assert (TQ15_31'Size = S);
end Fixed_Point_Defs;
```

Listing 76: show_conversion.adb

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

procedure Show_Conversion is
   F  : Float;
   FP : TQ15_31;
   I  : Integer;
begin
   FP := TQ15_31 (10.25);
   I  := Integer (FP);

   Put_Line ("FP : "
```

(continues on next page)

(continued from previous page)

```
               & FP'Image);
   Put_Line ("I : "
               & I'Image);

   I  := 128;
   FP := TQ15_31 (I);
   F  := Float (FP);

   Put_Line ("FP : "
               & FP'Image);
   Put_Line ("F :    "
               & F'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Numeric_Type_
↪Conversion
MD5: 70714ba396b03469397b982e00299561
```

Runtime output

```
FP :  10.25000
I :  10
FP :  128.00000
F :    1.28000E+02
```

As we can see in the examples above, converting between different numeric types works in all directions. (Of course, rounding is applied when converting from floating-point to integer types, but this is expected.)

Enumeration conversion

We can also convert between an enumeration type and a type derived from it:

Listing 77: custom_enumerations.ads

```
package Custom_Enumerations is

   type Priority is (Low, Mid, High);

   type Important_Priority is new
     Priority range Mid .. High;

end Custom_Enumerations;
```

Listing 78: show_conversion.adb

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

procedure Show_Conversion is
   P  : Priority           := Low;
   IP : Important_Priority := High;
begin
   P := Priority (IP);

   Put_Line ("P:  "
               & P'Image);

```

(continues on next page)

(continued from previous page)

```
   P  := Mid;
   IP := Important_Priority (P);

   Put_Line ("IP: "
             & IP'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Enumeration_Type_
↪Conversion
MD5: b1e42cbd8b57291d3b3a9968c41efdd7
```

Runtime output

```
P:  HIGH
IP: MID
```

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 79: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Enumeration_Type_
↪Conversion
MD5: 6bbc777d4b44023bf572ca5dc6c2b4f8
```

Build output

```
show_conversion.adb:9:10: warning: value not in range of type "Important_Priority"␣
↪defined at custom_enumerations.ads:5 [enabled by default]
show_conversion.adb:9:10: warning: Constraint_Error will be raised at run time␣
↪[enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_conversion.adb:9 range check failed
```

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

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:

Listing 80: custom_arrays.ads

```
package Custom_Arrays is

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

   type Derived_Integer_Array is new
     Integer_Array;

end Custom_Arrays;
```

Listing 81: show_conversion.adb

```
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: 72cdf4850bec78893b6985b0c7ef02b9
```

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 82: 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 83: 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 84: show_conversion.adb

```
with Ada.Text_IO;     use Ada.Text_IO;
with Custom_Arrays_1; use Custom_Arrays_1;
with Custom_Arrays_2; use Custom_Arrays_2;

procedure Show_Conversion is
   subtype Common_Range is Positive range 1 .. 3;

   AI_1 : Integer_Array_1 (Common_Range);
   AI_2 : Integer_Array_2 (Common_Range);
   AF_1 : Float_Array_1 (Common_Range);
   AF_2 : Float_Array_2 (Common_Range);
begin
   AI_2 := [1, 2, 3];
   AI_1 := Integer_Array_1 (AI_2);

   Put_Line ("AI_1: "
             & AI_1'Image);

   AI_1 := [4, 5, 6];
   AI_2 := Integer_Array_2 (AI_1);

   Put_Line ("AI_2: "
             & AI_2'Image);

   --  ERROR: Cannot convert arrays whose
   --         components have different types:
```

(continues on next page)

(continued from previous page)

```
   --
   --  AF_1 := Float_Array_1 (AI_1);
   --
   --  Instead, use array aggregate where each
   --  component is converted from integer to
   --  float:
   --
   AF_1 := [for I in AF_1'Range =>
             Float (AI_1 (I))];

   Put_Line ("AF_1: "
            & AF_1'Image);

   AF_2 := Float_Array_2 (AF_1);

   Put_Line ("AF_2: "
            & AF_2'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Array_Type_
↪Conversion
MD5: 42b89fa5fe1f20af26b5da4586cea8e8
```

Runtime output

```
AI_1:
[ 1,  2,  3]
AI_2:
[ 4,  5,  6]
AF_1:
[ 4.00000E+00,  5.00000E+00,  6.00000E+00]
AF_2:
[ 4.00000E+00,  5.00000E+00,  6.00000E+00]
```

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 266).)

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 85: custom_arrays.ads

```
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 86: show_conversion.adb

```
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: f5348b3bed5cbd93dab44394358e1ce6
```

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 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.

1.9.2 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:

Listing 87: some_tagged_types.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Types_View
MD5: 2e18ba972682f1ae1d38e38842fde48e
```

Here, `Obj` is an object of type `T_Derived`. When we *view* this object, we notice that it has two components: A and B. However, we could *view* this object as being of type T. From that perspective, this object only has one component: A. (Note that changing the perspective doesn't change the object itself.) Therefore, a view conversion from `T_Derived` to T just makes us *view* the object `Obj` from a different angle.

In this sense, a view conversion changes the view of a given object to the target type's view, both in terms of components that exist and operations that are available. It doesn't really change anything at all in the value itself.

There are basically two kinds of view conversions: the ones using tagged types and the ones using untagged types. We discuss these kinds of conversion in this section.

View conversion of tagged types

A conversion between tagged types is a view conversion. Let's consider a typical code example that declares one, two and three-dimensional points:

Listing 88: points.ads

```
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);
```

(continues on next page)

(continued from previous page)

```

end Points;
```

Listing 89: points.adb

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

package body Points is

   procedure Display (P : Point_1D) is
   begin
      Put_Line ("(X => " & P.X'Image & ")");
   end Display;

   procedure Display (P : Point_2D) is
   begin
      Put_Line ("(X => " & P.X'Image
                & ", Y => " & P.Y'Image & ")");
   end Display;

   procedure Display (P : Point_3D) is
   begin
      Put_Line ("(X => " & P.X'Image
                & ", Y => " & P.Y'Image
                & ", Z => " & P.Z'Image & ")");
   end Display;

end Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Type_
↪Conversion
MD5: 0acc05ae2310ab4ba038dfdb6bae0495
```

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

Listing 90: show_conversion.adb

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

procedure Show_Conversion is
   P_1D : Point_1D;
   P_3D : Point_3D;
begin
   P_3D := (X => 0.1, Y => 0.5, Z => 0.3);
   P_1D := Point_1D (P_3D);

   Put ("P_3D : ");
   Display (P_3D);

   Put ("P_1D : ");
   Display (P_1D);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Type_
↪Conversion
MD5: fb8e07c8f2399cfae935179d8f413150
```

Runtime output

```
P_3D : (X =>  1.00000E-01, Y =>  5.00000E-01, Z =>  3.00000E-01)
P_1D : (X =>  1.00000E-01)
```

In this example, as expected, we're able to convert from the Point_3D type (which has three components) to the Point_1D type, which has only one component.

View conversion of untagged types

For untagged types, a view conversion is the one that happens when we have an object of an untagged type as an actual parameter for a formal **in out** or **out** parameter.

Let's see a code example. Consider the following simple procedure:

Listing 91: double.ads

```
procedure Double (X : in out Float);
```

Listing 92: 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:

Listing 93: show_conversion.adb

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

procedure Show_Conversion is
   I : Integer;
begin
   I := 2;
   Put_Line ("I : "
             & I'Image);

   --  Calling Double with
   --  Integer parameter:
   Double (Float (I));
   Put_Line ("I : "
             & I'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_
↪Conversion
MD5: 2256d3c120d569789dcd4c9959ed9d0f
```

Runtime output

```
I :  2
I :  4
```

In this case, the `Float (I)` conversion in the call to `Double` creates a temporary floating-point variable. This is the same as if we had written the following code:

Listing 94: show_conversion.adb

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

procedure Show_Conversion is
   I : Integer;
begin
   I := 2;
   Put_Line ("I : "
             & I'Image);

   declare
      F : Float := Float (I);
   begin
      Double (F);
      I := Integer (F);
   end;
   Put_Line ("I : "
             & I'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_
↪Conversion
MD5: 3b90caf789952710ece42141a7b60968
```

Runtime output

```
I :  2
I :  4
```

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.

1.9.3 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 95: 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 96: show_conversion.adb

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

procedure Show_Conversion is
   Int_Var       : Int;
   Sub_Int_1_Var : Sub_Int_1;
   Sub_Int_2_Var : Sub_Int_2;
   Integer_Var   : Integer;
begin
   Integer_Var := 5;
   Int_Var     := Int (Integer_Var);

   Put_Line ("Int_Var :       "
             & Int_Var'Image);

   --  Implicit conversions:
   --  no explicit conversion required!
   Sub_Int_1_Var := Integer_Var;
   Sub_Int_2_Var := Integer_Var;

   Put_Line ("Sub_Int_1_Var : "
             & Sub_Int_1_Var'Image);
   Put_Line ("Sub_Int_2_Var : "
             & Sub_Int_2_Var'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Implicit_Subtype_
↪Conversion
MD5: dbbe498fa66701ca94f48119b1bc1a91
```

Runtime output

```
Int_Var :        5
Sub_Int_1_Var :  5
Sub_Int_2_Var :  5
```

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.

1.9.4 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 97: custom_rec.ads

```
package Custom_Rec is

   type Rec is record
      X : Integer;
   end record;

   function To_Integer (R : Rec)
                        return Integer is
     (R.X);

end Custom_Rec;
```

Listing 98: show_conversion.adb

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

procedure Show_Conversion is
   R : Rec;
   I : Integer;
begin
   R := (X => 2);
   I := To_Integer (R);

   Put_Line ("I : " & I'Image);
end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Other_Type_
 ↪Conversions
MD5: d52a4fde48243a7dd6942f0b2b91ce62
```

Runtime output

```
I :  2
```

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:
>
> ```
> #include <iostream>
>
> class T1 {
> public:
>     T1 (float x) :
>       x(x) {}
>
>     // If class T3 is declared before class
>     // T1, we can overload the "=" operator.
>     //
>     // void operator=(T3 v) {
>     //     x = static_cast<float>(v);
>     // }
> ```

```
    void display();
private:
   float x;
};

class T3 {
public:
    T3 (float x, float y, float z) :
      x(x), y(y), z(z) {}

    // implicit conversion
    operator float() const {
        return (x + y + z) / 3.0;
    }

    // implicit conversion
    //
    // operator T1() const {
    //     return T1((x + y + z) / 3.0);
    // }

    // explicit conversion (C++11)
    explicit operator T1() const {
        return T1(float(*this));
    }

    void display();

private:
    float x, y, z;
};

void T1::display()
{
    std::cout << "(x => " << x
              << ")" << std::endl;
}

void T3::display()
{
    std::cout << "(x => " << x
              << "y => "  << y
              << "z => "  << z
              << ")" << std::endl;
}

int main ()
{
    const T3 t_3 (0.5, 0.4, 0.6);
    T1 t_1 (0.0);
    float f;

    // Implicit conversion
    f = t_3;

    std::cout << "f : " << f
              << std::endl;

    // Explicit conversion
    f = static_cast<float>(t_3);
```

```
    // f = (float)t_3;

    std::cout << "f : " << f
              << std::endl;

    // Explicit conversion
    t_1 = static_cast<T1>(t_3);

    // t_1 = (T1)t_3;

    std::cout << "t_1 : ";
    t_1.display();
    std::cout << std::endl;
}
```

Here, we're using `operator float()` and `operator T1()` to cast from an object of class T3 to a floating-point value and an object of class T1, respectively. (If we switch the order and declare the T3 class before the T1 class, we could overload the = operator, as you can see in the commented-out lines.)

In Ada, this kind of conversions isn't available. Instead, we have to implement conversion functions such as the `To_Integer` function from the previous code example. This is the corresponding implementation:

Listing 99: custom_defs.ads

```ada
package Custom_Defs is

   type T1 is private;

   function Init (X : Float)
                  return T1;

   procedure Display (Obj : T1);

   type T3 is private;

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

   function To_Float (Obj : T3)
                      return Float;

   function To_T1 (Obj : T3)
                   return T1;

   procedure Display (Obj : T3);

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 100: custom_defs.adb

```
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
      Put_Line ("(X => "     & Obj.X'Image
                & ", Y => " & Obj.Y'Image
                & ", Z => " & Obj.Z'Image
                & ")");
   end Display;

end Custom_Defs;
```

Listing 101: show_conversion.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Explicit_Rec_
↪Conversion
MD5: b3e7be5488fb8026b4386063ba16aaeb
```

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.)

1.10 Qualified Expressions

We already saw qualified expressions in the Introduction to Ada[39] 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 102: simple_integers.ads

```
package Simple_Integers is

   type Int is new Integer;

   subtype Int_Not_Zero is Int
     with Dynamic_Predicate => Int_Not_Zero /= 0;

end Simple_Integers;
```

Listing 103: show_qualified_expressions.adb

```
with Simple_Integers; use Simple_Integers;

procedure Show_Qualified_Expressions is
   I : Int;
begin
   --  Using qualified expression Int'(N)
   I := Int'(0);
end Show_Qualified_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example
MD5: 0a83e10b51c72827e322984bd5c8009d
```

Here, the qualified expression Int'(0) indicates that the value zero is of Int type.

In the Ada Reference Manual

- 4.7 Qualified Expressions[40]

1.10.1 Verifying subtypes

Note

This feature was introduced in Ada 2022.

[39] https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_types.html#intro-ada-qualified-expressions
[40] http://www.ada-auth.org/standards/22rm/html/RM-4-7.html

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

Listing 104: show_qualified_expressions.adb

```
with Simple_Integers; use Simple_Integers;

procedure Show_Qualified_Expressions is
   I : Int;
begin
   I := Int_Not_Zero'(0);
end Show_Qualified_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example
MD5: 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 ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_qualified_
↪expressions.adb:6
```

Here, the qualified expression `Int_Not_Zero'(0)` checks the dynamic predicate of the subtype. (This predicate check fails at runtime.)

1.11 Default initial values

In the Introduction to Ada course[41], we've seen that record components can have default values. For example:

Listing 105: defaults.ads

```
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:

[41] https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-default-values

Listing 106: 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
```

Note that we cannot specify a default value for a subtype:

Listing 107: defaults.ads

```ada
package Defaults is

   subtype T is Integer
     with Default_Value => -1;
   --  ERROR!!

end Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_3
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:

Listing 108: defaults.ads

```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:

Listing 109: defaults.ads

```ada
package Defaults is

```

(continues on next page)

(continued from previous page)

```
   type E is (E1, E2, E3)
     with Default_Value => E1;

   type T is new Integer
     with Default_Value => -1;

   --  We cannot specify default
   --  values for subtypes:
   --
   --  subtype T is Integer
   --    with Default_Value => -1;

   type R is record
     X : Positive := 1;
     Y : Positive := 10;
   end record;

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

end Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults
MD5: e9263ff5b96523c129a3d2d9bbb5a4dd
```

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

Listing 110: use_defaults.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults
MD5: f8e55d31cbda2447fe14eb07eaad1975
```

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[42]
- 3.6 Array Types[43]

1.12 Deferred Constants

Deferred constants are declarations where the value of the constant is not specified immediately, but rather *deferred* to a later point. In that sense, if a constant declaration is deferred, it is actually declared twice:

1. in the deferred constant declaration, and
2. in the full constant declaration.

The simplest form of deferred constant is the one that has a full constant declaration in the private part of the package specification. For example:

Listing 111: deferred_constants.ads

```
package Deferred_Constants is

   type Speed is new Long_Float;

   Light : constant Speed;
   --      ^ deferred constant declaration

private

   Light : constant Speed := 299_792_458.0;
   --      ^ full constant declaration

end Deferred_Constants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
↪Constant_Private
MD5: f76e42326889f70fa7e1e216576f9771
```

Another form of deferred constant is the one that imports a constant from an external implementation — using the Import keyword. We can use this to import a constant declaration from an implementation in C. For example, we can declare the light constant in a C file:

[42] http://www.ada-auth.org/standards/22rm/html/RM-3-5.html
[43] http://www.ada-auth.org/standards/22rm/html/RM-3-6.html

Listing 112: constants.c

```c
double light = 299792458.0;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
↪Constant_C
MD5: 71194a329dc5adaac3e01aff143a9943
```

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

Listing 113: deferred_constants.ads

```ada
package Deferred_Constants is

   type Speed is new Long_Float;

   Light : constant Speed with
     Import, Convention => C;
   --  ^^^^ deferred constant
   --       declaration; imported
   --       from C file

end Deferred_Constants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
↪Constant_C
MD5: 9355d194e973c6c6540485178b2259c9
```

In this case, we don't have a full declaration in the `Deferred_Constants` package, as the `Light` constant is imported from the `constants.c` file.

As a rule, the deferred and the full declarations should match — except, of course, for the actual value that is missing in the deferred declaration. For instance, we're not allowed to use different types in both declarations. However, we may use a subtype in the full declaration — as long as it's compatible with the type that was used in the deferred declaration. For example:

Listing 114: deferred_constants.ads

```ada
package Deferred_Constants is

   type Speed is new Long_Float;

   subtype Positive_Speed is
     Speed range 0.0 .. Speed'Last;

   Light : constant Speed;
   --      ^ deferred constant declaration

private

   Light : constant Positive_Speed :=
             299_792_458.0;
   --      ^ full constant declaration
   --        using a subtype

end Deferred_Constants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
↪Constant_Subtype
MD5: ad6e13e30bacb6d97ccfa6c7345ffb67
```

Here, we're using the Speed type in the deferred declaration of the Light constant, but we're using the Positive_Speed subtype in the full declaration.

A useful application of deferred constants is when the value of the constant is calculated using entities not meant to be compile-time visible to clients. As such, these other entities are only visible in the private part of the package, so that's where the value of the deferred constant must be computed. For example, the full constant declaration may be computed by a call to an expression function:

Listing 115: deferred_constants.ads

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
↪Constant_Function
MD5: f0b1a9521af31a4b48bbd54891f1c32b
```

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[44]

1.13 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

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

an integer literal:

Listing 116: 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

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Simple_
↪Integer_Literal
MD5: 9f65e7c319be2b292dc1fdf02dd7cfb4
```

Runtime output

```
 10
```

Here, `10` is the integer literal that we use to initialize the integer variable V. Other examples of literals are real literals and string literals, as we'll see later.

When we declare an enumeration type, we limit the set of literals that we can use to initialize objects of that type:

Listing 117: simple_enumeration.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Simple_
↪Enumeration
MD5: 075df146fcb567817dadfdb245659773
```

Runtime output

```
ON
```

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:

Literal	Example	Aspect
Integer	1	Integer_Literal
Real	1.0	Real_Literal
String	"On"	String_Literal

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 118: 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

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.User_Defined_
↪Literals
MD5: 37c497105ea3a5ad67f72955911eb31a
```

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 119: 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;
```

(continues on next page)

(continued from previous page)

```

end Activation_States;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.User_Defined_
↪Literals
MD5: c130c42ee2b91e4306c0b49bd6d5d322
```

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

Listing 120: activation_states.ads

```ada
package Activation_States is

   type Activation_State is (Unknown, Off, On)
     with String_Literal  =>
            To_Activation_State,
          Integer_Literal =>
            Integer_To_Activation_State,
          Real_Literal    =>
            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;
```

Listing 121: activation_states.adb

```ada
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
```

(continues on next page)

(continued from previous page)

```ada
         when 0      => return Off;
         when 1      => return On;
         when others => return Unknown;
      end case;
   end Integer_To_Activation_State;

   function Real_To_Activation_State
     (S : String)
      return Activation_State
   is
      V : constant Float := Float'Value (S);
   begin
      if V < 0.0 then
         return Unknown;
      elsif V < 1.0 then
         return Off;
      else
         return On;
      end if;
   end Real_To_Activation_State;

end Activation_States;
```

Listing 122: 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: 8b34393e9b624cb5620ca77d5502ec8e
```

Runtime output

```
String: Off  => OFF
Integer: 1   => ON
Real:    1.5 => ON
```

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 123: using_string_literal.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Activation_
↪States
MD5: 975a3c56e7a938a89a617dc59c5302a7
```

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 124: misspelling_example.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Activation_
↪States
MD5: 81a8ff17e0fb8c7dce18780d8c11a6ad
```

Build output

```
misspelling_example.adb:6:10: error: "Offf" is undefined
misspelling_example.adb:6:10: error: possible misspelling of "Off"
gprbuild: *** compilation phase failed
```

As expected, the compiler detects this error. However, this error is accepted when using the corresponding string literal:

Listing 125: 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

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Activation_
↪States
MD5: 23b708a133ce1e24c0e9f7f72fa2eb29
```

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 126: silly_records.ads

```ada
package Silly_Records is

   type Silly is record
```

(continues on next page)

(continued from previous page)

```
      X : Integer;
      Y : Float;
   end record
     with String_Literal => To_Silly;

   function To_Silly (S : Wide_Wide_String)
                      return Silly;
end Silly_Records;
```

Listing 127: silly_records.adb

```
package body Silly_Records is

   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 128: 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

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Record_
↪Literals
MD5: 7395145a41de38dbbee117e27aec8c64
```

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[45]

[45] http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html

CHAPTER

TWO

TYPES AND REPRESENTATION

2.1 Enumeration Representation Clauses

We have talked about the internal code of an enumeration *in another section* (page 31). We may change this internal code by using a representation clause, which has the following format:

```
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:

Listing 1: days.ads

```
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;
```

Listing 2: show_days.adb

```
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 = "
```

(continues on next page)

(continued from previous page)

```
                & Integer'Image
                    (Day'Enum_Rep (D)));
   end loop;
end Show_Days;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Enumeration_
↪Representation_Clauses.Enumeration_Values
MD5: a70c3f8a967c355a4bf8f2d669f9c541
```

Runtime output

```
MON position      =  0
MON internal code =  1
TUE position      =  1
TUE internal code =  2
WED position      =  2
WED internal code =  4
THU position      =  3
THU internal code =  8
FRI position      =  4
FRI internal code =  16
SAT position      =  5
SAT internal code =  32
SUN position      =  6
SUN internal code =  64
```

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:

Listing 3: days.ads

```
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;
```

2.2 Data Representation

The following sections provide 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[46] course.

In the Ada Reference Manual

- 13.2 Packed Types[47]
- 13.3 Operational and Representation Attributes[48]
- 13.5.3 Bit Ordering[49]

2.3 Sizes

Ada offers multiple attributes to retrieve the size of a type or an object:

Attribute	Description
`Size`	Size of the representation of a subtype or an object (in bits).
`Object_Size`	Size of a component or an aliased object (in bits).
`Component_Size`	Size of a component of an array (in bits).
`Storage_Size`	Number of storage elements reserved for an access type or a task object.

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.

2.3.1 Size attribute and aspect

Let's start with a code example using the `Size` attribute:

Listing 4: custom_types.ads

```
package Custom_Types is

```

(continues on next page)

[46] https://learn.adacore.com/courses/intro-to-embedded-sys-prog/chapters/low_level_programming.html#intro-embedded-sys-prog-low-level-programming
[47] http://www.ada-auth.org/standards/22rm/html/RM-13-2.html
[48] http://www.ada-auth.org/standards/22rm/html/RM-13-3.html
[49] http://www.ada-auth.org/standards/22rm/html/RM-13-5-3.html

(continued from previous page)

```
   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

```
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'Size'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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↪Sizes
MD5: e0da7cd23dc6989bea3d2902221f033e
```

Build output

```
show_sizes.adb:6:04: warning: variable "V1" is read but never assigned [-gnatwv]
show_sizes.adb:7:04: warning: variable "V2" is read but never assigned [-gnatwv]
```

Runtime output

```
UInt_7'Size:            7
UInt_7'Object_Size:     8
V1'Size:                8

UInt_7_S32'Size:        32
UInt_7_S32'Object_Size: 32
V2'Size:                32
```

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

```
UInt_7'Size:            7
UInt_7'Object_Size:     8
V1'Size:                8

UInt_7_S32'Size:        32
```

(continues on next page)

(continued from previous page)

```
UInt_7_S32'Object_Size:  32
V2'Size:                 32
```

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!"

2.3.2 Component size

Let's continue our discussion on sizes with an example that makes use of the Component_Size attribute:

Listing 6: custom_types.ads

```ada
package Custom_Types is

   type UInt_7 is range 0 .. 127;

   type UInt_7_Array is
     array (Positive range <>) of UInt_7;

   type UInt_7_Array_Comp_32 is
     array (Positive range <>) of UInt_7
       with Component_Size => 32;

end Custom_Types;
```

Listing 7: show_sizes.adb

```
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'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

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↪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
UInt_7_Array'Component_Size:        8
Arr_1'Component_Size:               8
```

(continues on next page)

(continued from previous page)

```
Arr_1'Size:                        160

UInt_7_Array_Comp_32'Object_Size:   68719476704
UInt_7_Array_Comp_32'Object_Size:   68719476704
UInt_7_Array_Comp_32'Component_Size: 32
Arr_2'Component_Size:              32
Arr_2'Size:                        640
```

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

```
UInt_7_Array'Size:                 17179869176
UInt_7_Array'Object_Size:          17179869176
UInt_7_Array'Component_Size:       8
Arr_1'Component_Size:              8
Arr_1'Size:                        160

UInt_7_Array_Comp_32'Size:         68719476704
UInt_7_Array_Comp_32'Object_Size:  68719476704
UInt_7_Array_Comp_32'Component_Size: 32
Arr_2'Component_Size:              32
Arr_2'Size:                        640
```

Here, the value we get for `Component_Size` of the `UInt_7_Array` type is 8 bits, which matches the `UInt_7'Object_Size` — as we've seen in the previous subsection. In general, we expect the component size to match the object size of the underlying type.

However, we might have component sizes that aren't equal to the object size of the component's type. For example, in the declaration of the `UInt_7_Array_Comp_32` type, we're using the `Component_Size` aspect to query whether the size of each component can be 32 bits:

```
type UInt_7_Array_Comp_32 is
  array (Positive range <>) 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).

2.3.3 Storage size

To complete our discussion on sizes, let's look at this example of storage sizes:

Listing 8: custom_types.ads

```
package Custom_Types is

   type UInt_7 is range 0 .. 127;

   type UInt_7_Access is access UInt_7;

end Custom_Types;
```

Listing 9: show_sizes.adb

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

with Custom_Types; use Custom_Types;

procedure Show_Sizes is
   AV1, AV2 : UInt_7_Access;
begin
   Put_Line
     ("UInt_7_Access'Storage_Size:          "
      & UInt_7_Access'Storage_Size'Image);
   Put_Line
     ("UInt_7_Access'Storage_Size (bits):   "
      & Integer'Image (UInt_7_Access'Storage_Size
                       * System.Storage_Unit));

   Put_Line
     ("UInt_7'Size:                "
      & UInt_7'Size'Image);
   Put_Line
     ("UInt_7_Access'Size:         "
      & UInt_7_Access'Size'Image);
   Put_Line
     ("UInt_7_Access'Object_Size: "
      & UInt_7_Access'Object_Size'Image);
   Put_Line
     ("AV1'Size:                   "
      & AV1'Size'Image);
   New_Line;

   Put_Line ("Allocating AV1...");
   AV1 := new UInt_7;
   Put_Line ("Allocating AV2...");
   AV2 := new UInt_7;
   New_Line;

   Put_Line
     ("AV1.all'Size:               "
      & AV1.all'Size'Image);
   New_Line;
end Show_Sizes;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↪Sizes
MD5: 5e652ee25b8550ac331f3ce98e24f7ba
```

Runtime output

```
UInt_7_Access'Storage_Size:          0
UInt_7_Access'Storage_Size (bits):   0
UInt_7'Size:                 7
UInt_7_Access'Size:          64
UInt_7_Access'Object_Size:   64
AV1'Size:                    64

Allocating AV1...
Allocating AV2...

AV1.all'Size:                8
```

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

```
UInt_7_Access'Storage_Size:          0
UInt_7_Access'Storage_Size (bits):   0

UInt_7'Size:                 7
UInt_7_Access'Size:          64
UInt_7_Access'Object_Size:   64
AV1'Size:                    64

Allocating AV1...
Allocating AV2...

AV1.all'Size:                8
```

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:

Listing 10: custom_types.ads

```
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;
```

Listing 11: show_sizes.adb

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

(continues on next page)

(continued from previous page)

```

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

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↪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

UInt_7_Reserved_Access'Size:        64
UInt_7_Reserved_Access'Object_Size: 64
RAV1'Size:                          64
```

(continues on next page)

(continued from previous page)

```
Allocating RAV1...
Allocating RAV2...

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

In this case, we're reserving 8 storage elements in the declaration of UInt_7_Reserved_Access.

```
type UInt_7_Reserved_Access is access UInt_7
  with Storage_Size => 8;
```

Since each storage element corresponds to one byte (8 bits) in this architecture, we're reserving a maximum of 64 bits (or 8 bytes) for the UInt_7_Reserved_Access type.

This example raises an exception at runtime — a storage error, to be more specific. This is because the maximum reserved size is 64 bits, and the size of a single access object is 64 bits as well. Therefore, after the first allocation, the reserved storage space is already consumed, so we cannot allocate a second access object.

This behavior might be quite limiting in many cases. However, for certain applications where memory is very constrained, this might be exactly what we want to see. For example, having an exception being raised when the allocated memory for this data type has reached its limit might allow the application to have enough memory to at least handle the exception gracefully.

2.4 Alignment

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

Listing 12: custom_types.ads

```
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;
```

Listing 13: show_alignment.adb

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

with Custom_Types; use Custom_Types;

procedure Show_Alignment is
   V         : constant UInt_7         := 0;
   Aligned_V : constant Aligned_UInt_7 := 0;
begin
   Put_Line
     ("UInt_7'Alignment:           "
      & UInt_7'Alignment'Image);
   Put_Line
     ("UInt_7'Size:                "
      & UInt_7'Size'Image);
```

(continues on next page)

(continued from previous page)

```
   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

Aligned_UInt_7'Alignment:   4
Aligned_UInt_7'Size:        7
Aligned_UInt_7'Object_Size: 32
```

(continues on next page)

(continued from previous page)

```
Aligned_V'Alignment:         4
Aligned_V'Size:              32
```

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):

```
type Aligned_UInt_7 is new UInt_7
  with Alignment => 4;
```

When using the Alignment attribute for the Aligned_UInt_7 type, we can confirm that its alignment is indeed 4 storage units (bytes).

Note that we can use the Alignment attribute for both data types and objects — in the code above, we're using UInt_7'Alignment and V'Alignment, for example.

Because of the alignment we're specifying for the Aligned_UInt_7 type, its size — indicated by the Object_Size attribute — is 32 bits instead of 8 bits as for the UInt_7 type.

Note that you can also retrieve the alignment associated with a class using S'Class'Alignment. For example:

Listing 14: show_class_alignment.adb

```
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;
```

2.5 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

   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.");
```

(continues on next page)

(continued from previous page)

```
      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;
```

Listing 17: main.adb

```
with Int_Array_Processing;
use  Int_Array_Processing;

procedure Main is
   A : Int_Array (1 .. 20) := (others => 3);
   B : Int_Array (1 .. 20) := (others => 4);
begin
   Process (A, A);
   --  In-place processing:
   --  sharing the exact same storage

   Process (A (1 .. 10), A (10 .. 20));
   --  Overlapping one component: A (10)

   Process (A (1 .. 10), A (11 .. 20));
   --  Out-of-place processing:
   --  same array, but not sharing any storage

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

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 ↪Overlapping_Storage
MD5: 0f599163c6f24c3ef46ec6577b501c21
```

Build output

```
int_array_processing.adb:29:24: warning: "Y" may be referenced before it has a␣
 ↪value [enabled by default]
```

Runtime output

```
==== PROCESS ====
Info: X and Y have the same storage.
Info: X and Y overlap.
In-place processing...

==== PROCESS ====
Info: X and Y don't havethe same storage.
Info: X and Y overlap.
Cannot process overlapping arrays...

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

==== PROCESS ====
Info: X and Y don't havethe 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.

2.6 Packed Representation

As we've seen previously, the minimum number of bits required to represent a data type might be less than the actual number of bits used to store an object of that same type. We've seen an example where `UInt_7'Size` was 7 bits, while `UInt_7'Object_Size` was 8 bits. The most extreme case is the one for the `Boolean` type: in this case, `Boolean'Size` is 1 bit, while `Boolean'Object_Size` might be 8 bits (or even more on certain architectures). In such cases, we have 7 (or more) unused bits in memory for each object of `Boolean` type. In other words, we're wasting memory. On the other hand, we're gaining speed of access because we can directly access each element without having to first change its internal representation back and forth. We'll come back to this point later.

The situation is even worse when implementing bit-fields, which can be declared as an array of **Boolean** components. For example:

Listing 18: flag_definitions.ads

```
package Flag_Definitions is

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

end Flag_Definitions;
```

Listing 19: show_flags.adb

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

procedure Show_Flags is
   Flags_1 : Flags (1 .. 8);
begin
   Put_Line ("Boolean'Size:           "
             & Boolean'Size'Image);
   Put_Line ("Boolean'Object_Size:    "
             & Boolean'Object_Size'Image);
   Put_Line ("Flags_1'Size:           "
             & Flags_1'Size'Image);
   Put_Line ("Flags_1'Component_Size: "
             & Flags_1'Component_Size'Image);
end Show_Flags;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↪Non_Packed_Flags
MD5: 6fd7a913e3c6717e846c2e822c1cbad7
```

Build output

```
show_flags.adb:5:04: warning: variable "Flags_1" is read but never assigned [-
↪gnatwv]
```

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

```
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

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

procedure Show_Packed_Flags is
   Flags_1 : Flags (1 .. 8);
   Flags_2 : Packed_Flags (1 .. 8);
begin
   Put_Line ("Boolean'Size:           "
             & Boolean'Size'Image);
   Put_Line ("Boolean'Object_Size:    "
             & Boolean'Object_Size'Image);
   Put_Line ("Flags_1'Size:           "
             & Flags_1'Size'Image);
   Put_Line ("Flags_1'Component_Size: "
             & Flags_1'Component_Size'Image);
   Put_Line ("Flags_2'Size:           "
             & Flags_2'Size'Image);
   Put_Line ("Flags_2'Component_Size: "
             & Flags_2'Component_Size'Image);
end Show_Packed_Flags;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↪Packed_Flags
MD5: c71cf68dc8bc41d0df2a5e3eb61b51fd
```

Build output

```
show_packed_flags.adb:5:04: warning: variable "Flags_1" is read but never assigned␣
↪[-gnatwv]
show_packed_flags.adb:6:04: warning: variable "Flags_2" is read but never assigned␣
↪[-gnatwv]
```

Runtime output

```
Boolean'Size:            1
Boolean'Object_Size:     8
Flags_1'Size:            64
Flags_1'Component_Size:  8
Flags_2'Size:            8
Flags_2'Component_Size:  1
```

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

```
Boolean'Size:           1
Boolean'Object_Size:    8
Flags_1'Size:           64
Flags_1'Component_Size: 8
Flags_2'Size:           8
Flags_2'Component_Size: 1
```

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 performs the type conversion.

Let's modify the previous example and introduce this type conversion:

Listing 22: flag_definitions.ads

```
package Flag_Definitions is

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

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

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

end Flag_Definitions;
```

Listing 23: show_flag_conversion.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
```

(continues on next page)

(continued from previous page)

```
↪Flag_Conversion
MD5: faff2079f6779097b6e0f7cd6cd48612
```

Runtime output

```
1: TRUE, TRUE
2: TRUE, TRUE
3: FALSE, FALSE
4: TRUE, TRUE
5: FALSE, FALSE
6: FALSE, FALSE
7: TRUE, TRUE
8: TRUE, TRUE
```

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:

```
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 111).

2.6.1 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:

Representation	More speed of access	Less memory consumption
Unpacked	X	
Packed		X

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.

2.7 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[50] 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

```
package P is

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

end P;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Rep_Clauses_1
MD5: 88171257118810bb7e02cea60ffb1ad9
```

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:

position	0	1	2	3	4	5	6	7
component	A				B			

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:

position	0	1	2	3	4	5	6	7
bits	#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31	#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31
component	A				B			

The memory representation we see in the graph above can be described in Ada using representation clauses, as you can see in the code starting at the `for R use record` line in the code example below — we'll discuss the syntax and further details right after this example.

50 https://learn.adacore.com/courses/intro-to-embedded-sys-prog/chapters/low_level_programming.html#intro-embedded-sys-prog-low-level-programming

Listing 25: p.ads

```
package P is

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

   --  Representation clause for record R:
   for R use record
      A at 0 range 0 .. 31;
      --   ^ starting memory position
      B at 4 range 0 .. 31;
      --           ^ first bit .. last bit
   end record;

end P;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Rep_Clauses_2
MD5: b6be86ae7e1a5c2e7d981fe37bad49ed
```

Here, we're specifying that the A component is stored in the bits #0 up to #31 starting at position #0. Note that the position itself doesn't represent an absolute address in the device's memory; instead, it's relative to the memory space reserved for that record. The B component has the same 32-bit range, but starts at position #4.

This is a generalized view of the syntax:

```
for Record_Type use record
   Component_Name at Start_Position
                  range First_Bit .. Last_Bit;
end record;
```

These are the elements we see above:

- `Component_Name`: name of the component (from the record type declaration);
- `Start_Position`: start position — in storage units — of the memory space reserved for that component;
- `First_Bit`: first bit (in the start position) of the component;
- `Last_Bit`: last bit of the component.

Note that the last bit of a component might be in a different storage unit. Since the `Integer` type has a larger width (32 bits) than the storage unit (8 bits), components of that type span over multiple storage units. Therefore, in our example, the first bit of component A is at position #0, while the last bit is at position #3.

Also note that the last eight bits of component A are bits #24 .. #31. If we think in terms of storage units, this corresponds to bits #0 .. #7 of position #3. However, when specifying the last bit in Ada, we always use the `First_Bit` value as a reference, not the position where those bits might end up. Therefore, we write `range 0 .. 31`, well knowing that those 32 bits span over four storage units (positions #0 .. #3).

In the Ada Reference Manual

- 13.5.1 Record Representation Clauses[51]

2.7.1 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:

Listing 26: p.ads

```
package P is

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

end P;
```

Listing 27: show_storage.adb

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

with P;           use P;

procedure Show_Storage is
   R1 : R;
begin
   Put_Line ("R'Size:              "
             & R'Size'Image);
   Put_Line ("R'Object_Size:       "
             & R'Object_Size'Image);
   New_Line;

   Put_Line ("System.Storage_Unit: "
             & System.Storage_Unit'Image);
   New_Line;

   Put_Line ("R1.A'Position  : "
             & R1.A'Position'Image);
   Put_Line ("R1.A'First_Bit : "
             & R1.A'First_Bit'Image);
   Put_Line ("R1.A'Last_Bit  : "
             & R1.A'Last_Bit'Image);
   New_Line;

   Put_Line ("R1.B'Position  : "
             & R1.B'Position'Image);
   Put_Line ("R1.B'First_Bit : "
             & R1.B'First_Bit'Image);
   Put_Line ("R1.B'Last_Bit  : "
             & R1.B'Last_Bit'Image);
end Show_Storage;
```

[51] http://www.ada-auth.org/standards/22rm/html/RM-13-5-1.html

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Storage_Place_Attributes
MD5: 05a402585ce71eb47cf972e68c02835e
```

Build output

```
show_storage.adb:7:04: warning: variable "R1" is read but never assigned [-gnatwv]
```

Runtime output

```
R'Size:               64
R'Object_Size:        64

System.Storage_Unit:  8

R1.A'Position   :  0
R1.A'First_Bit :  0
R1.A'Last_Bit   :  31

R1.B'Position   :  4
R1.B'First_Bit :  0
R1.B'Last_Bit   :  31
```

First of all, we see that the size of the R type is 64 bits, which can be explained by those two 32-bit integer components. Then, we see that components A and B start at positions #0 and #4, and each one makes use of bits in the range from #0 to #31. This matches the graph we've seen above.

In the Ada Reference Manual

- 13.5.2 Storage Place Attributes[52]

2.7.2 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:

position	0	1	2	3	4	5	6	7	8
bits	#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31		#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31
component	A					B			

This is the code that implements that:

Listing 28: p.ads

```
package P is

   type R is record
      A : Integer;
```

(continues on next page)

52 http://www.ada-auth.org/standards/22rm/html/RM-13-5-2.html

(continued from previous page)

```ada
      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

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Rep_Clauses_Empty_Byte
MD5: c616e534e95a06f2e8b3052a3e8a9aab
```

Runtime output

```
R'Size:         72
R'Object_Size:  96
```

When running the application above, we see that, due to the extra byte in the record representation, the sizes increase. On a typical 64-bit PC, R'Size is now 76 bits, which reflects the additional eight bits that we introduced between components A and B. Depending on the target architecture, you may also see that R'Object_Size is now 96 bits, which is the size the compiler selects as the most appropriate for this record type. As we've mentioned in the previous section, we can use aspects to request a specific size to the compiler. In this case, we could use the Object_Size aspect:

Listing 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

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

with P;           use P;

procedure Show_Empty_Byte is
begin
   Put_Line ("R'Size:        "
             & R'Size'Image);
   Put_Line ("R'Object_Size: "
             & R'Object_Size'Image);
end Show_Empty_Byte;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Rep_Clauses_Empty_Byte
MD5: 9d7bae2b2aabeda4bc03752544cee9b9
```

Runtime output

```
R'Size:         72
R'Object_Size:  72
```

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

2.7.3 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:

Listing 32: p.ads

```
package P is

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

end P;
```

Listing 33: p-rep.ads

```
package P.Rep is

   type R_New is new R
     with Object_Size => 72;

   for R_New use record
```

(continues on next page)

(continued from previous page)

```
      A at 0 range 0 .. 31;
      B at 5 range 0 .. 31;
   end record;

end P.Rep;
```

Listing 34: show_empty_byte.adb

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

with P;           use P;
with P.Rep;       use P.Rep;

procedure Show_Empty_Byte is
begin
   Put_Line ("R'Size:        "
             & R'Size'Image);
   Put_Line ("R'Object_Size: "
             & R'Object_Size'Image);

   Put_Line ("R_New'Size:        "
             & R_New'Size'Image);
   Put_Line ("R_New'Object_Size: "
             & R_New'Object_Size'Image);
end Show_Empty_Byte;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Derived_Rep_Clauses_Empty_Byte
MD5: 3a1e0837f8bd8250f20fc7b274b869d5
```

Runtime output

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

When running this example, we see that the R type retains the memory representation selected by the compiler for the target architecture, while the R_New has the memory representation that we specified.

2.7.4 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:

bit	0	1	2	3	4	5	6	7
component	S		(reserved)		Error	V1		

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:

Listing 35: p.ads

```
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;
```

Listing 36: show_simple_reg.adb

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

with P;           use P;

procedure Show_Simple_Reg is
begin
   Put_Line ("Simple_Reg'Size:        "
             & Simple_Reg'Size'Image);
   Put_Line ("Simple_Reg'Object_Size: "
             & Simple_Reg'Object_Size'Image);
end Show_Simple_Reg;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Rep_Clauses_Simple_Reg
MD5: cbac444336572460062f922767c226a5
```

Runtime output

```
Simple_Reg'Size:        8
Simple_Reg'Object_Size:  8
```

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:

Listing 37: show_simple_reg.adb

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

with P;           use P;
```

(continues on next page)

(continued from previous page)

```

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
↪Storage_Clauses.Rep_Clauses_Simple_Reg
MD5: e442396e43d6609c1c837165bbc21641
```

Runtime output

```
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 `R.S`. To update the *V1* field of the register with the value 4, we just have to write `R.V1 := 4`. No extra code — such as bit-masking or bit-shifting — is needed here.

In other languages

Some programming languages require that developers use complicated, error-prone approaches — which may include manually bit-shifting and bit-masking variables — to retrieve information from or store information to individual bits or registers. In Ada, however, this is efficiently handled by the compiler, so that developers only need to correctly describe the register mapping using representation clauses.

2.8 Changing Data Representation

Note

This section was originally written by Robert Dewar and published as Gem #27: Changing Data Representation[53] and Gem #28[54].

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:

[53] https://www.adacore.com/gems/gem-27
[54] https://www.adacore.com/gems/gem-28

Listing 38: communication.ads

```
package Communication is

   type Com_Packet is record
      Key : Boolean;
      Id  : Character;
      Val : Integer range 100 .. 227;
   end record;

   for Com_Packet use record
      Key at 0 range 0 .. 0;
      Id  at 0 range 1 .. 8;
      Val at 0 range 9 .. 15;
   end record;

end Communication;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
 ↪Representation.Com_Packet
MD5: cbd7f5547c5b0458853ac21d03aa41f8
```

Build output

```
communication.ads:12:11: warning: component clause forces biased representation␣
 ↪for "Val" [-gnatw.b]
```

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

Listing 39: array_representation.ads

```
package Array_Representation is

   type Val is (A, B, C, D, E, F, G, H);

   type Arr is array (1 .. 16) of Val
     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
```

which forces the components to take only 3 bits, crossing byte boundaries as needed. A final example is:

Listing 40: enumeration_representation.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
↪Representation.Enum_Rep
MD5: 3c3e9f4ae11e9bb2482588d27ba43c30
```

which allows specified values for an enumeration type, instead of the efficient default values of 0, 1, 2.

In all these cases, we might use these representation clauses to match external specifications, which can be very useful. The disadvantage of such layouts is that they are inefficient, and accessing individual components, or, in the case of the enumeration type, looping through the values can increase space and time requirements for the program code.

One approach that is often effective is to read or write the data in question in this specified form, but internally in the program represent the data in the normal default layout, allowing efficient access, and do all internal computations with this more efficient form.

To follow this approach, you will need to convert between the efficient format and the specified format. Ada provides a very convenient method for doing this, as described in RM 13.6 "Change of Representation"[55].

The idea is to use type derivation, where one type has the specified format and the other has the normal default format. For instance for the array case above, we would write:

Listing 41: array_representation.ads

```
package Array_Representation is

   type Val is (A, B, C, D, E, F, G, H);
   type Arr is array (1 .. 16) of Val;

   type External_Arr is new Arr
     with Component_Size => 3;

end Array_Representation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
↪Representation.Array_Rep
MD5: d4e90f6ef8ff81771980771356eab235
```

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);

```

(continues on next page)

55 http://www.ada-auth.org/standards/22rm/html/RM-13-6.html

(continued from previous page)

```
   --  (computations using efficient
   --   Work_Data form)

   --  Convert back to external form
   Output_Data := External_Arr (Work_Data);

end Using_Array_For_IO;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
↪Representation.Array_Rep
MD5: 88efe4b8a7f07e0c32f11131d6eafbc1
```

Build output

```
using_array_for_io.adb:5:04: warning: variable "Input_Data" is read but never␣
↪assigned [-gnatwv]
```

Using this approach, the quite complex task of copying all the data of the array from one form to another, with all the necessary masking and shift operations, is completely automatic.

Similar code can be used in the record and enumeration type cases. It is even possible to specify two different representations for the two types, and convert from one form to the other, as in:

Listing 43: enumeration_representation.ads

```
package Enumeration_Representation is

   type Status_In is (Off, On, Unknown);
   type Status_Out is new Status_In;

   for Status_In use (Off     => 2#001#,
                      On      => 2#010#,
                      Unknown => 2#100#);
   for Status_Out use (Off     => 103,
                       On      => 1045,
                       Unknown => 7700);

end Enumeration_Representation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
↪Representation.Enum_Rep
MD5: f78c3718280f9265ff54270c5834b458
```

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[56] (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:

[56] http://www.ada-auth.org/standards/22rm/html/RM-13-1.html

Listing 44: array_representation.ads

```
package Array_Representation is

   type Val is (A, B, C, D, E, F, G, H);
   type Arr is array (1 .. 16) of Val;

   procedure Rearrange (Arg : in out Arr);

   type External_Arr is new Arr
     with Component_Size => 3;

end Array_Representation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
 ↪Representation.Array_Rep_2
MD5: 70201932d40e3fb356bc1d8ab188f2df
```

Build output

```
array_representation.ads:9:11: error: representation item not permitted before Ada␣
 ↪2022
array_representation.ads:9:11: error: parent type "Arr" has primitive operations
gprbuild: *** compilation phase failed
```

Why these restrictions? Well, the answer is a little complex, and has to do with efficiency considerations, which we will address below.

2.8.1 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:

```
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 13.1[57] (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

```
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

```
package body My_Ints is

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

end My_Ints;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
↪Representation.My_Int
MD5: a2940169830799828 8f02b349d04d1d2
```

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

[57] http://www.ada-auth.org/standards/22rm/html/RM-13-1.html

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
↪Representation.My_Int
MD5: f68272d55e68687b7102885313c7831b
```

Build output

```
using_my_int.adb:4:04: warning: variable "Var" is read but never assigned [-gnatwv]
```

The compiler translates this as:

Listing 48: using_my_int.adb

```ada
with My_Ints; use My_Ints;

procedure Using_My_Int is
   Var : My_Int_2;
begin

   if Odd (My_Int_1 (Var)) then
      --   ^ Converting My_Int_2 to
      --     My_Int_1 type before
      --     calling Odd function.
      null;
   end if;

end Using_My_Int;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_
↪Representation.My_Int
MD5: b3d0053c61412a2b985cd580b645e048
```

Build output

```
using_my_int.adb:4:04: warning: variable "Var" is read but never assigned [-gnatwv]
```

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.

2.9 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);
```

(continues on next page)

(continued from previous page)

```
   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

```
with Ada.Sequential_IO;

package States is

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

   for State use (Off     => 1,
                  On      => 2,
                  Waiting => 4);

   package State_Sequential_IO is new
     Ada.Sequential_IO (State);

   procedure Read_Display_States
     (File_Name : String);

end States;
```

Listing 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_Sequential_IO;

      F : State_Sequential_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);
```

(continues on next page)

(continued from previous page)

```
         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);
   Read_Display_States (File_Name);
end Show_States_From_File;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Valid_Attribute.Valid_
↪States
MD5: f7af2946ebe663932494448a0d3d3020
```

Runtime output

```
OFF
ON
WAITING
Invalid value detected!
ON
Invalid value detected!
```

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:

```
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[58]

2.10 Unchecked Union

We've introduced variant records back in the Introduction to Ada course[59]. 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**. (We talk again about *variant records* (page 231) in another chapter.)

The `State_Or_Integer` declaration in the `States` package below is an example of a variant record:

Listing 54: states.ads

```
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;
```

Listing 55: states.adb

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

package body States is

   procedure Display_State_Value
     (V : State_Or_Integer)
   is
   begin
```

(continues on next page)

58 http://www.ada-auth.org/standards/22rm/html/RM-13-9-2.html
59 https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_records.html#intro-ada-variant-records

(continued from previous page)

```
      Put_Line ("State: " & V.S'Image);
      Put_Line ("Value: " & V.I'Image);
   end Display_State_Value;

end States;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_
↪Or_Integer
MD5: fa72f52a4396a2e66931ff6932c567fc
```

As mentioned in the previous course, if you try to access a component that is not valid for your record, a `Constraint_Error` exception is raised. For example, in the implementation of the `Display_State_Value` procedure, we're trying to retrieve the value of the integer component (`I`) of the `V` record. When calling this procedure, the `Constraint_Error` exception is raised as expected because `Use_Enum` is set to **True**, so that the `I` component is invalid — only the `S` component is valid in this case.

Listing 56: show_variant_rec_error.adb

```
with States; use States;

procedure Show_Variant_Rec_Error is
   V : State_Or_Integer (Use_Enum => True);
begin
   V.S := On;
   Display_State_Value (V);
end Show_Variant_Rec_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_
↪Or_Integer
MD5: b8cf215dd55bfdec6950df35c7bc19b9
```

Runtime output

```
State: ON

raised CONSTRAINT_ERROR : states.adb:10 discriminant check failed
```

In addition to not being able to read the value of a component that isn't valid, assigning a value to a component that isn't valid also raises an exception at runtime. In this example, we cannot assign to `V.I`:

Listing 57: show_variant_rec_error.adb

```
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: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
```

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);
```

(continues on next page)

(continued from previous page)

```
   end Display_State_Value;

end States;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.
↪Unchecked_State_Or_Integer
MD5: e97271a24aab23d2db450308401667ac
```

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

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.
↪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

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.Unchecked_State_Or_Integer
MD5: bb472e91c5e7b7e63d6246dbcf5226a0
```

Runtime output

```
State: ON
Value:  2
State: WAITING
Value:  4
```

In the example above, we're use an aggregate in the assignments to V. By doing so, we avoid that `Use_Enum` is set to the *wrong* component. For example:

Listing 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

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.Unchecked_State_Or_Integer
MD5: 74ac11a3effdafd3959fface295a86da
```

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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.
↪Unchecked_State_Or_Integer
MD5: f63e64df137cfc3c29e41f784306f0e4
```

Runtime output

```
raised CONSTRAINT_ERROR : states.adb:9 invalid data
```

To mitigate this problem, we could use the `Valid` attribute — discussed in the previous section — for the `S` component before trying to use its value in the implementation of the `Display_State_Value` procedure:

Listing 64: states.adb

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

package body States is

   procedure Display_State_Value
     (V : Unchecked_State_Or_Integer)
   is
   begin
      if V.S'Valid then
         Put_Line ("State: " & V.S'Image);
      else
         Put_Line ("State: <invalid>");
      end if;
      Put_Line ("Value: " & V.I'Image);
   end Display_State_Value;

end States;
```

Listing 65: show_unchecked_union.adb

```
with States; use States;

procedure Show_Unchecked_Union is
   V : Unchecked_State_Or_Integer
         (Use_Enum => True);
begin
   V := (Use_Enum => False, I => 3);
   Display_State_Value (V);
end Show_Unchecked_Union;
```

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[60]

2.11 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 593) in another chapter. Also, later on in that chapter, we discuss the *relation between access types and addresses* (page 706).) 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 130) and *address arithmetic* (page 132) (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[61]

2.11.1 Address attribute

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

Listing 66: use_address.adb

```
with System; use System;

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

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Address_
↪Attribute
MD5: 1ee71b7cd3ed278647eb72f383da877f
```

Here, we're assigning the address of the `I` object to the `A` address.

[60] http://www.ada-auth.org/standards/22rm/html/RM-B-3-3.html
[61] http://www.ada-auth.org/standards/22rm/html/RM-13-7.html

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:

```
function System.Address_Image
  (A : System.Address) return String;
```

We can use this function to display the address in an user message, for example:

Listing 67: show_address_attribute.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Show_
↪Address_Attribute
MD5: 72efddedc57701665594de5ee1939d3d
```

Runtime output

```
Address : 00007FFCC0BEA884
```

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes[62]
- 13.7 The Package System[63]

2.11.2 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:

Listing 68: show_address.adb

```
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;
```

(continues on next page)

[62] http://www.ada-auth.org/standards/22rm/html/RM-13-3.html
[63] http://www.ada-auth.org/standards/22rm/html/RM-13-7.html

(continued from previous page)

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Address_
↪Aspect
MD5: 6339c743b1ca2b1adf58c977540b43d5
```

Runtime output

```
I_Main'Address   : 00007FFF413746A4
I_Mapped'Address : 00007FFF413746A4
```

This approach allows us to create an overlay. For example:

Listing 69: simple_overlay.adb

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

procedure Simple_Overlay is
   type State is (Off, State_1, State_2)
     with Size => Integer'Size;

   for State use (Off     => 0,
                  State_1 => 32,
                  State_2 => 64);

   S : State;
   I : Integer
     with Address => S'Address, Import, Volatile;
begin
   S := State_2;
   Put_Line ("I = " & Integer'Image (I));
end Simple_Overlay;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Simple_
↪Overlay
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:

Listing 70: 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[64], 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 131) function later on.)

Note that we're using the Atomic aspect here, which we discuss *in another chapter* (page 146).

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes[65]
- 13.7 The Package System[66]
- 13.7.1 The Package System.Storage_Elements[67]

2.11.3 Address comparison

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

Listing 71: show_address.adb

```
with System; use System;
with System.Address_Image;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Address is

   I, J : Integer;
begin
   Put_Line ("I'Address   : "
             & System.Address_Image
                 (I'Address));
```

(continues on next page)

64 https://gcc.gnu.org/onlinedocs/gnat_rm/Attribute-To_005fAddress.html
65 http://www.ada-auth.org/standards/22rm/html/RM-13-3.html
66 http://www.ada-auth.org/standards/22rm/html/RM-13-7.html
67 http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html

(continued from previous page)

```
   Put_Line ("J'Address   : "
             & System.Address_Image
                 (J'Address));

   if I'Address = J'Address then
      Put_Line ("I'Address = J'Address");
   elsif I'Address < J'Address then
      Put_Line ("I'Address < J'Address");
   else
      Put_Line ("I'Address > J'Address");
   end if;
end Show_Address;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Address_
↪Aspect
MD5: 24ddb7d05159f26ef3b2ff6bcc2691e8
```

Runtime output

```
I'Address   : 00007FFDC2764DEC
J'Address   : 00007FFDC2764DE8
I'Address > J'Address
```

In this example, we compare the address of the I object with the address of the J object using the =, < and > operators.

In the Ada Reference Manual

- 13.7 The Package System[68]

2.11.4 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 72: show_address.adb

```
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;
```

(continues on next page)

[68] http://www.ada-auth.org/standards/22rm/html/RM-13-7.html

(continued from previous page)

```
begin
   A1 := I'Address;
   IA := To_Integer (A1);
   A2 := To_Address (IA);

   Put_Line ("A1 : "
             & System.Address_Image (A1));
   Put_Line ("IA : "
             & Integer_Address'Image (IA));
   Put_Line ("A2 : "
             & System.Address_Image (A2));
end Show_Address;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_
 ↪Arith_Ada
MD5: 69e053886fb8e8571d6c94247dc9f30f
```

Runtime output

```
A1 : 00007FFD482C4224
IA :  140725814313508
A2 : 00007FFD482C4224
```

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[69]

2.11.5 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:

Listing 73: show_address.adb

```
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

```

(continues on next page)

[69] http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html

(continued from previous page)

```
   Arr : array (1 .. 10) of Integer;
   A   : Address := Arr'Address;
   --                ^^^^^^^^^^^
   --   Initializing address object with
   --   address of the first component of Arr.
   --
   --   We could write this as well:
   --   ___ := Arr (1)'Address

begin
   for I in Arr'Range loop
      declare
         Curr : Integer
                  with Address => A;
      begin
         Curr := I;
         Put_Line ("Curr'Address : "
                   & System.Address_Image
                       (Curr'Address));
      end;

      --
      --  Address arithmetic
      --
      A := A + Storage_Offset (Integer'Size)
                / Storage_Unit;
      --     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
      --       Moving to next component
   end loop;

   for I in Arr'Range loop
     Put_Line ("Arr ("
               & Integer'Image (I)
               & ") :"
               & Integer'Image (Arr (I)));
   end loop;
end Show_Address;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_
↪Arith_Ada
MD5: 2c1cdd6874036fb9a527baae63a312d9
```

Runtime output

```
Curr'Address : 00007FFF49BF84D0
Curr'Address : 00007FFF49BF84D4
Curr'Address : 00007FFF49BF84D8
Curr'Address : 00007FFF49BF84DC
Curr'Address : 00007FFF49BF84E0
Curr'Address : 00007FFF49BF84E4
Curr'Address : 00007FFF49BF84E8
Curr'Address : 00007FFF49BF84EC
Curr'Address : 00007FFF49BF84F0
Curr'Address : 00007FFF49BF84F4
Arr ( 1) : 1
Arr ( 2) : 2
Arr ( 3) : 3
Arr ( 4) : 4
Arr ( 5) : 5
```

(continues on next page)

(continued from previous page)

```
Arr ( 6) : 6
Arr ( 7) : 7
Arr ( 8) : 8
Arr ( 9) : 9
Arr ( 10) : 10
```

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 89).)

In other languages

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

Listing 74: main.c

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int i;
    int arr[10];

    int *a = arr;
    /* int *a = &arr[0]; */

    for (i = 0; i < 10; i++)
    {
        *a++ = i;
        printf("curr address: %p\n", a);
    }

    for (i = 0; i < 10; i++)
    {
        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: 0x7ffe55bdf634
curr address: 0x7ffe55bdf638
curr address: 0x7ffe55bdf63c
curr address: 0x7ffe55bdf640
curr address: 0x7ffe55bdf644
curr address: 0x7ffe55bdf648
curr address: 0x7ffe55bdf64c
curr address: 0x7ffe55bdf650
curr address: 0x7ffe55bdf654
curr address: 0x7ffe55bdf658
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[70]
- 13.7.1 The Package System.Storage_Elements[71]

2.12 Discarding names

As we know, we can use the Image attribute of a type to get a string associated with this type. This is useful for example when we want to display a user message for an enumeration type:

Listing 75: show_enumeration_image.adb

```ada
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;
```

Code block metadata

[70] http://www.ada-auth.org/standards/22rm/html/RM-13-3.html
[71] http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Discarding_Names.
↪Enumeration_Image
MD5: 3863c5e06641d96b59edb9e76daa7560
```

Runtime output

```
Month: JANUARY
```

This is similar to having this code:

Listing 76: show_enumeration_image.adb

```
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";
         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 Show_Enumeration_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Discarding_Names.
↪Enumeration_Image
MD5: 2db86044d2045bd9d4c3998cca36d51c
```

Runtime output

```
Month: JANUARY
```

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 77: show_discard_names.adb

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

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

   M : constant Months := January;
begin
   null;
end Show_Discard_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Discarding_Names.
↪Discard_Names
MD5: 7891caac459a4be2096d443ca3190036
```

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

If we add this statement to the Show_Discard_Names procedure above:

```
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[72]

[72] http://www.ada-auth.org/standards/22rm/html/RM-C-5.html

CHAPTER

THREE

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[73]

3.1 Volatile

A volatile[74] 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[75], 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 1: show_volatile_object.adb

```
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

[73] http://www.ada-auth.org/standards/22rm/html/RM-C-6.html
[74] https://en.wikipedia.org/wiki/Volatile_(computer_programming)
[75] https://en.wikipedia.org/wiki/Memory-mapped_I/O

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Volatile.Object_
↪Ada
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:

Listing 2: 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 3: 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.Shared_Variable_Control.Volatile.Type
MD5: 0d31156d47b2edcfb94debd016c8bb87
```

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 4: 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
```

(continues on next page)

(continued from previous page)

```
   Arr := (others => 0.0);

   for I in 0 .. 999 loop
      Arr (1) := Arr (1) +  2.0 * Long_Float (I);
      Arr (2) := Arr (2) + 10.0 * Long_Float (I);
   end loop;

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

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.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 5: 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.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.

3.2 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 6: shared_var_types.ads

```
package Shared_Var_Types is

   I : Integer with Independent;

end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Object
MD5: d90fef37584ca8802b8a3e3858c0095b
```

Similarly, we can use this aspect when declaring types:

Listing 7: shared_var_types.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type
MD5: 7bcbee5b73067149b14c4b1b061f803c
```

In this example, we're declaring the Independent_Boolean type and using it in the declaration of the Flag record type. Let's now derive the Flags type and use a representation clause for the derived type:

Listing 8: shared_var_types-representation.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type
MD5: bb9d5badf33401660e7e20a7cd612dab
```

Build output

```
shared_var_types-representation.ads:6:26: error: size for independent "F1" must be
↪multiple of Storage_Unit
shared_var_types-representation.ads:7:21: error: position for independent "F2"
↪must be multiple of Storage_Unit
shared_var_types-representation.ads:7:26: error: size for independent "F2" must be
↪multiple of Storage_Unit
gprbuild: *** compilation phase failed
```

As you can see when trying to compile this example, the representation clause that we used for Rep_Flags isn't following these limitations:

1. The size of each independent component must be a multiple of a storage unit.
2. The start position of each independent component must be a multiple of a storage unit.

For example, for architectures that have a storage unit of one byte — such as standard desktop computers —, this means that the size and the position of independent components must be a multiple of a byte. Let's correct the issues in the code above by:

- setting the size of each independent component to correspond to `Storage_Unit` — using a range between 0 and `Storage_Unit - 1` —, and
- setting the start position to zero.

This is the corrected version:

Listing 9: shared_var_types-representation.ads

```ada
with System;

package Shared_Var_Types.Representation is

   type Rep_Flags is new Flags;

   for Rep_Flags use record
      F1 at 0 range 0 .. System.Storage_Unit - 1;
      F2 at 1 range 0 .. System.Storage_Unit - 1;
   end record;

end Shared_Var_Types.Representation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type
MD5: ed57e57cd746698909a4f7ce40a29dfc
```

Note that the representation that we're now using for Rep_Flags is most likely the representation that the compiler would have chosen for this data type. We could, however, have added an empty storage unit between F1 and F2 — by simply writing `F2 at 2 ...`:

Listing 10: shared_var_types-representation.ads

```ada
with System;

package Shared_Var_Types.Representation is

   type Rep_Flags is new Flags;
```

(continues on next page)

(continued from previous page)

```

   for Rep_Flags use record
      F1 at 0 range 0 .. System.Storage_Unit - 1;
      F2 at 2 range 0 .. System.Storage_Unit - 1;
   end record;

end Shared_Var_Types.Representation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type
MD5: 71fedf8aac7c19bca1ba3b487efa9b17
```

As long as we follow the rules for independent objects, we're still allowed to use representation clauses that don't correspond to the one that the compiler might select.

For arrays, we can use the Independent_Components aspect:

Listing 11: shared_var_types.ads

```
package Shared_Var_Types is

   Flags : array (1 .. 8) of Boolean
             with Independent_Components;

end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.
 ↪Components
MD5: b331d0a13adf45624b664839fe4ba42c
```

We've just seen in a previous example that some representation clauses might not work with objects and types that have the Independent aspect. The same restrictions apply when we use the Independent_Components aspect. For example, this aspect prevents that array components are packed when the Pack aspect is used. Let's discuss the following erroneous code example:

Listing 12: shared_var_types.ads

```
package Shared_Var_Types is

   type Flags is
     array (Positive range <>) of Boolean
       with Independent_Components, Pack;

   F : Flags (1 .. 8) with Size => 8;

end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.
 ↪Packed_Independent_Components
MD5: dbaff4f2559ef8a449dad251f42cddc0
```

Build output

```
shared_var_types.ads:5:37: warning: cannot pack independent components (RM 13.2(7))
shared_var_types.ads:7:36: error: size for "F" too small, minimum allowed is 64
gprbuild: *** compilation phase failed
```

As expected, this code doesn't compile. Here, we can have either independent components, or packed components. We cannot have both at the same time because packed components aren't independently addressable. The compiler warns us that the Pack aspect won't have any effect on independent components. When we use the Size aspect in the declaration of F, we confirm this limitation. If we remove the Size aspect, however, the code is compiled successfully because the compiler ignores the Pack aspect and allocates a larger size for F:

Listing 13: shared_var_types.ads

```ada
package Shared_Var_Types is

   type Flags is
     array (Positive range <>) of Boolean
       with Independent_Components, Pack;

end Shared_Var_Types;
```

Listing 14: show_flags_size.adb

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

with Shared_Var_Types; use Shared_Var_Types;

procedure Show_Flags_Size is
   F : Flags (1 .. 8);
begin
   Put_Line ("Flags'Size:      "
             & F'Size'Image & " bits");
   Put_Line ("Flags (1)'Size:  "
             & F (1)'Size'Image & " bits");
   Put_Line ("# storage units: "
             & Integer'Image
                 (F'Size /
                  System.Storage_Unit));
end Show_Flags_Size;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.
 ↪Packed_Independent_Components
MD5: b96f921b08b1d8207749517f833fc121
```

Build output

```
show_flags_size.adb:7:04: warning: variable "F" is read but never assigned [-
 ↪gnatwv]
shared_var_types.ads:5:37: warning: cannot pack independent components (RM 13.2(7))
```

Runtime output

```
Flags'Size:       64 bits
Flags (1)'Size:   8 bits
# storage units:  8
```

As you can see in the output of the application, even though we specify the Pack aspect for the Flags type, the compiler allocates eight storage units, one per each component of the

F array.

3.3 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 15: 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.Shared_Variable_Control.Atomic.Object
MD5: 5c2d8e0a9615084c2a15f896c61adaa6
```

Note that the **Address** aspect allows for assigning a variable to a specific location in the memory. In this example, we're using this aspect to specify the address of the memory-mapped register.

Later on, we talk again about the *Address aspect* (page 128) and the GNAT-specific *System'To_Address attribute* (page 129).

In addition to atomic objects, we can declare atomic types — similar to what we've seen before for volatile types. For example:

Listing 16: shared_var_types.ads

```
with System;

package Shared_Var_Types is

   type Atomic_Integer is new Integer
     with Atomic;

private
   R : Atomic_Integer
         with Address =>
                System'To_Address (16#FFFF00A0#);

end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic.Types
MD5: 009632ba0155d70def8281ba590f3d12
```

In this example, we're declaring the `Atomic_Integer` type, which is an atomic type. Objects of this type — such as R in this example — are automatically atomic.

We can also declare atomic array components:

Listing 17: shared_var_types.ads

```
package Shared_Var_Types is

private
   Arr : array (1 .. 2) of Integer
           with Atomic_Components;

end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic.Array_
 ↪Components
MD5: 7501bdf618621a822d451da8d731ef75
```

This example shows the declaration of the `Arr` array, which has atomic components — the atomicity of its components is indicated by the `Atomic_Components` aspect.

Note that if an object is atomic, it is also volatile and independent. In other words, these type declarations are equivalent:

Listing 18: shared_var_types.ads

```
package Shared_Var_Types is

   type Atomic_Integer_1 is new Integer
     with Atomic;

   type Atomic_Integer_2 is new Integer
     with Atomic,
```

(continues on next page)

(continued from previous page)

```
        Volatile,
        Independent;

end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic.Volatile_
↪Independent
MD5: 3034c7a07698491f961d9b4fb74f03d8
```

A simular 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 19: 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;
```

3.4 Full-access only

Note

This feature was introduced in Ada 2022.

A full-access object is an object that requires that read or write operations on this object are performed by reading or writing all bits of the object (i.e. the *full object*) at once. Accordingly, a full-access type is a type whose objects follow this requirement. Note that a full-access type must be simultaneously a *volatile type* (page 139) or an *atomic type* (page 146). (In other words, if a type is neither volatile nor atomic, it cannot be a full-access type.)

Important

Just as a reminder, any atomic type is automatically also *volatile* (page 139) and *independent* (page 141).

Let's see some examples:

Listing 20: show_full_access_only_types.ads

```
package Show_Full_Access_Only_Types is

   type Nonatomic_Full_Access_Type is
```

(continues on next page)

(continued from previous page)

```
    new Long_Float
      with Volatile, Full_Access_Only;

   type Atomic_Full_Access_Type is
     new Long_Float
       with Atomic, Full_Access_Only;

end Show_Full_Access_Only_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↪Control.Full_Access_Only_Types
MD5: 6e7d4ee2e89b943d25319de9d8cebdcd
```

Likewise, we can define nonatomic and atomic full-access objects:

Listing 21: show_full_access_only_objects.ads

```
package Show_Full_Access_Only_Objects is

   Nonatomic_Full_Access_Obj : Long_Float
     with Volatile, Full_Access_Only;

   Atomic_Full_Access_Obj : Long_Float
     with Atomic, Full_Access_Only;

end Show_Full_Access_Only_Objects;
```

Relevant topics

- 9.10 Shared Variables[76]
- C.6 Shared Variable Control[77]

3.4.1 Nonatomic full-access

As we already know, the value of a volatile object may be constantly changing, so the compiler generates code to read the value of the volatile object from memory for each access. (In other words, the value cannot be stored in a register for further processing.)

In the case of nonatomic full-access objects, the value of the object must not only be read from memory or updated to memory every time, but those operations must also be performed for the complete record object — not just parts of it.

Consider the following example:

Listing 22: registers.ads

```
with System;

package Registers is

   type Boolean_Bit is new Boolean
     with Size => 1;

   type UInt1 is mod 2**1
```

(continues on next page)

[76] http://www.ada-auth.org/standards/22rm/html/RM-9-10.html
[77] http://www.ada-auth.org/standards/22rm/html/RM-C-6.html

(continued from previous page)

```
     with Size => 1;

   type UInt2 is mod 2**2
     with Size => 2;

   type UInt14 is mod 2**14
     with Size => 14;

   type Window_Register is record
      --  horizontal line count
      Horizontal_Cnt : UInt14 := 16#0#;

      --  unspecified
      Reserved_14_15 : UInt2  := 16#0#;

      --  vertical line count
      Vertical_Cnt   : UInt14 := 16#0#;

      --  refresh signalling
      Refresh_Needed : Boolean_Bit := False;

      --  unspecified
      Reserved_30    : UInt1  := 16#0#;
   end record
     with Size      => 32,
          Bit_Order => System.Low_Order_First,
          Volatile,
          Full_Access_Only;

   for Window_Register use record
      Horizontal_Cnt at 0 range 0 .. 13;
      Reserved_14_15 at 0 range 14 .. 15;
      Vertical_Cnt   at 0 range 16 .. 29;
      Refresh_Needed at 0 range 30 .. 30;
      Reserved_30    at 0 range 31 .. 31;
   end record;

   procedure Show (WR : Window_Register);

end Registers;
```

Listing 23: registers.adb

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

package body Registers is

   procedure Show (WR : Window_Register) is
   begin
      Put_Line ("WR = (Horizontal_Cnt => "
                & WR.Horizontal_Cnt'Image
                & ",");
      Put_Line ("       Vertical_Cnt   => "
                & WR.Vertical_Cnt'Image
                & ",");
      Put_Line ("       Refresh_Needed => "
                & WR.Refresh_Needed'Image
                & ")");
   end Show;

end Registers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↪Control.Nonatomic_Full_Access_Register
MD5: b825ec2dbbc54201203bf71e4e32fb57
```

In this example, we have a 32-bit register (of `Window_Register` type) that contains window information for a display:

position	0			
bits	#0 .. 13	#14 .. #15	#16 .. #29	#30 .. #31
component	Horizontal_Cnt	Reserved_14_15	Vertical_Cnt	Reserved_30_31

Let's use the `Window_Register` type from the `Registers` package in a test application:

Listing 24: show_register.adb

```
with Registers;   use Registers;

procedure Show_Register is
   WR : Window_Register;
begin
   --  Nonatomic full-access assignments
   WR.Horizontal_Cnt := 800;
   WR.Vertical_Cnt   := 600;
   WR.Refresh_Needed := True;

   Show (WR);
end Show_Register;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↪Control.Nonatomic_Full_Access_Register
MD5: 7ff302d6cb282a6276747e8e17f26dfd
```

Runtime output

```
WR = (Horizontal_Cnt =>  800,
      Vertical_Cnt   =>  600,
      Refresh_Needed => TRUE)
```

The example contains assignments such as `WR.Horizontal_Cnt := 800` and `WR.Vertical_Cnt:= 600`. Because `Window_Register` is a full-access type, these assignments are performed for the complete 32-bit register, even though we're updating just a single component of the record object.

Note that if `Window_Register` wasn't a *full-access* object, an assignment such as `WR.Horizontal_Cnt := 800` could be performed with a 16-bit operation. In fact, this is what a compiler would most probably select for this assignment, because that is more efficient than manipulating the entire object. Therefore, using a *full-access* object prevents the compiler from generating operations that could lead to unexpected results.

Whenever possible, this *full-access* assignment is performed in a single machine operation. However, if it's not possible to generate a single machine operation on the target machine, the compiler may generate multiple operations for the update of the record components.

Note that we could combine these two assignments into a single one using an aggregate:

Listing 25: show_register.adb

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

with Registers;   use Registers;

procedure Show_Register is
   WR : Window_Register;
begin
   --  Nonatomic full-access assignment
   --  using an aggregate:
   WR := (Horizontal_Cnt => 800,
          Vertical_Cnt   => 600,
          Refresh_Needed => True,
          others         => <>);

   Show (WR);
end Show_Register;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Nonatomic_Full_Access_Register
MD5: 9caf39e4a01ee1ec62f0b24747640c01
```

Runtime output

```
WR = (Horizontal_Cnt =>  800,
      Vertical_Cnt   =>  600,
      Refresh_Needed => TRUE)
```

Again, this assignment is performed for the complete 32-bit register — ideally, using a single 32-bit machine operation — by reading the value from the memory.

Let's add another statement to the code example:

Listing 26: show_register.adb

```ada
with Registers;   use Registers;

```

(continues on next page)

(continued from previous page)

```
procedure Show_Register is
   WR : Window_Register :=
          (Horizontal_Cnt => 800,
           Vertical_Cnt   => 600,
           Refresh_Needed => True,
           others         => <>);
begin
   WR := (Horizontal_Cnt =>
            WR.Horizontal_Cnt * 2,
          Vertical_Cnt   =>
            Wr.Vertical_Cnt   * 2,
          others         => <>);

   Show (WR);
end Show_Register;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↪Control.Nonatomic_Full_Access_Register
MD5: cc4e218aef11af34e6d3262084a5c9ce
```

Runtime output

```
WR = (Horizontal_Cnt =>  1600,
      Vertical_Cnt   =>  1200,
      Refresh_Needed => FALSE)
```

In this example, we have an initialization using the same aggregate as in the previous code example. We also have an assignment, in which we read the value of WR and use it in the calculation.

Delta aggregates

If we want to just change two components, but leave the information of other components untouched, we can use a *delta aggregate* (page 291). (Note that we haven't discussed the topic of delta aggregates yet: we'll do that *later on in this course* (page 291). However, in simple terms, we can use them to modify specific components of a record without changing the remaining components of the record.)

For example, we might want to update just the vertical count and indicate that update via the Refresh_Needed flag, but keep the same horizontal count:

Listing 27: show_registers.adb

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

with Registers;   use Registers;

procedure Show_Registers is
   WR : Window_Register :=
          (Horizontal_Cnt => 800,
           Vertical_Cnt   => 600,
           others         => <>);
begin
   --  Delta assignment
   WR := (WR with delta
               Vertical_Cnt   => 800,
               Refresh_Needed => True);

```

(continues on next page)

(continued from previous page)

```
   Show (WR);
end Show_Registers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↪Control.Nonatomic_Full_Access_Register
MD5: 29df44d4fb13539cbd6070c37c217f8a
```

Runtime output

```
WR = (Horizontal_Cnt =>  800,
      Vertical_Cnt   =>  800,
      Refresh_Needed => TRUE)
```

A delta assignment using an aggregate such as (`WR with delta ...`) includes reading the value of the complete 32-bit `WR` object from memory, changing the components specified after `with delta`, and writing the complete 32-bit `WR` object back to memory. The reason is that we need to retrieve the information that is supposed to remain intact — the `Horizontal_Cnt` and the reserved components — in order to write them back as a *full-access* operation.

3.4.2 Atomic full-access

As we already know, *atomic objects* (page 146) only accept atomic reads and updates, which — as a whole — are indivisible, i.e. they must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes. (Again, if an object is atomic, this implies it is also volatile.)

In the case of atomic full-access objects, the complete object must be read and updated. Ideally, this operation corresponds to a single atomic operation on the target machine, but it can also translate to multiple atomic operations.

Let's adapt the previous example to illustrate this. First, we adapt the type in the package:

Listing 28: registers.ads

```
with System;

package Registers is

   type Boolean_Bit is new Boolean
     with Size => 1;

   type UInt1 is mod 2**1
     with Size => 1;

   type UInt2 is mod 2**2
     with Size => 2;

   type UInt14 is mod 2**14
     with Size => 14;

   type Window_Register is record
      --  horizontal line count
      Horizontal_Cnt : UInt14 := 16#0#;

      --  unspecified
      Reserved_14_15 : UInt2  := 16#0#;

```

(continues on next page)

(continued from previous page)

```
      --  vertical line count
      Vertical_Cnt   : UInt14 := 16#0#;

      --  refresh signalling
      Refresh_Needed : Boolean_Bit := False;

      --  unspecified
      Reserved_30    : UInt1  := 16#0#;
   end record
     with Size      => 32,
         Bit_Order => System.Low_Order_First,
         Atomic,
         Full_Access_Only;

   for Window_Register use record
      Horizontal_Cnt at 0 range 0 .. 13;
      Reserved_14_15 at 0 range 14 .. 15;
      Vertical_Cnt   at 0 range 16 .. 29;
      Refresh_Needed at 0 range 30 .. 30;
      Reserved_30    at 0 range 31 .. 31;
   end record;

   procedure Show (WR : Window_Register);

end Registers;
```

Listing 29: registers.adb

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

package body Registers is

   procedure Show (WR : Window_Register) is
   begin
      Put_Line ("WR = (Horizontal_Cnt => "
                & WR.Horizontal_Cnt'Image
                & ",");
      Put_Line ("      Vertical_Cnt   => "
                & WR.Vertical_Cnt'Image
                & ",");
      Put_Line ("      Refresh_Needed => "
                & WR.Refresh_Needed'Image
                & ")");
   end Show;

end Registers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↪Control.Atomic_Full_Access_Register
MD5: dc088d1b0df1af5086a1ae8b46bb6d4d
```

We then use the package in our test application:

Listing 30: show_register.adb

```
with Registers;   use Registers;

procedure Show_Register is
   WR : Window_Register :=
```

(continues on next page)

(continued from previous page)

```
          (Horizontal_Cnt => 800,
           Vertical_Cnt   => 600,
           Refresh_Needed => True,
           others         => <>);
begin
   WR := (Horizontal_Cnt =>
            WR.Horizontal_Cnt * 2,
          Vertical_Cnt   =>
            Wr.Vertical_Cnt   * 2,
          others         => <>);

   Show (WR);
end Show_Register;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↪Control.Atomic_Full_Access_Register
MD5: cc4e218aef11af34e6d3262084a5c9ce
```

In this example, we first have an atomic initialization of WR using an aggregate. Then, we have an atomic assignment to the atomic full-access object WR. Because its type is an atomic full-access type, the operations are atomic operations that always access the full object from and to memory.

3.4.3 Comparison: full-access and non-full-access types

An interesting exercise for the reader is to compare the Assembly code generated for the code example above with a version of this code where the `Window_Register` is not a full-access type.

Relevant topics

On a Linux platform, you can use *objdump* to retrieve the Assembly code and *diff* to see the difference between both versions of the type. For example:

```
objdump --target=elf64-x86-64 -d -S ./show_register > full_access.txt

#  [...]

diff --width=80 -t -y full_access.txt no_full_access.txt
```

By doing this kind of comparisons, you might gain more insights on the impact of the `Full_Access_Only` aspect.

For further reading...

By running on a PC, we can compare the Intel Assembly[78] code for various versions of the code. Let's start with the version using a nonatomic full-access version of `Window_Register` vs. the nonatomic (non-full-access) version of `Window_Register`:

```
type Window_Register is record
   --  [...]
end record
  with Size      => 32,
       Bit_Order => System.Low_Order_First,
       Volatile,
       Full_Access_Only;
```

```ada
type Window_Register is record
   --  [...]
end record
  with Size      => 32,
       Bit_Order => System.Low_Order_First,
       Volatile;
```

These are the manually-adapted differences between both versions:

```
--  Volatile, Full_Access_Only         |  --  Volatile

procedure Show_Register is                procedure Show_Register is
    push   %rbp                               push   %rbp
    mov    %rsp,%rbp                          mov    %rsp,%rbp
    sub    $0x20,%rsp                  |      sub    $0x10,%rsp
   WR : Window_Register :=                   WR : Window_Register :=
          (Horizontal_Cnt => 800,                   (Horizontal_Cnt => 800,
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0xffffc000,%eax                   and    $0xffffc000,%eax
    or     $0x320,%eax                        or     $0x320,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0x3f,%ah                          and    $0x3f,%ah
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0xc000ffff,%eax                   and    $0xc000ffff,%eax
    or     $0x2580000,%eax                    or     $0x2580000,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    or     $0x40000000,%eax                   or     $0x40000000,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0x7fffffff,%eax                   and    $0x7fffffff,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax             <
    mov    %eax,-0x14(%rbp)            <
    mov    -0x14(%rbp),%eax            <
    mov    %eax,-0x8(%rbp)             <
           Vertical_Cnt   => 600,                    Vertical_Cnt   => 600,
           Refresh_Needed => True,                   Refresh_Needed => True,
           others         => <>);                    others         => <>);
begin                                     begin
   WR := (Horizontal_Cnt =>                  WR := (Horizontal_Cnt =>
            WR.Horizontal_Cnt * 2,                    WR.Horizontal_Cnt * 2,
    mov    -0x8(%rbp),%eax             |      mov    -0x4(%rbp),%eax
    mov    %eax,%ecx                   <
    and    $0x3fff,%cx                 |      and    $0x3fff,%ax
                                       >      add    %eax,%eax
   WR := (Horizontal_Cnt =>            <
    mov    -0xc(%rbp),%eax             <
    mov    %eax,%edx                   <
            WR.Horizontal_Cnt * 2,     <
    lea    (%rcx,%rcx,1),%eax          <
    and    $0x3fff,%ax                        and    $0x3fff,%ax
   WR := (Horizontal_Cnt =>                  WR := (Horizontal_Cnt =>
    movzwl %ax,%eax                           movzwl %ax,%eax
    and    $0x3fff,%eax                       and    $0x3fff,%eax
    and    $0xffffc000,%edx            <
    or     %edx,%eax                   <
    mov    %eax,%edx                          mov    %eax,%edx
    mov    %edx,%eax                   |      mov    -0x8(%rbp),%eax
    mov    %eax,-0xc(%rbp)             |      and    $0xffffc000,%eax
```

```
    mov    -0xc(%rbp),%eax              |      or     %edx,%eax
                                        >      mov    %eax,-0x8(%rbp)
                                        >      mov    -0x8(%rbp),%eax
    and    $0x3f,%ah                           and    $0x3f,%ah
    mov    %eax,-0xc(%rbp)              |      mov    %eax,-0x8(%rbp)
          Vertical_Cnt   =>                          Vertical_Cnt   =>
            Wr.Vertical_Cnt   * 2,                     Wr.Vertical_Cnt   * 2,
    mov    -0x8(%rbp),%eax              |      mov    -0x4(%rbp),%eax
    shr    $0x10,%eax                          shr    $0x10,%eax
    mov    %eax,%ecx                    |      and    $0x3fff,%ax
    and    $0x3fff,%cx                  |      add    %eax,%eax
   WR := (Horizontal_Cnt =>             <
    mov    -0xc(%rbp),%eax              <
    mov    %eax,%edx                    <
            Wr.Vertical_Cnt   * 2,      <
    lea    (%rcx,%rcx,1),%eax           <
    and    $0x3fff,%ax                         and    $0x3fff,%ax
   WR := (Horizontal_Cnt =>                   WR := (Horizontal_Cnt =>
    movzwl %ax,%eax                            movzwl %ax,%eax
    and    $0x3fff,%eax                        and    $0x3fff,%eax
    shl    $0x10,%eax                          shl    $0x10,%eax
    and    $0xc000ffff,%edx             <
    or     %edx,%eax                    <
    mov    %eax,%edx                           mov    %eax,%edx
    mov    %edx,%eax                    |      mov    -0x8(%rbp),%eax
    mov    %eax,-0xc(%rbp)              |      and    $0xc000ffff,%eax
    mov    -0xc(%rbp),%eax              |      or     %edx,%eax
                                        >      mov    %eax,-0x8(%rbp)
                                        >      mov    -0x8(%rbp),%eax
    and    $0xbfffffff,%eax                    and    $0xbfffffff,%eax
    mov    %eax,-0xc(%rbp)              |      mov    %eax,-0x8(%rbp)
    mov    -0xc(%rbp),%eax              |      mov    -0x8(%rbp),%eax
    and    $0x7fffffff,%eax                    and    $0x7fffffff,%eax
    mov    %eax,-0xc(%rbp)              <
    mov    -0xc(%rbp),%eax              <
    mov    %eax,-0x8(%rbp)                     mov    %eax,-0x8(%rbp)
                                        >      mov    -0x8(%rbp),%eax
                                        >      mov    %eax,-0x4(%rbp)
          others         => <>);                     others         => <>);
```

As we can see, although parts of the Assembly code are the same or look very similar, there are some differences between both versions. These differences are mostly related to the fact that we have to operate on the full object when reading it from memory.

Likewise, we can compare the Assembly code for the atomic full-access version of `Window_Register` vs. the atomic (non-full-access) version of `Window_Register`:

```ada
type Window_Register is record
   --  [...]
end record
  with Size      => 32,
       Bit_Order => System.Low_Order_First,
       Atomic,
       Full_Access_Only;

type Window_Register is record
   --  [...]
end record
  with Size      => 32,
       Bit_Order => System.Low_Order_First,
       Atomic;
```

These are the manually-adapted differences between both versions:

```
--  Atomic, Full_Access_Only            |  --  Atomic

procedure Show_Register is                 procedure Show_Register is
    push   %rbp                                push   %rbp
    mov    %rsp,%rbp                           mov    %rsp,%rbp
    sub    $0x20,%rsp                   |      sub    $0x10,%rsp
   WR : Window_Register :=                    WR : Window_Register :=
          (Horizontal_Cnt => 800,                    (Horizontal_Cnt => 800,
    mov    -0x4(%rbp),%eax                     mov    -0x4(%rbp),%eax
    and    $0xffffc000,%eax                    and    $0xffffc000,%eax
    or     $0x320,%eax                         or     $0x320,%eax
    mov    %eax,-0x4(%rbp)                     mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                     mov    -0x4(%rbp),%eax
    and    $0x3f,%ah                           and    $0x3f,%ah
    mov    %eax,-0x4(%rbp)                     mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                     mov    -0x4(%rbp),%eax
    and    $0xc000ffff,%eax                    and    $0xc000ffff,%eax
    or     $0x2580000,%eax                     or     $0x2580000,%eax
    mov    %eax,-0x4(%rbp)                     mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                     mov    -0x4(%rbp),%eax
    or     $0x40000000,%eax                    or     $0x40000000,%eax
    mov    %eax,-0x4(%rbp)                     mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                     mov    -0x4(%rbp),%eax
    and    $0x7fffffff,%eax                    and    $0x7fffffff,%eax
    mov    %eax,-0x4(%rbp)                     mov    %eax,-0x4(%rbp)
   WR : Window_Register :=                    WR : Window_Register :=
    mov    -0x4(%rbp),%eax                     mov    -0x4(%rbp),%eax
    mov    %eax,-0x14(%rbp)             <
    mov    -0x14(%rbp),%eax             <
    mov    %eax,-0x8(%rbp)                     mov    %eax,-0x8(%rbp)
           Vertical_Cnt   => 600,                     Vertical_Cnt   => 600,
           Refresh_Needed => True,                    Refresh_Needed => True,
           others         => <>);                     others         => <>);
begin                                      begin
   WR := (Horizontal_Cnt =>                   WR := (Horizontal_Cnt =>
            WR.Horizontal_Cnt * 2,                     WR.Horizontal_Cnt * 2,
    mov    -0x8(%rbp),%eax                     mov    -0x8(%rbp),%eax
    mov    %eax,%ecx                    <
    and    $0x3fff,%cx                  |      and    $0x3fff,%ax
                                        |      add    %eax,%eax

   WR := (Horizontal_Cnt =>             <
    mov    -0xc(%rbp),%eax              <
    mov    %eax,%edx                    <
            WR.Horizontal_Cnt * 2,      <
    lea    (%rcx,%rcx,1),%eax           <
    and    $0x3fff,%ax                         and    $0x3fff,%ax
   WR := (Horizontal_Cnt =>                   WR := (Horizontal_Cnt =>
    movzwl %ax,%eax                            movzwl %ax,%eax
    and    $0x3fff,%eax                        and    $0x3fff,%eax
                                        >      mov    %eax,%edx
                                        >      mov    -0xc(%rbp),%eax
    and    $0xffffc000,%edx             |      and    $0xffffc000,%eax
    or     %edx,%eax                           or     %edx,%eax
    mov    %eax,%edx                    <
    mov    %edx,%eax                    <
    mov    %eax,-0xc(%rbp)                     mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                     mov    -0xc(%rbp),%eax
    and    $0x3f,%ah                           and    $0x3f,%ah
    mov    %eax,-0xc(%rbp)                     mov    %eax,-0xc(%rbp)
          Vertical_Cnt   =>                          Vertical_Cnt   =>
            Wr.Vertical_Cnt   * 2,                     Wr.Vertical_Cnt   * 2,
    mov    -0x8(%rbp),%eax                     mov    -0x8(%rbp),%eax
```

```
    shr    $0x10,%eax                          shr    $0x10,%eax
    mov    %eax,%ecx                   <
    and    $0x3fff,%cx                 |       and    $0x3fff,%ax
                                       >       add    %eax,%eax
   WR := (Horizontal_Cnt =>            <
    mov    -0xc(%rbp),%eax             <
    mov    %eax,%edx                   <
            Wr.Vertical_Cnt   * 2,     <
    lea    (%rcx,%rcx,1),%eax          <
    and    $0x3fff,%ax                         and    $0x3fff,%ax
   WR := (Horizontal_Cnt =>                   WR := (Horizontal_Cnt =>
    movzwl %ax,%eax                            movzwl %ax,%eax
    and    $0x3fff,%eax                        and    $0x3fff,%eax
    shl    $0x10,%eax                          shl    $0x10,%eax
                                       >       mov    %eax,%edx
                                       >       mov    -0xc(%rbp),%eax
    and    $0xc000ffff,%edx            |       and    $0xc000ffff,%eax
    or     %edx,%eax                           or     %edx,%eax
    mov    %eax,%edx                   <
    mov    %edx,%eax                   <
    mov    %eax,-0xc(%rbp)                     mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                     mov    -0xc(%rbp),%eax
    and    $0xbfffffff,%eax                    and    $0xbfffffff,%eax
    mov    %eax,-0xc(%rbp)                     mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                     mov    -0xc(%rbp),%eax
    and    $0x7fffffff,%eax                    and    $0x7fffffff,%eax
    mov    %eax,-0xc(%rbp)                     mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                     mov    -0xc(%rbp),%eax
    xchg   %eax,-0x8(%rbp)                     xchg   %eax,-0x8(%rbp)
          others         => <>);                     others         => <>);
```

Again, there are some differences between both versions, even though some parts of the Assembly code are the same or look very similar.

Finally, we might want to compare the nonatomic full-access version vs. the atomic full-access version of the `Window_Register` type:

```
type Window_Register is record
   --  [...]
end record
  with Size      => 32,
       Bit_Order => System.Low_Order_First,
       Volatile,
       Full_Access_Only;

type Window_Register is record
   --  [...]
end record
  with Size      => 32,
       Bit_Order => System.Low_Order_First,
       Atomic,
       Full_Access_Only;
```

These are the differences between both versions:

```
--  Volatile, Full_Access_Only         |  --  Atomic, Full_Access_Only

procedure Show_Register is                procedure Show_Register is
    push   %rbp                               push   %rbp
    mov    %rsp,%rbp                          mov    %rsp,%rbp
    sub    $0x20,%rsp                         sub    $0x20,%rsp
   WR : Window_Register :=                   WR : Window_Register :=
          (Horizontal_Cnt => 800,                   (Horizontal_Cnt => 800,
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0xffffc000,%eax                   and    $0xffffc000,%eax
```

```
    or     $0x320,%eax                        or     $0x320,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0x3f,%ah                          and    $0x3f,%ah
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0xc000ffff,%eax                   and    $0xc000ffff,%eax
    or     $0x2580000,%eax                    or     $0x2580000,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    or     $0x40000000,%eax                   or     $0x40000000,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    and    $0x7fffffff,%eax                   and    $0x7fffffff,%eax
    mov    %eax,-0x4(%rbp)                    mov    %eax,-0x4(%rbp)
   WR : Window_Register :=                   WR : Window_Register :=
    mov    -0x4(%rbp),%eax                    mov    -0x4(%rbp),%eax
    mov    %eax,-0x14(%rbp)                   mov    %eax,-0x14(%rbp)
    mov    -0x14(%rbp),%eax                   mov    -0x14(%rbp),%eax
    mov    %eax,-0x8(%rbp)                    mov    %eax,-0x8(%rbp)
           Vertical_Cnt   => 600,                    Vertical_Cnt   => 600,
           Refresh_Needed => True,                   Refresh_Needed => True,
           others         => <>);                    others         => <>);
begin                                     begin
   WR := (Horizontal_Cnt =>                  WR := (Horizontal_Cnt =>
            WR.Horizontal_Cnt * 2,                    WR.Horizontal_Cnt * 2,
    mov    -0x8(%rbp),%eax                    mov    -0x8(%rbp),%eax
    mov    %eax,%ecx                          mov    %eax,%ecx
    and    $0x3fff,%cx                        and    $0x3fff,%cx
   WR := (Horizontal_Cnt =>                  WR := (Horizontal_Cnt =>
    mov    -0xc(%rbp),%eax                    mov    -0xc(%rbp),%eax
    mov    %eax,%edx                          mov    %eax,%edx
            WR.Horizontal_Cnt * 2,                    WR.Horizontal_Cnt * 2,
    lea    (%rcx,%rcx,1),%eax                 lea    (%rcx,%rcx,1),%eax
    and    $0x3fff,%ax                        and    $0x3fff,%ax
   WR := (Horizontal_Cnt =>                  WR := (Horizontal_Cnt =>
    movzwl %ax,%eax                           movzwl %ax,%eax
    and    $0x3fff,%eax                       and    $0x3fff,%eax
    and    $0xffffc000,%edx                   and    $0xffffc000,%edx
    or     %edx,%eax                          or     %edx,%eax
    mov    %eax,%edx                          mov    %eax,%edx
    mov    %edx,%eax                          mov    %edx,%eax
    mov    %eax,-0xc(%rbp)                    mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                    mov    -0xc(%rbp),%eax
    and    $0x3f,%ah                          and    $0x3f,%ah
    mov    %eax,-0xc(%rbp)                    mov    %eax,-0xc(%rbp)
          Vertical_Cnt   =>                         Vertical_Cnt   =>
            Wr.Vertical_Cnt   * 2,                    Wr.Vertical_Cnt   * 2,
    mov    -0x8(%rbp),%eax                    mov    -0x8(%rbp),%eax
    shr    $0x10,%eax                         shr    $0x10,%eax
    mov    %eax,%ecx                          mov    %eax,%ecx
    and    $0x3fff,%cx                        and    $0x3fff,%cx
   WR := (Horizontal_Cnt =>                  WR := (Horizontal_Cnt =>
    mov    -0xc(%rbp),%eax                    mov    -0xc(%rbp),%eax
    mov    %eax,%edx                          mov    %eax,%edx
            Wr.Vertical_Cnt   * 2,                    Wr.Vertical_Cnt   * 2,
    lea    (%rcx,%rcx,1),%eax                 lea    (%rcx,%rcx,1),%eax
    and    $0x3fff,%ax                        and    $0x3fff,%ax
   WR := (Horizontal_Cnt =>                  WR := (Horizontal_Cnt =>
    movzwl %ax,%eax                           movzwl %ax,%eax
    and    $0x3fff,%eax                       and    $0x3fff,%eax
    shl    $0x10,%eax                         shl    $0x10,%eax
```

```
    and    $0xc000ffff,%edx                  and    $0xc000ffff,%edx
    or     %edx,%eax                         or     %edx,%eax
    mov    %eax,%edx                         mov    %eax,%edx
    mov    %edx,%eax                         mov    %edx,%eax
    mov    %eax,-0xc(%rbp)                   mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                   mov    -0xc(%rbp),%eax
    and    $0xbfffffff,%eax                  and    $0xbfffffff,%eax
    mov    %eax,-0xc(%rbp)                   mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                   mov    -0xc(%rbp),%eax
    and    $0x7fffffff,%eax                  and    $0x7fffffff,%eax
    mov    %eax,-0xc(%rbp)                   mov    %eax,-0xc(%rbp)
    mov    -0xc(%rbp),%eax                   mov    -0xc(%rbp),%eax
    mov    %eax,-0x8(%rbp)            |      xchg   %eax,-0x8(%rbp)
          others         => <>);                   others         => <>);
```

As we can see, the code is basically the same — except for the last Assembly instruction, which is a *mov* instruction in the volatile version and an *xchg* instruction in the atomic version — which is an atomic instruction on this platform.

3.5 Atomic operations

Note

This feature was introduced in Ada 2022.

Ada offers four packages to handle atomic operations. Those packages are child packages of the `System.Atomic_Operations` package. We will discuss each of those package individually in this section.

Relevant topics

- C.6.1 The Package System.Atomic_Operations[79]

3.5.1 Atomic Exchange

The generic `System.Atomic_Operations.Exchange` package provides operations to compare and exchange objects atomically.

Atomic_Exchange function

One of those operations is the `Atomic_Exchange` function, which performs the following operations atomically:

```
function Atomic_Exchange
  (Item  : aliased in out Atomic_Type;
   Value :                Atomic_Type)
   return Atomic_Type
is
   Old_Item : Atomic_Type := Item;
begin
   Item := Value;
```

(continues on next page)

[78] https://en.wikipedia.org/wiki/X86_instruction_listings
[79] http://www.ada-auth.org/standards/22rm/html/RM-C-6-1.html

(continued from previous page)

```
   return Old_Item;
end Atomic_Exchange;
```

As mentioned in the Ada Reference Manual[80], we can use this function to implement a spinlock[81]. For example:

Listing 31: spinlocks.ads

```
with System.Atomic_Operations.Exchange;

package Spinlocks is

   type Lock is new Boolean with Atomic;

   package Lock_Exchange is new
     System.Atomic_Operations.Exchange (Lock);

end Spinlocks;
```

Listing 32: show_locks.adb

```
with Spinlocks;
use Spinlocks;
use Spinlocks.Lock_Exchange;

procedure Show_Locks is
   L : aliased Lock := False;
begin
   --  Get the lock
   while Atomic_Exchange (Item  => L,
                          Value => True) loop
      null;
   end loop;

   --  At this point, we got the lock.
   --  Do some stuff here...

   --  Release the lock.
   L := False;
end Show_Locks;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
↪Exchange
MD5: 36699b917485f14c4e8a905a6c48027b
```

In this example, we call the `Atomic_Exchange` function for the L lock until we get it. Then, we can use the resource that we protected via the lock. After we finish our work, we can release the lock by setting L to **False**.

Note that `System.Atomic_Operations.Exchange` is a generic package, so we have to instantiate it for a specific atomic type — in this case, the atomic Boolean Lock type.

We can use multiple tasks to illustrate a situation where using a lock is important to ensure that no race conditions[82] occur:

[80] http://www.ada-auth.org/standards/22rm/html/RM-C-6-2.html
[81] https://en.wikipedia.org/wiki/Spinlock
[82] https://en.wikipedia.org/wiki/Race_condition

Listing 33: spinlocks.ads

```
with System.Atomic_Operations.Exchange;

package Spinlocks is

   type Lock is new Boolean with Atomic;

   package Lock_Exchange is new
     System.Atomic_Operations.Exchange (Lock);

end Spinlocks;
```

Listing 34: show_locks.adb

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

with Spinlocks;
use Spinlocks;
use Spinlocks.Lock_Exchange;

procedure Show_Locks is
   L          : aliased Lock := False;
   Task_Count : Integer      := 0;

   task type A_Task;

   task body A_Task is
      Task_Number : Integer;
   begin
      --  Get the lock
      while Atomic_Exchange (Item  => L,
                             Value => True) loop
         null;
      end loop;

      --  At this point, we got the lock.
      Task_Count  := Task_Count + 1;
      Task_Number := Task_Count;

      --  Release the lock.
      L := False;

      Put_Line ("Task_Number: "
                & Task_Number'Image);

   end A_Task;

   A, B, C, D, E, F : A_Task;
begin
   null;
end Show_Locks;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↪Exchange
MD5: af7aad741c20be1e8433b04def90dcdb
```

Runtime output

```
Task_Number:  1
Task_Number:  2
Task_Number:  3
Task_Number:  4
Task_Number:  5
Task_Number:  6
```

In this example, we create multiple tasks (A, B, C, D, E, F) and initialize the Task_Number of each task based on the value of the Task_Count variable. To avoid multiple tasks accessing the Task_Count variable at the same time, we use the L lock, which we get before updating the Task_Count.

Atomic_Compare_And_Exchange function

Another function from the System.Atomic_Operations.Exchange package is Atomic_Compare_And_Exchange, which performs the following operations atomically:

```
function Atomic_Compare_And_Exchange
  (Item    : aliased in out Atomic_Type;
   Prior   : aliased in out Atomic_Type;
   Desired :                Atomic_Type)
   return Boolean is
 begin
    if Item = Prior then
       Item := Value;
       --  The item is only updated if its
       --  value and the prior value match

       return True;
    else
       Prior := Item;
       return False;
    end if;
 end Atomic_Exchange;
```

This function can be used for lazy initialization[83] of variables. For example, consider an application with multiple tasks that make use of a certain value that isn't initialized at its declaration, but at a later point in time by an arbitrary task. We can use Atomic_Compare_And_Exchange to ensure that we only update that value if it wasn't already initialized.

Let's start with the package specification:

Listing 35: lazy_initialization.ads

```
with System.Atomic_Operations.Exchange;
with Ada.Numerics.Discrete_Random;

package Lazy_Initialization is

   subtype Lazy_Value_Total_Range is
     Integer range 99 .. 1000;

   Lazy_Value_Default_Value : constant
      := Lazy_Value_Total_Range'First;

   subtype Lazy_Value_Range is Integer
     range Lazy_Value_Default_Value + 1 ..
           Lazy_Value_Total_Range'Last;
```

(continues on next page)

[83] https://en.wikipedia.org/wiki/Lazy_initialization

(continued from previous page)

```

   type Lazy_Value is new Lazy_Value_Total_Range
     with Atomic,
          Default_Value =>
            Lazy_Value_Default_Value;

   package Value_Exchange is new
     System.Atomic_Operations.Exchange
       (Lazy_Value);

   package Lazy_Value_Random is new
     Ada.Numerics.Discrete_Random
       (Lazy_Value_Range);

end Lazy_Initialization;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
↪Compare_And_Exchange
MD5: 09d49998aa7e3d5c0cfb4b74af8e542b
```

In this package, we declare the Lazy_Value type with a default value (specified by the Lazy_Value_Default_Value constant). Note that we have two ranges here: Lazy_Value_Total_Range and Lazy_Value_Range. We use the Lazy_Value_Total_Range in the declaration of the Lazy_Value type: it indicates the *total range* of the type. We use the Lazy_Value_Range as a constraint for the total range. This range doesn't contain the default value (Lazy_Value_Default_Value), and we use it to indicate the valid values of the type. (We discuss the application of Lazy_Value_Range later on.)

Also, in addition to instantiating the System.Atomic_Operations.Exchange package, we instantiate the Ada.Numerics.Discrete_Random package, which we'll use to generate random numbers in the expected range (Lazy_Value_Range) for the Lazy_Value type. (We discussed the Ada.Numerics.Discrete_Random package in the Introduction to Ada[84] course.)

Let's use this package in the Show_Lazy_Initialization procedure:

Listing 36: show_lazy_initialization.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Discrete_Random;

with Lazy_Initialization;
use Lazy_Initialization;
use Lazy_Initialization.Value_Exchange;

procedure Show_Lazy_Initialization is
  subtype A_Task_Number is Natural;

   Value            : aliased Lazy_Value;
   Value_Modified_By : A_Task_Number := 0;

   task type A_Task is
      entry Start (This : A_Task_Number);
      entry Stop;
   end A_Task;

   task body A_Task is
```

(continues on next page)

[84] https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_numerics.html#intro-ada-random-number-generation

(continued from previous page)

```
      Task_Number : A_Task_Number;
   begin
      accept Start (This : A_Task_Number) do
         Task_Number := This;
      end Start;

      Sleep_Some_Time : declare
         subtype Sleep_Range is
           Integer range 1 .. 3;

         package Random_Sleep is new
           Ada.Numerics.Discrete_Random
             (Sleep_Range);
         use Random_Sleep;

         G : Generator;
      begin
         Reset (G);
         delay Duration (Random (G));
      end Sleep_Some_Time;

      Generate_Value : declare
         use Lazy_Value_Random;

         G             :         Generator;
         Initial_Value :         Lazy_Value_Range;
         Prior         : aliased Lazy_Value;
      begin
         Reset (G);
         Initial_Value := Random (G);

         if Atomic_Compare_And_Exchange
           (Item    => Value,
            Prior   => Prior,
            Desired => Lazy_Value (Initial_Value))
         then
            Value_Modified_By := Task_Number;
         end if;

      end Generate_Value;

      accept Stop do
         Put_Line ("Current task number:     "
                   & Task_Number'Image);
         Put_Line ("Value:                   "
                   & Value'Image);
         Put_Line ("Modified by task number: "
                   & Value_Modified_By'Image);
         Put_Line ("---------------------");
      end Stop;
   end A_Task;

   Some_Tasks : array (1 .. 5) of A_Task;
begin
   for I in Some_Tasks'Range loop
      Some_Tasks (I).Start (I);
   end loop;
   for I in Some_Tasks'Range loop
      Some_Tasks (I).Stop;
   end loop;
end Show_Lazy_Initialization;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
↪Compare_And_Exchange
MD5: 9cc898edd767f8bbcfe2c81e7ca0e442
```

Runtime output

```
Current task number:      1
Value:                  428
Modified by task number:  5
---------------------
Current task number:      2
Value:                  428
Modified by task number:  5
---------------------
Current task number:      3
Value:                  428
Modified by task number:  5
---------------------
Current task number:      4
Value:                  428
Modified by task number:  5
---------------------
Current task number:      5
Value:                  428
Modified by task number:  5
---------------------
```

In the `Show_Lazy_Initialization` procedure, the most important variable is `Value`, which is the variable we have to protect via a lock. In addition, we have the auxiliary `Value_Modified_By` variable, which indicates the number of the task that initialized the `Value` variable.

In this procedure, we also see two main *block statements* (page 460):

- the block statement with the `Sleep_Some_Time` identifier, where we make the task *sleep* for a random amount of time (in the `Sleep_Range` range); and
- the block statement with the `Generate_Value` identified, where we generate a new value randomly and attempt to update the `Value` variable (of `Lazy_Value` type).

Let's discuss some details about the `Generate_Value` block statement. We start by declaring some variables. Here, it's important to highlight that the `Prior` variable is initialized with the default value (`Lazy_Value_Default_Value`). We then call the `Atomic_Compare_And_Exchange` function, and pass `Value` and `Prior` as actual parameters. We can have two possible outcomes:

1. If `Value` hasn't been modified by a task yet, it will contain the default value — which means that the values of the `Prior` and `Value` variables match. In this case, the call to `Atomic_Compare_And_Exchange` will update the `Value` variable and return **True**. (Note that we also update the `Value_Modified_By` variable when `Atomic_Compare_And_Exchange` returns **True**.)
2. If `Value` has already been modified by a task, its value doesn't match the (default) value of `Prior` anymore, so the call to `Atomic_Compare_And_Exchange` doesn't modify the `Value` variable.

As mentioned before, we use a stricter range for the random number generator: the `Lazy_Value_Range`. Because this range doesn't contain the default value (`Lazy_Value_Default_Value`), we will never generate a random value that matches the default value.

Relevant topics

- C.6.2 The Package System.Atomic_Operations.Exchange[85]

Atomic Test and Set

The `System.Atomic_Operations.Test_And_Set` package provides atomic operations to set and clear atomic flags. To declare flags, we use the `Test_And_Set_Flag` type. The following operations are available:

1. the `Atomic_Test_And_Set` function, which we call to verify whether the flag can be set and, if positive, set it accordingly.
   - The function returns **True** if the flag has been set, and **False** otherwise.
2. the `Atomic_Clear` procedure, which we call to clear the flag.

We can use these functions to implement an application similar to the *spinlocks* (page 163) that we've seen before:

Listing 37: show_test_and_set.adb

```
with System.Atomic_Operations.Test_And_Set;
use  System.Atomic_Operations.Test_And_Set;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Test_And_Set is
   Lock       : aliased Test_And_Set_Flag;
   Task_Count : Integer := 0;

   task type A_Task;

   task body A_Task is
      Task_Number : Integer;
   begin
      --  Get the lock
      while Atomic_Test_And_Set (Lock) loop
         null;
      end loop;

      --  At this point, we got the lock.
      Task_Count  := Task_Count + 1;
      Task_Number := Task_Count;

      --  Release the lock.
      Atomic_Clear (Lock);

      Put_Line ("Task_Number: "
                & Task_Number'Image);

   end A_Task;

   A, B, C, D, E, F : A_Task;
begin
   null;
end Show_Test_And_Set;
```

Code block metadata

[85] http://www.ada-auth.org/standards/22rm/html/RM-C-6-2.html

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
↪Test_And_Set
MD5: 45814e2e157d3fd45f876c89914a5cc5
```

Runtime output

```
Task_Number:  1
Task_Number:  2
Task_Number:  3
Task_Number:  4
Task_Number:  5
Task_Number:  6
```

Here, we call `Atomic_Test_And_Set` in a loop until it returns **True**. Then, we update the **Task_Count** and **Task_**`Number`. When we're finished, we call the `Atomic_Clear` procedure to release the lock.

> **Relevant topics**
>
> - C.6.3 The Package System.Atomic_Operations.Test_and_Set[86]

Atomic Operations using Integer Arithmetic

The generic `System.Atomic_Operations.Integer_Arithmetic` package is used to perform atomic operations on atomic integer types. It provides the following operations: the procedures `Atomic_Add` and `Atomic_Subtract`, and the functions `Atomic_Fetch_And_Add` and `Atomic_Fetch_And_Subtract`. The procedures and the corresponding `Atomic_Fetch_` functions do basically the same thing, with the difference that `Atomic_Fetch` functions return the previous (older) value of the input item.

The `Atomic_Add` procedure performs the following operations atomically:

```
procedure Atomic_Add
  (Item  : aliased in out Atomic_Type;
   Value :                Atomic_Type) is
begin
   Item := Item + Value;
end Atomic_Add;
```

The corresponding `Atomic_Fetch_And_Add` function performs the following operations atomically:

```
function Atomic_Fetch_And_Add
  (Item  : aliased in out Atomic_Type;
   Value :                Atomic_Type)
   return Atomic_Type
is
   Old_Item : Atomic_Type := Item;
begin
   Item := Item + Value;
   return Old_Item;
end Atomic_Fetch_And_Add;
```

The `Atomic_Subtract` procedure performs the following operations atomically:

```
procedure Atomic_Subtract
  (Item  : aliased in out Atomic_Type;
```

(continues on next page)

86 http://www.ada-auth.org/standards/22rm/html/RM-C-6-3.html

(continued from previous page)

```
   Value :                 Atomic_Type) is
begin
   Item := Item - Value;
end Atomic_Subtract;
```

The corresponding Atomic_Fetch_And_Subtract function performs the following operations atomically:

```
function Atomic_Fetch_And_Subtract
  (Item  : aliased in out Atomic_Type;
   Value :                 Atomic_Type)
   return Atomic_Type
is
   Old_Item : Atomic_Type := Item;
begin
   Item := Item - Value;
   return Old_Item;
end Atomic_Fetch_And_Subtract;
```

Let's reuse a *previous code example* (page 163) that sets a unique number for each task. In this case, instead of using locks, we use the atomic operations from the System.Atomic_Operations.Integer_Arithmetic package:

Listing 38: atomic_integers.ads

```
with System.Atomic_Operations.Integer_Arithmetic;

package Atomic_Integers is

   type Atomic_Integer is new Integer
     with Atomic;

   package Atomic_Integer_Arithmetic is new
     System.Atomic_Operations.Integer_Arithmetic
       (Atomic_Integer);

end Atomic_Integers;
```

Listing 39: show_atomic_integers.adb

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

with Atomic_Integers;
use  Atomic_Integers;
use  Atomic_Integers.Atomic_Integer_Arithmetic;

procedure Show_Atomic_Integers is
   Task_Count : aliased Atomic_Integer := 0;

   task type A_Task;

   task body A_Task is
      Task_Number : Atomic_Integer;
   begin
      Task_Number :=
        Atomic_Fetch_And_Add (Task_Count, 1);

      Put_Line ("Task_Number: "
                & Task_Number'Image);

```

(continues on next page)

(continued from previous page)

```
   end A_Task;

   A, B, C, D, E, F : A_Task;
begin
   null;
end Show_Atomic_Integers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
↪Integer_Arithmetic
MD5: 835093f90b9efe37b93ca84fe1ce3444
```

Runtime output

```
Task_Number:  0
Task_Number:  1
Task_Number:  2
Task_Number:  3
Task_Number:  4
Task_Number:  5
```

In this example, we call the Atomic_Fetch_And_Add function to update the Task_Count variable and, at the same time, initialize the Task_Number variable of the current task.

Relevant topics

- C.6.4 The Package System.Atomic_Operations.Integer_Arithmetic[87]

Atomic Operations using Modular Arithmetic

The generic System.Atomic_Operations.Modular_Arithmetic package is very similar to the System.Atomic_Operations.Integer_Arithmetic package. In fact, it provides the same operations: the procedures Atomic_Add and Atomic_Subtract, and the functions Atomic_Fetch_And_Add and Atomic_Fetch_And_Subtract. The only difference is that it is used for modular types instead of integer types.

Let's reuse the *previous code example* (page 171), but replace the atomic integer type by an atomic modular type:

Listing 40: atomic_modulars.ads

```
with System.Atomic_Operations.Modular_Arithmetic;

package Atomic_Modulars is

   type Atomic_Modular is mod 100
     with Atomic;

   package Atomic_Modular_Arithmetic is new
     System.Atomic_Operations.Modular_Arithmetic
       (Atomic_Modular);

end Atomic_Modulars;
```

[87] http://www.ada-auth.org/standards/22rm/html/RM-C-6-4.html

Listing 41: show_atomic_modulars.adb

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

with Atomic_Modulars;
use  Atomic_Modulars;
use  Atomic_Modulars.Atomic_Modular_Arithmetic;

procedure Show_Atomic_Modulars is
   Task_Count : aliased Atomic_Modular := 0;

   task type A_Task;

   task body A_Task is
      Task_Number : Atomic_Modular;
   begin
      Task_Number :=
        Atomic_Fetch_And_Add (Task_Count, 1);

      Put_Line ("Task_Number: "
                & Task_Number'Image);

   end A_Task;

   A, B, C, D, E, F : A_Task;
begin
   null;
end Show_Atomic_Modulars;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↪Modular_Arithmetic
MD5: 3a5a85febacd13f5e053cf00b19746ff
```

Runtime output

```
Task_Number:  0
Task_Number:  1
Task_Number:  2
Task_Number:  3
Task_Number:  4
Task_Number:  5
```

As we did in the previous example, we again call the `Atomic_Fetch_And_Add` function to update the `Task_Count` variable and, at the same time, initialize the `Task_Number` variable of the current task. The only difference is that we use a modular type (`Atomic_Modular`).

Relevant topics

- C.6.5 The Package System.Atomic_Operations.Modular_Arithmetic[88]

[88] http://www.ada-auth.org/standards/22rm/html/RM-C-6-5.html

CHAPTER FOUR

RECORDS

4.1 Default Initialization

As mentioned in the Introduction to Ada[89] course, record components can have default initial values. Also, we've seen that other kinds of types can have *default values* (page 69).

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

```
package Show_Default_Initialization is

   function Init return Integer is
     (42);

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

end Show_Default_Initialization;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Simple_
↪Example
MD5: 6d06be7f087513b669ba5481d6ee5004
```

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[90]

4.1.1 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:

[89] https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-default-values
[90] http://www.ada-auth.org/standards/22rm/html/RM-3-8.html

Listing 2: show_default_initialization_dependency.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.No_
↪Dependency
MD5: ca23cbd7e4a54d0b9c6974aed0ee77c8
```

Build output

```
show_default_initialization_dependency.ads:8:25: error: component "Rec.A" cannot␣
↪be used before end of record declaration
gprbuild: *** compilation phase failed
```

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.)

4.1.2 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

```
package Simple_Recs is

   function Init (S : 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

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

package body Simple_Recs is

   function Init (S : String;
                  I : Integer)
                  return Integer is
```

(continues on next page)

(continued from previous page)

```
   begin
      Put_Line (S & ": " & I'Image);
      return I;
   end Init;

end Simple_Recs;
```

Listing 5: show_initialization_order.adb

```
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.
↪Initialization_Order
MD5: e3ab92ea9b2a99815cea8c2ea11cbbfb
```

Runtime output

```
A:  1
B:  2
```

When running this code example, you might see this:

```
A: 1
B: 2
```

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

```
B: 2
A: 1
```

Therefore, we must write the default expression of each individual record components in such a way that the resulting initialization value is always correct, independently of the order that those expressions are evaluated.

4.1.3 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:

Listing 6: show_initialization_order.adb

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

```

(continues on next page)

(continued from previous page)

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.
 ↪Initialization_Order
MD5: 126e3edfe4cb8033f40b939ff9922958
```

Runtime output

```
Some processing first...
Now, let's declare an object of the record type Rec...
A:  1
B:  2
An object of Rec type has just been created.
```

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[91]

4.1.4 Defaults and object declaration

Note

This subsection was originally written by Robert A. Duff and published as Gem #12: Limited Types in Ada 2005[92].

Consider the following type declaration:

Listing 7: type_defaults.ads

```
package Type_Defaults is
   type Color_Enum is (Red, Blue, Green);

```

(continues on next page)

[91] http://www.ada-auth.org/standards/22aarm/html/AA-3-8.html
[92] https://www.adacore.com/gems/ada-gem-12

(continued from previous page)

```
   type T is private;
private
   type T is
      record
         Color     : Color_Enum := Red;
         Is_Gnarly : Boolean := False;
         Count     : Natural;
      end record;

   procedure Do_Something;
end Type_Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_
↪Init
MD5: 218154278081f89595534bc02e34539b
```

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

Listing 8: type_defaults.adb

```
package body Type_Defaults is

   Object_100 : constant T :=
                  (Color     => <>,
                   Is_Gnarly => <>,
                   Count     => 100);

   procedure Do_Something is null;

end Type_Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_
↪Init
MD5: e64f8881ee74b90dd6058ca8961aae31
```

> **Historically**
>
> Prior to Ada 2005, the following style was common:
>
> Listing 9: type_defaults.adb
>
> ```
> package body Type_Defaults is
>
>    Object_100 : constant T :=
>                   (Color     => Red,
>                    Is_Gnarly => False,
>                    Count     => 100);
>
>    procedure Do_Something is null;
>
> end Type_Defaults;
> ```
>
> **Code block metadata**
>
> ```
> Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_
> ↪Init
> MD5: c1ddfae75d7f0c691356027903a6d144
> ```

Here, we only wanted `Object_100` to be a default-initialized T, with **Count** equal to 100. It's a little bit annoying that we had to write the default values Red and **False** twice. What if we change our mind about Red, and forget to change it in all the relevant places? Since Ada 2005, the <> notation comes to the rescue, as we've just seen.

On the other hand, if we want to say, "make **Count** equal 100, but initialize all other components, including the ones we might add next week, to their defaults", we can do this:

Listing 10: type_defaults.adb

```
package body Type_Defaults is

   Object_100 : constant T := (Count  => 100,
                               others => <>);

   procedure Do_Something is null;

end Type_Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_
↪Init
MD5: 93f5d71ae80ff0ebad54f2569539f536
```

Note that if we add a component `Glorp : Integer;` to type T, then the **others** case leaves `Glorp` undefined just as this code would do:

Listing 11: type_defaults.adb

```
package body Type_Defaults is

   procedure Do_Something is
      Object_100 : T;
   begin
      Object_100.Count := 100;
   end Do_Something;

end Type_Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_
↪Init
MD5: 6d328318e2695516794df33466fa5283
```

Therefore, you should be careful and think twice before using **others**.

4.1.5 Advanced Usages

In addition to expressions such as subprogram calls, we can use *per-object expressions* (page 247) for the default value of a record component. (We discuss this topic later on in more details.)

For example:

Listing 12: rec_per_object_expressions.ads

```
package Rec_Per_Object_Expressions is

```

(continues on next page)

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Per_Object_
↪Expressions
MD5: 92591ea482db2b009b8eeafe633ca6cd
```

In this example, component V is initialized by default with the per-object expression `D - 1`, where D refers to the discriminant D.

4.2 Mutually dependent types

In this section, we discuss how to use *incomplete types* (page 41) to declare mutually dependent types. Let's start with this example:

Listing 13: mutually_dependent.ads

```
package Mutually_Dependent is

   type T1 is record
      B : T2;
   end record;

   type T2 is record
      A : T1;
   end record;

end Mutually_Dependent;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Mutually_Dependent_Types.Mutually_
↪Dependent
MD5: ffa8d6ab83a1172dcbae0978952dacb2
```

Build output

```
mutually_dependent.ads:4:11: error: "T2" is undefined
gprbuild: *** compilation phase failed
```

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:

Listing 14: 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 T2 is record
      A : T1;
   end record;

end Mutually_Dependent;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Mutually_Dependent_Types.Mutually_
↪Dependent
MD5: 1ae10638624a97fa18b9d8f96bfa74ed
```

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:

Listing 15: mutually_dependent.ads

```ada
package Mutually_Dependent is

   type T1;
   type T1_Access is access T1;

   type T2;
   type T2_Access is access T2;

   type T1 is record
      B : T2_Access;
   end record;

   type T2 is record
      A : T1_Access;
   end record;

end Mutually_Dependent;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Mutually_Dependent_Types.Mutually_
↪Dependent
MD5: 9a9899cd0dd2525bd27d67d6629a0071
```

Later on, we'll see that these code examples can be written using *anonymous access types* (page 734).

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations[93]

4.3 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

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Null_Record
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Null_Record
MD5: 1da1746ce5b0a237276272d2b620e282
```

Although a null record doesn't have components, we can still specify subprograms for it. For example, we could specify an addition operation for it:

Listing 18: null_recs.ads

```
package Null_Recs is

   type Null_Record is null record;

   function "+" (A, B : Null_Record)
                 return Null_Record;

end Null_Recs;
```

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

Listing 19: null_recs.adb

```
package body Null_Recs is

   function "+" (A, B : Null_Record)
                 return Null_Record
   is
      pragma Unreferenced (A, B);
   begin
      return (null record);
   end "+";

end Null_Recs;
```

Listing 20: show_null_rec.adb

```
with Null_Recs; use Null_Recs;

procedure Show_Null_Rec is
   A, B : Null_Record;
begin
   B := A + A;
   A := A + B;
end Show_Null_Rec;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Null_Record
MD5: 3a1c2fbae75541dfb0b2ff4c14d22039
```

In the Ada Reference Manual

- 4.3.1 Record Aggregates[94]

4.3.1 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;

```

(continues on next page)

94 http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html

(continued from previous page)

```
   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);

   Put_Line ("Resetting device...");
   Reset (A);
end Show_Device;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Device
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 500).

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.

4.3.2 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:

Listing 23: devices.ads

```
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 24: devices.adb

```
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
```

(continues on next page)

(continued from previous page)

```
      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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Device
MD5: 1a21b41f3847f6c132ccbc9696ab7689
```

Runtime output

```
Creating device...
Deactivating device...
Resetting device...
Activating device...
Processing on device...
```

As we changed the specification of the Devices package to not use null procedures, we now need a corresponding package body for it. In this package body, we implement the operations on the Device type, which actually just display a user message indicating which operation is being called.

Let's focus on this updated version of the Show_Device procedure. Now that we've removed all those calls to Put_Line from this procedure and just have the calls to operations associated with the Device type, it becomes more apparent that, even though Device is just a null record, we can design an application with a sequence of various commands operating on it. Also, when we just read the source-code of the Show_Device procedure, there's no clear indication that the Device type doesn't actually hold any information.

4.3.3 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

```
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

```
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);

   Process (B);
end Show_Derived_Device;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Derived_Device
MD5: 757a3def24c8333a27b64943727d8d4e
```

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.

4.3.4 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

```
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

```
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 (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;

```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Device
MD5: 348ce0c110b47a6b6fd1c9fe73ef0558
```

Build output

```
devices.adb:11:25: warning: aspect Unreferenced specified 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.

4.3.5 Tagged null records

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

Listing 31: null_recs.ads

```
package Null_Recs is

   type Tagged_Null_Record is
     tagged null record;
```

(continues on next page)

(continued from previous page)

```

   type Abstract_Tagged_Null_Record is
     abstract tagged null record;

end Null_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Tagged_Null_Record
MD5: 918572d2c50911b84c80a9c601b75439
```

As we see in this example, a type can be **tagged**, or even **abstract tagged**. We discuss abstract types later on in the course.

As expected, in addition to deriving from tagged types, we can also extend them. For example:

Listing 32: devices.ads

```
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);

   type Derived_Device is new Device with record
      Active : Boolean;
   end record;

   function Create (Active : Boolean)
                    return Derived_Device is
     (Active => Active);

end Devices;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Extended_Device
MD5: 15e06a5115cbcb131477b5224a6594db
```

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.

4.4 Record discriminants

We introduced the topic of record discriminants in the Introduction to Ada course[95]. Also, in a previous chapter, we mentioned that record types with unconstrained discriminants without defaults are *indefinite types* (page 35).

In this section, we discuss a couple of details about record discriminants that we haven't covered yet. Although the discussion will be restricted to record discriminants, keep in mind that tasks and protected types can also have discriminants. We'll focus on discriminants for tasks and protected types in separate chapters.

In addition, discriminants can be used to write *per-object expressions* (page 244). We discuss this topic later in this chapter.

In the Ada Reference Manual

- 3.7 Discriminants[96]

4.4.1 Known and unknown discriminant parts

When it comes to discriminants, a type declaration falls into one of the following three categories: it has either no discriminants at all, known discriminants or unknown discriminants.

In order to have no discriminants, a type simply doesn't have a discriminant part in its declaration. For example:

Listing 33: show_discriminants.ads

```
package Show_Discriminants is

   type T_No_Discr is private;
   --            ^^^
   --   no discriminant part

private

   type T_No_Discr is null record;

end Show_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.No_Discriminant_Part
MD5: f6701bd9c46b265753a258a6f99a5c7a
```

By using parentheses after the type name, we're defining a discriminant part. In this case, the type can either have unknown or known discriminants. For example:

Listing 34: show_discriminants.ads

```
package Show_Discriminants is

   type T_Unknown_Discr (<>) is
   --                    ^^
   --   Unknown discriminant
     private;
```

(continues on next page)

[95] https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_records.html#intro-ada-record-discriminants

[96] http://www.ada-auth.org/standards/12rm/html/RM-3-7.html

(continued from previous page)

```

   type T_Known_Discr (D : Integer) is
   --                  ^^^^^^^^^^^
   --   Known discriminant
     private;

private

   type T_Unknown_Discr is
     null record;

   type T_Known_Discr (D : Integer) is
     null record;

end Show_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminant_Parts
MD5: 486edc81b72473e022bb9e56ebaca559
```

An unknown discriminant part is represented by (<>) in the partial view — this is basically the so-called *box notation* <> (also known as *box compound delimiter*) in parentheses. We discuss unknown discriminant parts and their peculiarities *later on in this chapter* (page 221). In this section, we mainly focus on known discriminants.

We've already seen examples of known discriminants in previous chapters. In simple terms, known discriminants are composed by one or more discriminant specifications, which are similar to subprogram parameters, but without parameter modes. In fact, we can think of discriminants as parameters for a type T, but with the goal of defining specific characteristics or constraints when declaring objects of type T.

4.4.2 Discriminant as constant property

We can think of discriminants as constant properties of a type. In fact, if you want to specify a record component C that shouldn't change, declaring it constant isn't allowed in Ada:

Listing 35: constant_properties.ads

```
package Constant_Properties is

   type Rec is record
      C : constant Integer;
      --  ^^^^^^^^
      --  ERROR: record components
      --         cannot be constant.
      V :          Integer;
   end record;

end Constant_Properties;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Constant_Properties
MD5: ba189f437348c5892847d067b0bc2e78
```

Build output

```
constant_properties.ads:4:11: error: constant component not permitted
gprbuild: *** compilation phase failed
```

A simple solution is to use a record discriminant:

Listing 36: constant_properties.ads

```
package Constant_Properties is

   type Rec (C : Integer) is
   record
      V :          Integer;
   end record;

end Constant_Properties;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Constant_Properties
MD5: b638c2fd78761def2b60e9ae7dceb765
```

A record discriminant can be accessed as a normal component, but it is read-only, so we cannot change it:

Listing 37: show_constant_property.adb

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

with Constant_Properties;
use  Constant_Properties;

procedure Show_Constant_Property is
   R : Rec (10);
begin
   Put_Line ("R.C = "
             & R.C'Image);

   R.C := R.C + 1;
   --  ERROR: cannot change
   --         record discriminant
end Show_Constant_Property;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Constant_Properties
MD5: 82cde0032f2cb022e690f1175216fd77
```

Build output

```
show_constant_property.adb:12:05: error: assignment to discriminant not allowed
gprbuild: *** compilation phase failed
```

In this code example, the compilation fails because we cannot change the C discriminant. In this sense, C is a basically a constant component of the R object.

4.4.3 Private types

As we've seen in previous chapters, private types can have discriminants. For example:

Listing 38: private_with_discriminants.ads

```
package Private_With_Discriminants is

   type T (L : Positive) is private;

```

(continues on next page)

(continued from previous page)

```
private

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

   type T (L : Positive) is
   record
      Arr : Integer_Array (1 .. L);
   end record;

end Private_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Private_With_
↪Discriminants
MD5: 8f63443479e31a187a038381d9a32831
```

Here, discriminant L is used to specify the constraints of the array component Arr. Note that the same discriminant part must appear in both *the partial and the full view* (page 43) of type T.

4.4.4 Object declaration

As we've already seen, we declare objects of a type T with a discriminant D by specifying the actual value of discriminant D. This is called a *discriminant constraint* (page 215). For example:

Listing 39: recs.ads

```
package Recs is

   type T (L : Positive;
           M : Positive) is
     null record;

end Recs;
```

Listing 40: show_object_declaration.adb

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

with Recs;        use Recs;

procedure Show_Object_Declaration is
   A : T (L => 5, M => 6);
   B : T (7, 8);
begin
   Put_Line ("A.L = "
             & A.L'Image);
   Put_Line ("A.M = "
             & A.M'Image);
   Put_Line ("B.L = "
             & B.L'Image);
   Put_Line ("B.M = "
             & B.M'Image);
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Objects_
↪Discriminants
MD5: 9daae29be9d0f99980ca152a3aca7363
```

Runtime output

```
A.L =  5
A.M =  6
B.L =  7
B.M =  8
```

As we can see in the declaration of objects A and B, for the discriminant values, we can use a positional ((7, 8)) or named association ((L => 5, M => 6)).

Object size

Discriminants can have an impact on the object size because we can set the discriminant to constraint a component of an *indefinite subtype* (page 35). For example:

Listing 41: recs.ads

```ada
package Recs is

   type Null_Rec (L : Positive;
                  M : Positive) is
     private;

   type Rec_Array (L : Positive) is
     private;

private

   type Null_Rec (L : Positive;
                  M : Positive) is
     null record;

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

   type Rec_Array (L : Positive) is
   record
      Arr : Integer_Array (1 .. L);
   end record;

end Recs;
```

Listing 42: show_object_sizes.adb

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

with Recs;        use Recs;

procedure Show_Object_Sizes is
   Null_Rec_A  : Null_Rec (1, 2);
   Null_Rec_B  : Null_Rec (5, 6);
   Rec_Array_A : Rec_Array (10);
   Rec_Array_B : Rec_Array (20);
begin
   Put_Line ("Null_Rec_A'Size = "
             & Null_Rec_A'Size'Image);
   Put_Line ("Null_Rec_B'Size = "
```

(continues on next page)

(continued from previous page)

```
             & Null_Rec_B'Size'Image);
   Put_Line ("Rec_Array_A'Size = "
             & Rec_Array_A'Size'Image);
   Put_Line ("Rec_Array_B'Size = "
             & Rec_Array_B'Size'Image);
end Show_Object_Sizes;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Objects_
↪Discriminants_Size
MD5: 0abbc12286aff9fe428ea585564cf6d4
```

Build output

```
show_object_sizes.adb:8:04: warning: variable "Rec_Array_A" is read but never␣
↪assigned [-gnatwv]
show_object_sizes.adb:9:04: warning: variable "Rec_Array_B" is read but never␣
↪assigned [-gnatwv]
```

Runtime output

```
Null_Rec_A'Size =  64
Null_Rec_B'Size =  64
Rec_Array_A'Size =  352
Rec_Array_B'Size =  672
```

In this example, `Null_Rec_A` and `Null_Rec_B` have the same size because the type is a null record. However, `Rec_Array_A` and `Rec_Array_B` have different sizes because we're setting the `L` discriminant — which we use to constraint the `Arr` array component of the `Rec_Array` type — to 10 and 20, respectively.

4.4.5 Object assignments

As we've just seen, when we set the values for the discriminants of a type in the object declaration, we're constraining the objects. Those constraints are checked at runtime by the *discriminant check* (page 515). If the discriminants don't match, the `Constraint_Error` exception is raised.

Let's see an example:

Listing 43: recs.ads

```
package Recs is

   type T (L : Positive;
           M : Positive) is
     null record;

end Recs;
```

Listing 44: show_object_assignments.adb

```
with Recs;         use Recs;

procedure Show_Object_Assignments is
   A1, A2 : T (5, 6);
   B      : T (7, 8);
begin
   A1 := A2;    --  OK
```

(continues on next page)

(continued from previous page)

```
   B  := A1;    --  ERROR!
end Show_Object_Assignments;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Object_Assignments
MD5: 199247f1c0575c6845d85fd1911e1cf2
```

Build output

```
show_object_assignments.adb:8:10: warning: incorrect value for discriminant "L"
 ↪[enabled by default]
show_object_assignments.adb:8:10: warning: Constraint_Error will be raised at run
 ↪time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_object_assignments.adb:8 discriminant check failed
```

In this example, the A1 := A2 assignment is accepted because both A1 and A2 have the same constraints ((5, 6)). However, the B := A1 assignment is not accepted because the discriminant check fails at runtime.

Note that the discriminant check is not performed when we use *mutable subtypes* (page 203) — we discuss this specific kind of subtypes later on.

4.4.6 Discriminant type

In a discriminant specification, the type of the discriminant can only be a discrete subtype or an *access type* (page 603). Other kinds of types — e.g. composite types such as record types — are illegal for discriminants. However, we can always use them indirectly by using access types. (We'll see an example later.)

In addition to that, we can also use a different kind of access types, namely *anonymous access-to-object subtypes* (page 715). This specific kind of discriminant is called *access discriminant* (page 725). We discuss this topic in more details in another chapter.

Let's see a code example:

Listing 45: recs.ads

```
package Recs is

   type Usage_Mode is (Off,
                       Simple_Usage,
                       Advanced_Usage);

   type Priv_Info is private;

   type Priv_Info_Access is access Priv_Info;

   type Proc_Access is
     access procedure (P : in out Priv_Info);

   type Priv_Rec (Last  : Positive;
                  Usage : Usage_Mode;
                  Info  : Priv_Info_Access;
                  Proc  : Proc_Access) is
     private;

```

(continues on next page)

(continued from previous page)

```
private

   type Priv_Info is record
     A : Positive;
     B : Positive;
   end record;

   type Priv_Rec (Last  : Positive;
                  Usage : Usage_Mode;
                  Info  : Priv_Info_Access;
                  Proc  : Proc_Access) is
     null record;

end Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminants_
↪Subtype
MD5: 4ddbc703d8ffcd6dc31e3715df62931a
```

In this example, we're declaring the `Priv_Rec` type with the following discriminants:

- The `Last` discriminant of the scalar (i.e. discrete) type **`Positive`**;
- The `Usage` discriminant of the enumeration (i.e. discrete) type `Usage_Mode`;
- The `Info` discriminant of the *access-to-object type* (page 593) `Priv_Info_Access`;
  - We discuss *access-to-object types as discriminant type* (page 603) in another chapter.
- The `Proc` discriminant of the *access-to-subprogram type* (page 677) `Proc_Access`;
  - We discuss *access-to-subprogram types as discriminant type* (page 683) in another chapter.

As indicated previously, it's illegal to use a private type or a record type as the type of a discriminant. For example:

Listing 46: recs.ads

```
package Recs is

   type Priv_Info is private;

   type Priv_Rec (Info : Priv_Info) is
     private;
   --             ^^^^^^^^^^^^^^^^
   --  ERROR: cannot use private type
   --         in discriminant.

private

   type Priv_Info is record
     A : Positive;
     B : Positive;
   end record;

   type Priv_Rec (Info  : Priv_Info) is
     null record;

end Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminants_
↪Subtype_Error
MD5: 17f36f0e09cb069d8215b38adbb46541
```

Build output

```
recs.ads:5:26: error: discriminants must have a discrete or access type
gprbuild: *** compilation phase failed
```

We cannot use the `Priv_Info` directly as a discriminant type because it's a private type. However, as we've just seen in the previous code example, we use it indirectly by using an access type to this private type (see `Priv_Info_Access` in the code example).

Indefinite subtypes as discriminants

As we already implied, we cannot use indefinite subtypes as discriminants. For example, the following code won't compile:

Listing 47: unconstrained_types.ads

```ada
package Unconstrained_Types is

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

   type Simple_Record (Arr : Integer_Array) is
   --                      ^^^^^^^^^^^^^^^^^^^
   --  ERROR: cannot use indefinite type
   --         in discriminant.
   record
      L : Natural := Arr'Length;
   end record;

end Unconstrained_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↪Indefinite_Types_Error
MD5: f373b401ef1b179fef15cce0d2077286
```

Build output

```
unconstrained_types.ads:6:30: error: discriminants must have a discrete or access␣
↪type
gprbuild: *** compilation phase failed
```

`Integer_Array` is a correct type declaration — although the type itself is indefinite after the declaration. However, we cannot use it as the discriminant in the declaration of `Simple_Record`. We could, however, have a correct declaration by using discriminants as access values:

Listing 48: unconstrained_types.ads

```ada
package Unconstrained_Types is

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

```

(continues on next page)

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↪Indefinite_Types_Error
MD5: dc8193e3684b172e8503e1c5427cf93d
```

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 603).

4.4.7 Default values

We can specify default values for discriminants. Note, however, that we must either specify default values for **all** discriminants of the discriminant part or for none of them. This contrasts with default values for subprogram parameters, where we can *specify default values for just a subset of all parameters of a specific subprogram* (page 474).

As expected, we can override the default values by specifying the values of each discriminant when declaring an object. Let's see a simple example:

Listing 49: recs.ads

```
package Recs is

   type T (L : Positive := 1;
           M : Positive := 2) is
     private;

private

   type T (L : Positive := 1;
           M : Positive := 2) is
     null record;

end Recs;
```

Listing 50: show_object_declaration.adb

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

with Recs;        use Recs;

procedure Show_Object_Declaration is
   A : T;
   B : T (7, 8);
begin
   Put_Line ("A.L = "
             & A.L'Image);
   Put_Line ("A.M = "
```

(continues on next page)

(continued from previous page)

```
               & A.M'Image);
   Put_Line ("B.L = "
               & B.L'Image);
   Put_Line ("B.M = "
               & B.M'Image);
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminant_
↪Default_Value
MD5: 33385c4ba4ed9fc90c55990bde0b70cb
```

Runtime output

```
A.L =  1
A.M =  2
B.L =  7
B.M =  8
```

In this example, object A makes use of the default values for the discriminants of type T, so it has the discriminants (`L => 1, M => 2`). In the case of object B, we're specifying the values (`L => 7, M => 8`), which are used instead of the default values.

Note that we cannot set default values for nonlimited tagged types. The same applies to generic formal types. For example:

Listing 51: recs.ads

```
package Recs is

   type TT (L : Positive := 1;
            M : Positive := 2) is
   --       ^^^^^^^^^^^^^^^^^
   --  ERROR: cannot assign default
   --         in discriminant of
   --         nonlimited tagged type.
     tagged private;

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     tagged limited private;

private

   type TT (L : Positive := 1;
            M : Positive := 2) is
     tagged null record;

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     tagged limited null record;

end Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminant_
↪Default_Value_Tagged_TYpe
MD5: 94b78f032efe49de0b8198083a25d79b
```

Build output

```
recs.ads:3:29: error: discriminants of nonlimited tagged type cannot have defaults
recs.ads:4:29: error: discriminants of nonlimited tagged type cannot have defaults
gprbuild: *** compilation phase failed
```

As we can see, compilation fails because of the default values for the discriminants of the nonlimited tagged type TT. In the case of the limited tagged type LTT, the default values for the discriminants are legal.

Mutable subtypes

An unconstrained discriminated subtype with defaults is called a mutable subtype, and a variable of such a subtype is called a mutable variable because the discriminants of such a variable can be changed. An important feature of mutable subtypes is that it allows for changing the discriminants of an object via assignments — in this case, no *discriminant check* (page 515) is performed.

Let's see an example:

Listing 52: mutability.ads

```ada
package Mutability is

   type T_Non_Mutable
     (L : Positive;
      M : Positive) is
     null record;

   type T_Mutable
     (L : Positive := 1;
      M : Positive := 2) is
     null record;

end Mutability;
```

Listing 53: show_mutable_subtype_assignment.adb

```ada
with Mutability; use Mutability;

procedure Show_Mutable_Subtype_Assignment is
   NM_1 : T_Non_Mutable (5, 6);
   NM_2 : T_Non_Mutable (7, 8);

   M_1  : T_Mutable (7, 8);
   M_2  : T_Mutable;
begin
   NM_2 := NM_1;  --  ERROR!
   M_2  := M_1;   --  OK
end Show_Mutable_Subtype_Assignment;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Mutable_Subtype
MD5: ace4470544bdc6efb0dca7039ca33cbc
```

Build output

```
show_mutable_subtype_assignment.adb:10:12: warning: incorrect value for
 discriminant "L" [enabled by default]
show_mutable_subtype_assignment.adb:10:12: warning: Constraint_Error will be
 raised at run time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_mutable_subtype_assignment.adb:10 discriminant␣
↪check failed
```

In this example, the NM_2 := NM_1 assignment fails because both objects are of a non-mutable subtype with different discriminants, so that the discriminant check fails at runtime. However, the M_2 := M_1 assignment is OK because both objects are mutable variables. In this case, this assignment changes the discriminants of M_2 from (L => 1, M => 2) to (L => 7, M => 8).

Note that assignments of mutable variables may not always work at runtime. For example, if a discriminant of a mutable subtype is used to constraint a component of indefinite subtype, we might see the corresponding checks fail at runtime. For example:

Listing 54: mutability.ads

```ada
package Mutability is

   type T_Mutable_Array (L : Positive := 10) is
     private;

private

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

   type T_Mutable_Array (L : Positive := 10) is
   record
      Arr : Integer_Array (1 .. L);
   end record;

end Mutability;
```

Listing 55: show_mutable_subtype_error.adb

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

with Mutability;  use Mutability;

procedure Show_Mutable_Subtype_Error is
   A : T_Mutable_Array (10);
   B : T_Mutable_Array (20);
begin
   Put_Line ("A'Size = "
             & A'Size'Image);
   Put_Line ("B'Size = "
             & B'Size'Image);

   A := B;  --  ERROR!
end Show_Mutable_Subtype_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Mutable_Subtype_
↪Error
MD5: 95bc55e7a01c1160dd2f7139778d2d16
```

Build output

```
show_mutable_subtype_error.adb:7:04: warning: variable "B" is read but never␣
↪assigned [-gnatwv]
```

(continues on next page)

(continued from previous page)

```
show_mutable_subtype_error.adb:14:09: warning: incorrect value for discriminant "L
↪" [enabled by default]
show_mutable_subtype_error.adb:14:09: warning: Constraint_Error will be raised at␣
↪run time [enabled by default]
mutability.ads:11:09: warning: creation of "T_Mutable_Array" object may raise␣
↪Storage_Error [enabled by default]
```

Runtime output

```
A'Size =  352
B'Size =  672

raised CONSTRAINT_ERROR : show_mutable_subtype_error.adb:14 discriminant check␣
↪failed
```

In this case, the assignment `A := B` raises the `Constraint_Error` exception at runtime. Here, the `Arr` component of each object has a different range: `1 .. 10` for object A and `1 .. 20` for object B. To prevent this situation, we should declare `T_Mutable_Array` as a limited type, so that assignments are not permitted.

4.4.8 Derived types and subtypes

As expected, we may derive types with discriminants or declare subtypes of it. However, there are a couple of details associated with this, which we discuss now.

Subtypes

When declaring a subtype of a type with discriminants, we have the choice to specify the value of the discriminants for the parent type, or specify no discriminants at all:

Listing 56: subtypes_with_discriminants.ads

```ada
package Subtypes_With_Discriminants is

   type T
     (L : Positive;
      M : Positive) is
     null record;

   subtype Sub_T is T;
   --  Discriminants are not specified:
   --  taking the ones from T.

   subtype Sub_T_2 is T
    (L => 3, M => 4);
   --  Discriminants are specified:
   --  taking the ones from Sub_T_2

end Subtypes_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Subtypes
MD5: 6f02c295f295c81fd20d06f7c710994c
```

For the Sub_T subtype declaration in this example, we don't specify values for the parent type's discriminants. For Sub_T_2, in contrast, we set the discriminants to `(L => 3, M => 4)`.

When declaring objects of these subtypes, we need to take the constraints into account:

Listing 57: subtypes_with_discriminants.ads

```ada
package Subtypes_With_Discriminants is

   type T
     (L : Positive;
      M : Positive) is
     null record;

   subtype Sub_T is T;
   --  Discriminants are not specified:
   --  taking the ones from T.

   subtype Sub_T_2 is T
    (L => 3, M => 4);
   --  Discriminants are specified:
   --  taking the ones from Sub_T_2

end Subtypes_With_Discriminants;
```

Listing 58: show_subtypes_with_discriminants.adb

```ada
with Subtypes_With_Discriminants;
use  Subtypes_With_Discriminants;

procedure Show_Subtypes_With_Discriminants is
   A1 : T (1, 2);
   A2 : T (3, 4);
   B1 : Sub_T (1, 2);
   B2 : Sub_T (3, 4);
   C2 : Sub_T_2;

   --  C1 : Sub_T_2 (1, 2);
   --                ^^^^
   --  ERROR: discriminants already
   --         constrained
begin
   B1 := A1;
   --  OK: discriminants match

   B2 := A1;
   --  CONSTRAINT_ERROR!

   B2 := A2;
   --  OK: discriminants match

   C2 := A1;
   --  CONSTRAINT_ERROR!

   C2 := A2;
   --  OK: discriminants match
end Show_Subtypes_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Subtypes
MD5: 9a8516d70e7a53ae332e5c5b6df7f04e
```

Build output

```
show_subtypes_with_discriminants.adb:19:10: warning: incorrect value for␣
```

(continues on next page)

(continued from previous page)

```
↪discriminant "L" [enabled by default]
show_subtypes_with_discriminants.adb:19:10: warning: Constraint_Error will be␣
↪raised at run time [enabled by default]
show_subtypes_with_discriminants.adb:25:10: warning: incorrect value for␣
↪discriminant "L" [enabled by default]
show_subtypes_with_discriminants.adb:25:10: warning: Constraint_Error will be␣
↪raised at run time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_subtypes_with_discriminants.adb:19 discriminant␣
↪check failed
```

For objects of Sub_T subtype, we *have to* specify the value of each discriminant. On the other hand, for objects of Sub_T_2 type, we *cannot* specify the constraints because they have already been defined in the subtype's declaration — in this case, they're always set to (3, 4).

When assigning objects of different subtypes, the discriminant check will be performed — as we *mentioned before* (page 197). In this example, the assignments B2 := A1 and C2 := A1 fail because the objects have different constraints.

Derived types

The behavior for derived types is very similar to the one we've just described for subtypes. For example:

Listing 59: derived_with_discriminants.ads

```ada
package Derived_With_Discriminants is

   type T
     (L : Positive;
      M : Positive) is
     null record;

   type T_Derived is new T;
   --  Discriminants are not specified:
   --  taking the ones from T.

   type T_Derived_2 is new T
     (L => 3, M => 4);
   --  Discriminants are specified:
   --  taking the ones from T_Derived_2

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types
MD5: 1e88f787bd9b568e43fc423c121f24f7
```

For the T_Derived type, we reuse the discriminants of the parent type T. For the T_Derived_2 type, we specify a value for each discriminant of T.

As you probably notice, this code looks very similar to the code using subtypes. The main difference between using subtypes and derived types is that, as expected, we have to perform a *type conversion* (page 49) in the assignments:

Listing 60: show_derived_with_discriminants.adb

```ada
with Derived_With_Discriminants;
use  Derived_With_Discriminants;

procedure Show_Derived_With_Discriminants is
   A1 : T (1, 2);
   A2 : T (3, 4);
   B1 : T_Derived (1, 2);
   B2 : T_Derived (3, 4);
   C2 : T_Derived_2;

   --  C1 : Sub_T_2 (1, 2);
   --                ^^^^
   --  ERROR: discriminants already
   --         constrained
begin
   B1 := T_Derived (A1);
   --  OK: discriminants match

   B2 := T_Derived (A1);
   --  ERROR!

   C2 := T_Derived_2 (A1);
   --  CONSTRAINT_ERROR!

   C2 := T_Derived_2 (A2);
   --  OK: discriminants match
end Show_Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types
MD5: 1b32807fcd3b343fbf8ab0d0287ca5bb
```

Build output

```
show_derived_with_discriminants.adb:22:23: warning: incorrect value for discriminant "L" [enabled by default]
show_derived_with_discriminants.adb:22:23: warning: Constraint_Error will be raised at run time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_derived_with_discriminants.adb:19 discriminant check failed
```

Once again, a discriminant check is performed when assigning objects to ensure that the type discriminants match. In this code example, the assignments `B2 := A1` and `C2 := A1` fail because the objects have different constraints.

Derived types with renamed discriminants

We could rewrite a type declaration such as `type T_Derived is new T` by explicitly declaring the discriminants. We can do that for the previous code example:

Listing 61: derived_with_discriminants.ads

```ada
package Derived_With_Discriminants is

```

(continues on next page)

(continued from previous page)

```
   type T
     (L : Positive;
      M : Positive) is
     null record;

   --  The declaration:
   --
   --     type T_Derived is new T;
   --
   --  is the same as:
   --
   type T_Derived
     (L : Positive;
      M : Positive) is
     new T (L => L, M => M);

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_Same_
↪Discriminants
MD5: 3ee4b3a70e8ab9ba2684c6a2c695f689
```

We may, however, rename the discriminants instead. For example, we could rename L and M to X and Y. For example:

Listing 62: derived_with_discriminants.ads

```
package Derived_With_Discriminants is

   type T
     (L : Positive;
      M : Positive) is
     null record;

   type T_Derived
     (X : Positive;
      Y : Positive) is
     new T (L => X, M => Y);

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Renamed_Discriminants
MD5: ec2954f538fa63b4d3c7c134527be35d
```

Of course, if we use named association when declaring objects, we have to use the correct discriminant names:

Listing 63: show_derived_with_discriminants.adb

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

with Derived_With_Discriminants;
use  Derived_With_Discriminants;

procedure Show_Derived_With_Discriminants is
   A : T (L => 1, M => 2);
```

(continues on next page)

(continued from previous page)

```
   B : T_Derived (X => 3, Y => 4);
   --              ^^^^^^^^^^^^^^
   --  Using correct discriminant names
begin
   Put_Line ("A.L = "
             & A.L'Image);
   Put_Line ("A.M = "
             & A.M'Image);
   Put_Line ("B.X = "
             & B.X'Image);
   Put_Line ("B.Y = "
             & B.Y'Image);
end Show_Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Renamed_Discriminants
MD5: cd3ca2d84c8b7d334b152ebab1955a5e
```

Runtime output

```
A.L =  1
A.M =  2
B.X =  3
B.Y =  4
```

In essence, the discriminants of both parent and derived types are the same: the only difference is that they are accessed by different names. This allows us to convert from a parent type to a derived type:

Listing 64: show_derived_with_discriminants.adb

```
with Derived_With_Discriminants;
use  Derived_With_Discriminants;

procedure Show_Derived_With_Discriminants is
   A : T (L => 1, M => 2);
   B : T_Derived (X => 1, Y => 2);
begin
   B := T_Derived (A);  --  OK
end Show_Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Renamed_Discriminants
MD5: d685f16bf3a9d64b4c1f182880455ad0
```

Here, even though objects A and B have discriminants with different names, the assignment `B := T_Derived (A)` is valid.

Derived types with more constrained discriminants

When deriving types with discriminants, we may use a more constrained type for the discriminants of derived type. For example, if the discriminant `D` of the parent type is of **`Integer`** type, the corresponding discriminant of the derived type may use a constrained subtype such as **`Natural`** or **`Positive`** — because both **`Natural`** and **`Positive`** are subtypes of type **`Integer`**. For example:

Listing 65: derived_with_discriminants.ads

```
package Derived_With_Discriminants is

   type T
     (L : Integer;
      M : Integer) is
     null record;

   type T_Derived_2
     (X : Natural;
      Y : Positive) is
     new T (L => X, M => Y);

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_More_
↪Constrained_Discriminants
MD5: 413f04f1f98dde2a6e0df3ee6955da7f
```

As expected, the constraints of each discriminant's type are taken into account when evaluating the value that is specified for each discriminant:

Listing 66: show_derived_with_discriminants.adb

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

with Derived_With_Discriminants;
use  Derived_With_Discriminants;

procedure Show_Derived_With_Discriminants is
   A : T (L => -1, M => -2);
   B : T_Derived_2 (X => 0, Y => 1);
begin
   Put_Line ("A.L = "
             & A.L'Image);
   Put_Line ("A.M = "
             & A.M'Image);
   Put_Line ("B.X = "
             & B.X'Image);
   Put_Line ("B.Y = "
             & B.Y'Image);
end Show_Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_More_
↪Constrained_Discriminants
MD5: 508bb7a6eb93005f8f1e5a937b55473c
```

Runtime output

```
A.L = -1
A.M = -2
B.X =  0
B.Y =  1
```

Here, we can use `(L => -1, M => -2)` in the declaration of object A because both discriminants are of **Integer** type. However, in the declaration of object B, we can only use values for the discriminants that are in the range of the **Natural** and **Positive** subtypes, respectively. (If you change the code to use negative values instead, a Constraint_Error exception is raised at runtime.)

Extending the discriminant part

As we've seen, we can rename discriminants or use more constrained subtypes for discriminants in derived types. We might also want to add a new discriminant to the derived type — in addition to the discriminants of the parent's type. However, this is considered a type extension, as the new discriminant is part of the type definition.

As an example, we may want to add the A discriminant of **Boolean** type to a derived type. For non-tagged types, such a declaration will trigger a compilation error as expected:

Listing 67: derived_with_discriminants.ads

```
package Derived_With_Discriminants is

   type T
     (L : Positive;
      M : Positive) is
     null record;

   type T_Derived
     (X : Positive;
      Y : Positive;
      A : Boolean) is
   --  ^^^^^^^^^^^
   --  ERROR: cannot extend type with new
   --         Boolean discriminant A
     new T (L => X, M => Y);

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Extension_Error
MD5: f9ba4ae1c344d63ed706005e30fe60c2
```

Build output

```
derived_with_discriminants.ads:11:07: error: new discriminants must constrain old␣
↪ones
gprbuild: *** compilation phase failed
```

To circumvent this issue, we could, of course, declare a component of T type instead of deriving from it:

Listing 68: derived_with_discriminants.ads

```
package Derived_With_Discriminants is

   type T
     (L : Positive;
      M : Positive) is
     null record;

   type T_2
     (X : Positive;
      Y : Positive;
      A : Boolean) is
   record
```

(continues on next page)

(continued from previous page)

```
      A_Comp : T (L => X, M => Y);
   end record;

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Extension_Error
MD5: 41e911cdc8486cd49931b2082586d8e7
```

In this case, A_Comp is a component of type T, and we're using the discriminant X and Y as the constraints of this component.

Naturally, using tagged types is another alternative:

Listing 69: derived_with_discriminants.ads

```
package Derived_With_Discriminants is

   type T
     (L : Positive;
      M : Positive) is
     tagged null record;

   type T_Derived_Extended
     (X : Positive;
      Y : Positive;
      A : Boolean) is  --  New discriminant
     new T (L => X, M => Y)
       with null record;

   type T_Derived_Extended_2
     (A : Boolean;      --  New discriminant
      X : Positive;
      Y : Positive) is
     new T (L => X, M => Y)
       with null record;

   type T_Derived_Extended_3
     (A : Boolean) is  --  New discriminant
     new T (L => 1, M => 2)
       with null record;

   type T_Derived_Extended_4
     (A : Boolean;      --  New discriminant
      X : Positive) is
     new T (L => X, M => X)
       with null record;

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Tagged_Types
MD5: b8124d132a4b5066826980c8cc43a7ad
```

In this code example, we're adding the A discriminant when declaring T_Derived_Extended. Because T is a tagged type, such a new discriminant is fine.

Note that the order of the discriminants can be rearranged: when deriving a new type, we don't need to specify the discriminants of the parent type before any new discriminants. In fact, in the declaration of T_Derived_Extended_2, the additional discriminant A is declared before the discriminants that match the parent type's discriminants.

In addition, we may even use literals to specify the constraints for the parent type — as we're doing in the declaration of T_Derived_Extended_3. Also, we can use the same discriminant from the derived type for the constraints of the parent type — in the declaration of T_Derived_Extended_4, we use the X discriminant for both L and M discriminants of type T.

Deriving with defaults

If the discriminants of the parent type have default values, those default values are inherited by the derived type. Alternatively, we can set different default values.

Let's see a code example:

Listing 70: derived_with_discriminants.ads

```
package Derived_With_Discriminants is

   type T
     (L : Positive := 1;
      M : Positive := 2) is
     null record;

   type T_Derived is new T;

   type T_Derived_2
     (L : Positive := 1;
      M : Positive := 3) is
     new T (L => L, M => M);

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Defaults
MD5: 4ffa513c0cd8b6812359a2fc4d8325d2
```

In this example, the derived type T_Derived has the same default values as the parent type T, namely (L => 1, M => 2). For the derived type T_Derived_2, we're changing the value of M to 3 and keeping the same value for L.

As we've seen before, instead of setting default values, we can set the constraints of the parent type in the declaration of the derived type:

Listing 71: derived_with_discriminants.ads

```
package Derived_With_Discriminants is

   type T
     (L : Positive := 1;
      M : Positive := 2) is
     null record;

   type T_Derived_Constrainted is new T
     (L => 1, M => 3);

end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Defaults_Constraints
MD5: d6053fc79a3e7010ec7b3ec73f51f4e5
```

In this case, we're constraining the discriminants of the parent type to `(L => 1, M => 3)`. Note that L has the same value as the default value set for the parent type T.

For further reading...

In other contexts (such as *record aggregates* (page 253), which we discuss in another chapter), we could use the so-called *box notation* (page 256) to specify that we want to use the default value. This, however, isn't possible with type discriminants:

Listing 72: derived_with_discriminants.ads

```ada
package Derived_With_Discriminants is

   type T
     (L : Positive := 1;
      M : Positive := 2) is
     null record;

   type T_Derived_Constraint is new T
     (L => <>, M => 3);
   --    ^^^^^^^
   --  ERROR: cannot use default values
   --         via box notation
end Derived_With_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↪Defaults_Constraints_Box_Notation
MD5: 18d755ec4de45164a47009ab25368452
```

Build output

```
derived_with_discriminants.ads:9:11: error: missing operand
gprbuild: *** compilation phase failed
```

Instead of using <>, we have to repeat the value explicitly.

4.5 Discriminant constraints and operations

In this section, we discuss some details about discriminant constraints and operations related to discriminants — more specifically, the `Constrained` attribute.

In the Ada Reference Manual

- 3.7.1 Discriminant Constraints[97]

4.5.1 Discriminant constraints

As we discussed before, when *declaring an object with a discriminant* (page 195), we have to specify the values of the all discriminants — unless, of course, those discriminants have a *default value* (page 201). The values we specify for the discriminants are called discriminant constraints.

[97] http://www.ada-auth.org/standards/12rm/html/RM-3-7-1.html

Let's revisit the code example we've seen earlier on:

Listing 73: recs.ads

```
package Recs is

   type T (L : Positive;
           M : Positive) is
      null record;

end Recs;
```

Listing 74: show_object_declaration.adb

```
with Recs;         use Recs;

procedure Show_Object_Declaration is
   A : T (L => 5, M => 6);
   B : T (7, 8);
   C : T (7, M => 8);
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_
↪Operations.Discriminant_Constraint
MD5: 9e37a1cde73f27b99fd2a9eb57f23c44
```

Here, `L => 5, M => 6` (for object A) are named constraints, while `7, 8` (for object B) are positional constraints.

It's possible to use both positional and named constraints, as we do for object C: `7, M => 8`. In this case, the positional associations must precede the named associations.

In the case of named constraints, we can use multiple selector names:

Listing 75: show_object_declaration.adb

```
with Recs;         use Recs;

procedure Show_Object_Declaration is
   A : T (L | M => 5);
   --      ^^^^^
   --  multiple selector names
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_
↪Operations.Discriminant_Constraint
MD5: b6fbe1d69bb520a7b6845536a1601978
```

This is only possible, however, if those named discriminants are all of the same type. (In this case, L and M are both of **Positive** subtype.)

In the Ada Reference Manual

- 3.7.1 Discriminant Constraints[98]

Discriminant constraint in subtypes

We can use discriminant constraints in the declaration of subtypes. For example:

Listing 76: show_object_declaration.adb

```
with Recs;        use Recs;

procedure Show_Object_Declaration is
   subtype T_5_6 is T (L => 5, M => 6);
   --                 ^^^^^^^^^^^^^^
   --  discriminant constraints for subtype

   A : T_5_6;
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_
↪Operations.Discriminant_Constraint
MD5: c6c4226073f4282d68d621519ca4d420
```

In this example, we use the named discriminant constraints `L => 5, M => 6` in the declaration of the subtype `T_5_6`.

4.5.2 Constrained Attribute

We can use the `Constrained` attribute to verify whether an object of discriminated type is constrained or not. Let's look at a simple example:

Listing 77: recs.ads

```
package Recs is

   type T (L : Positive := 1) is
     null record;

end Recs;
```

Listing 78: show_constrained_attribute.adb

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

with Recs;        use Recs;

procedure Show_Constrained_Attribute is
   Constr   : T (L => 5);
   --            ^^^^^^ constrained.
   Unconstr : T;
   --          ^ unconstrained;
   --            using defaults.
begin
   Put_Line ("Constr'Constrained:   "
             & Constr'Constrained'Image);
   Put_Line ("Unconstr'Constrained: "
             & Unconstr'Constrained'Image);
end Show_Constrained_Attribute;
```

Code block metadata

[98] http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_
↪Operations.Simple_Constrained_Attribute
MD5: 6a9a807f5af132a07949d2887fa5bfe5
```

Runtime output

```
Constr'Constrained:   TRUE
Unconstr'Constrained: FALSE
```

As the `Constrained` attribute indicates, the `Constr` object is constrained (by the `L => 5` discriminant constraint), while the `Unconstr` object is unconstrained. Note that, even though `Unconstr` is using the default value for `L` — which would correspond to the discriminant constraint `L => 1` — the object itself hasn't been constraint at its declaration.

Let's continue our discussion with a more complex example by reusing the `Unconstrained_Types` package that we declared in a *previous section* (page 35). In this version of the package, we're adding a `Reset` procedure for the discriminated record type `Simple_Record`:

Listing 79: unconstrained_types.ads

```ada
package Unconstrained_Types is

   type Simple_Record
     (Extended : Boolean := False) is
   record
      V : Integer;
      case Extended is
         when False =>
            null;
         when True  =>
            V_Float : Float;
      end case;
   end record;

   procedure Reset (R : in out Simple_Record);

end Unconstrained_Types;
```

Listing 80: 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
```

(continues on next page)

(continued from previous page)

```
         if R.Extended then
            R := Zero_Extended;
         else
            R := Zero_Not_Extended;
         end if;
      end if;
   end Reset;

end Unconstrained_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_
↪Operations.Constrained_Attribute
MD5: b56e6d71fd4f05e8490412d7fe40b923
```

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.

Note that the `Simple_Record` type has a *variant part* (page 231). We discuss this topic later on in this chapter.

Note as well that, in the initialization of the `Zero_Not_Extended` and `Zero_Extended` constants, we have to indicate the discriminant as a component of the aggregates (e.g.: `(Extended => False, V => 0)`). We discuss this topic in another chapter when we learn more about *aggregates and record discriminants* (page 262).

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**:

Listing 81: using_constrained_attribute.adb

```
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);
```

(continues on next page)

(continued from previous page)

```
      Put_Line ("----------------");
   end Show_Rs;
begin
   Show_Rs;

   Reset (R1);
   Reset (R2);
   Put_Line ("----------------");

   Show_Rs;
end Using_Constrained_Attribute;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_
↪Operations.Constrained_Attribute
MD5: f7517fcd3c68a784f55064f188d4e7bb
```

Runtime output

```
R1'Constrained => FALSE
R1.Extended => FALSE
--
R2'Constrained => TRUE
R2.Extended => FALSE
----------------
---- Reset: R'Constrained => FALSE
---- Reset: R'Constrained => TRUE
----------------
R1'Constrained => FALSE
R1.Extended => TRUE
--
R2'Constrained => TRUE
R2.Extended => FALSE
----------------
```

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 we 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[99]

4.6 Unknown discriminants

As we've seen *previously* (page 192), a type with discriminants can have known discriminants or unknown discriminants. In this section, we focus on unknown discriminants. Because the discriminants are unknown, this is an *indefinite type* (page 35). Let's start with a simple example:

Listing 82: unknown_discriminants.ads

```ada
package Unknown_Discriminants is

   type T_Unknown_Discr (<>) is
   --                   ^^^^
   --   Unknown discriminant part
     private;

private

   type T_Unknown_Discr is
     null record;

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Simple_
↪Example
MD5: 5f673c957132b1bca633c247f857e37b
```

Note that we can only use an unknown discriminant part in the *partial view* (page 43); we cannot use it in the full view of a type:

Listing 83: unknown_discriminants.ads

```ada
package Unknown_Discriminants is

   type T_Unknown_Discr (<>) is
     null record;

end Unknown_Discriminants;
```

Code block metadata

[99] http://www.ada-auth.org/standards/22rm/html/RM-3-7-2.html

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Wrong_Full_
↪View
MD5: dfce1471556af87b6a99314b1ee32446
```

Build output

```
unknown_discriminants.ads:3:25: error: full type declaration cannot have unknown␣
↪discriminants
gprbuild: *** compilation phase failed
```

To be more precise, an unknown discriminant part can only be used in the declaration of a private type, a private extension or an *incomplete type* (page 41). In addition, as we'll see in another chapter, it can also be used in the generic equivalents: generic private types, generic private extensions, generic incomplete types, and formal derived types.

For example:

Listing 84: unknown_discriminants.ads

```ada
package Unknown_Discriminants is

   --  Private type
   type Rec (<>) is
     private;

   --  Tagged private type
   type Tagged_Rec (<>) is
     tagged private;

   --  Incomplete type
   type T_Incomplete (<>);

   type T_Incomplete (<>) is
     private;

private

   type Rec is
     null record;

   type Tagged_Rec is
     tagged null record;

   type T_Incomplete is
     null record;

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Simple_
↪Example
MD5: e601ab326c43501e36e0d4656dc1629e
```

In this example, we have three forms of private types using an unknown discriminant part: an untagged private type (Rec), a tagged type (Tagged_Rec) and an incomplete type (T_Incomplete) that becomes an untagged private type.

In the Ada Reference Manual

- 3.7 Discriminants[100]

4.6.1 Object declaration

Now, let's talk about objects of types with unknown discriminants. Consider the Rec type below:

Listing 85: unknown_discriminants.ads

```
package Unknown_Discriminants is

   type Rec (<>) is private;

private

   type Rec is
   record
      I : Integer;
   end record;

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Object_
↪Declaration
MD5: 9f588870ec70ea30c795a6a0a602f589
```

We cannot declare objects of type Rec *directly*, as this type is *indefinite* (page 35):

Listing 86: show_object_declaration.adb

```
with Unknown_Discriminants;
use  Unknown_Discriminants;

procedure Show_Object_Declaration is
   A : Rec;
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Object_
↪Declaration
MD5: 5f30773fc17096943939468faf50338b
```

Build output

```
show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need␣
↪initialization)
gprbuild: *** compilation phase failed
```

Because the type is indefinite, it requires explicit initialization — we can do this by introducing a subprogram that initializes the type. In our code example, we can implement a simple Init function for this type:

Listing 87: unknown_discriminants.ads

```
package Unknown_Discriminants is

   type Rec (<>) is private;

```

(continues on next page)

100 http://www.ada-auth.org/standards/12rm/html/RM-3-7.html

(continued from previous page)

```
   function Init return Rec;

private

   type Rec is
   record
      I : Integer;
   end record;

   function Init return Rec is
     ((I => 0));

end Unknown_Discriminants;
```

Listing 88: show_constructor_function.adb

```
with Unknown_Discriminants;
use  Unknown_Discriminants;

procedure Show_Constructor_Function is
   R : Rec := Init;
begin
   null;
end Show_Constructor_Function;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Object_Declaration
MD5: 1cee0c4b883b3a0c25fae0a5111db2a8
```

In the Show_Constructor_Function procedure from this example, we call the Init function to initialize the R object in its declaration (of Rec type). Note that for this specific type, this is the only possible way to declare the R object. In fact, compilation fails if we write R : Rec;.

Using a private type with unknown discriminants is an important Ada idiom, as we gain extra control over its initialization. For example, if we have to ensure that certain components of the private record are initialized when an object is being declared, we can perform this initialization in the Init function — instead of just hoping that an initialization function is called for this object at some point. Also, if further information is needed to initialize an object, we can add parameters to the Init function, thereby forcing the user to provide this information.

For even more control over objects, we can use *limited types with unknown discriminants* (page 812).

4.6.2 Partial and full view

As we've just seen, if we declare a type with an unknown discriminant part, we can only use it in the partial view. In the full view. we cannot use an unknown discriminant part, but have to use either no discriminants or known discriminants. For example:

Listing 89: unknown_discriminants.ads

```
package Unknown_Discriminants is

   type Rec_No_Discr (<>) is private;

   type Rec_Known_Discr (<>) is private;
```

(continues on next page)

(continued from previous page)

```

private

   type Rec_No_Discr is null record;

   type Rec_Known_Discr
     (L : Positive) is null record;

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Partial_
↪Full_View
MD5: 3d37dcc9d1b12bf9a189cf515b168430
```

In this example, Rec_No_Discr has no discriminants in its full view, while Rec_Known_Discr has the discriminant L.

In addition, the full view can be an (unconstrained) array type as well:

Listing 90: unknown_discriminants.ads

```
package Unknown_Discriminants is

   type Arr (<>) is private;

private

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

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Partial_
↪Full_View
MD5: d1e0f60048c6ca6bcf863a8c0cf68314
```

Here, the full view of Arr is an array type.

In the Ada Reference Manual

- 3.7 Discriminants[101]

4.6.3 Derived types

As expected, we can derive from types with unknown discriminants. Consider the following package:

Listing 91: unknown_discriminants.ads

```
package Unknown_Discriminants is

   type Rec (<>) is private;
```

(continues on next page)

[101] http://www.ada-auth.org/standards/12rm/html/RM-3-7.html

(continued from previous page)

```

private

   type Rec is null record;

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 948e7c7ecd00915fa23a98cbaf2bbcbe
```

We can then declare the Derived_Rec type:

Listing 92: unknown_discriminants-children.ads

```
package Unknown_Discriminants.Children is

   type Derived_Rec is
     new Rec;

end Unknown_Discriminants.Children;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 1fa7e905c794d48bf6c76ff51e1abd8d
```

Note that Derived_Rec has unknown discriminants, even though we're not explicitly using an unknown discriminant part ((<>)) in its declaration. (In fact, we're not allowed to use an unknown discriminant part in this case.) Therefore, declaring objects of this type directly isn't possible, just like the parent type Rec:

Listing 93: show_object_declaration.adb

```
with Unknown_Discriminants.Children;
use  Unknown_Discriminants.Children;

procedure Show_Object_Declaration is
   A : Derived_Rec;
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 5d0b8980e6f60595b9de8a2ea8fa2132
```

Build output

```
show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need␣
↪initialization)
gprbuild: *** compilation phase failed
```

Deriving from tagged types

We can also derive from tagged types with unknown discriminants. Consider the following package:

Listing 94: unknown_discriminants.ads

```
package Unknown_Discriminants is

   type Rec (<>) is tagged private;

private

   type Rec is tagged null record;

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
↪Tagged_Type
MD5: ef66d098df1c93495bf5f6c6ac86f203
```

We can derive from the Rec type. In this case, however, we can use an unknown discriminant part, a known discriminant part, or no discriminants:

Listing 95: unknown_discriminants-children.ads

```
package Unknown_Discriminants.Children is

   type Derived_Rec_Unknown_Discr (<>) is
     new Rec with private;

   type Derived_Rec_Known_Discr (L : Positive) is
     new Rec with private;

   type Derived_Rec_No_Discr is
     new Rec with private;

private

   type Derived_Rec_Unknown_Discr is
     new Rec with null record;

   type Derived_Rec_Known_Discr (L : Positive) is
     new Rec with null record;

   type Derived_Rec_No_Discr is
     new Rec with null record;

end Unknown_Discriminants.Children;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
↪Tagged_Type
MD5: 98583f0b39c6f8bea49d1781844bb33e
```

In this example, we declare Derived_Rec_Unknown_Discr with an unknown discriminant part, Derived_Rec_Known_Discr with a known discriminant part, and Derived_Rec_No_Discr with no discriminants.

As expected, Derived_Rec_Unknown_Discr has unknown discriminants because it has an unknown discriminant part. In the case of Derived_Rec_No_Discr, which has no discriminants, we're deriving the unknown discriminants of Rec, so it also has unknown discriminants. In contrast, because Derived_Rec_Known_Discr has a known discriminant part, those discriminants are overriding the unknown discriminants of the parent type Rec.

Therefore, we can declare objects of Derived_Rec_Known_Discr type without explicit initialization:

Listing 96: show_object_declaration.adb

```
with Unknown_Discriminants.Children;
use  Unknown_Discriminants.Children;

procedure Show_Object_Declaration is
   A : Derived_Rec_Unknown_Discr;
   --  ERROR: unknown discriminants
   --         because of the type's
   --         unknown discriminant part

   B : Derived_Rec_Known_Discr (1);
   --  OK: known discriminants

   C : Derived_Rec_No_Discr;
   --  ERROR: unknown discriminants
   --         because of parent type's
   --         unknown discriminant part
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
↪Tagged_Type
MD5: 91f6ae2abf88976833d0e4eff02d4c40
```

Build output

```
show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need␣
↪initialization)
show_object_declaration.adb:13:08: error: unconstrained subtype not allowed (need␣
↪initialization)
gprbuild: *** compilation phase failed
```

As we can see, we can only directly declare objects of type Derived_Rec_Known_Discr because it has known discriminants, while the other two derived types have unknown discriminants — which are explicitly specified (Derived_Rec_Unknown_Discr) or implicitly derived from the parent (Derived_Rec_No_Discr).

Note that the parent type Rec had a requirement for explicit initialization. By using known discriminants in the declaration of Derived_Rec_Known_Discr, we're removing this requirement for the derived type.

The contrary is also true: we can derive a type with known discriminants and use an unknown discriminant part:

Listing 97: unknown_discriminants-children-grand.ads

```
package Unknown_Discriminants.Children.Grand is

   type Grand_Rec_Unknown_Discr (<>) is
     new Derived_Rec_Known_Discr (1)
       with private;

private

   type Grand_Rec_Unknown_Discr is
     new Derived_Rec_Known_Discr (1)
```

(continues on next page)

(continued from previous page)

```
      with null record;

end Unknown_Discriminants.Children.Grand;
```

Listing 98: show_object_declaration.adb

```
with Unknown_Discriminants.Children.Grand;
use  Unknown_Discriminants.Children.Grand;

procedure Show_Object_Declaration is
   A : Grand_Rec_Unknown_Discr;
   --  ERROR: unknown discriminants
   --         because of the type's
   --         unknown discriminant part
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
↪Tagged_Type
MD5: 0e931df8294cee1f49b187c43614aa20
```

Build output

```
show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need␣
↪initialization)
show_object_declaration.adb:5:08: error: provide initial value or explicit␣
↪discriminant values
show_object_declaration.adb:5:08: error: or give default discriminant values for␣
↪type "Grand_Rec_Unknown_Discr"
gprbuild: *** compilation phase failed
```

In this example, `Grand_Rec_Unknown_Discr` has unknown discriminants and requires explicit initialization, even though its parent type `Derived_Rec_Known_Discr` has known discriminants.

In the Ada Reference Manual

- 3.7 Discriminants[102]

4.7 Unconstrained subtypes

A subtype is called an unconstrained subtype if its type has unknown discriminants. Consider a simple Rec type:

Listing 99: unknown_discriminants.ads

```
package Unknown_Discriminants is

   type Rec (<>) is private;

private

```

(continues on next page)

[102] http://www.ada-auth.org/standards/12rm/html/RM-3-7.html

(continued from previous page)

```
   type Rec is null record;

end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.
↪Unconstrained_Subtype
MD5: 948e7c7ecd00915fa23a98cbaf2bbcbe
```

A subtype of Rec type is unconstrained:

Listing 100: unknown_discriminants-children.ads

```
package Unknown_Discriminants.Children is

   subtype Rec_Unconstrained is Rec;

end Unknown_Discriminants.Children;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.
↪Unconstrained_Subtype
MD5: 6b76b6a94d8c9487dbeea3256d5de01f
```

In this example, Rec_Unconstrained is an unconstrained subtype because it's derived from the Rec type. We can verify this by triggering a compilation error:

Listing 101: show_object_declaration.adb

```
with Unknown_Discriminants.Children;
use  Unknown_Discriminants.Children;

procedure Show_Object_Declaration is
   A : Rec_Unconstrained;
begin
   null;
end Show_Object_Declaration;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 442fab4d174de31f27d0de56bf9b8422
```

Build output

```
show_object_declaration.adb:5:08: error: "Rec_Unconstrained" is undefined
gprbuild: *** compilation phase failed
```

In addition, if we declare a subtype based on a type that allows range, index, or discriminant constraints, but we don't constraint the subtype, this subtype is also considered an unconstrained subtype. For example:

Listing 102: unconstrained_subtypes.ads

```
package Unconstrained_Subtypes is

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

(continues on next page)

(continued from previous page)

```

   type Rec (L : Positive) is
     null record;

   subtype Arr_Sub is Arr;
   --                 ^^^
   --   no constraints

   subtype Rec_Sub is Rec;
   --                 ^^^
   --   no constraints

end Unconstrained_Subtypes;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Other_
↪Unconstrained_Subtypes
MD5: 3ebc2eb371472dc76eb543b4633e59b3
```

In this example, Arr_Sub and Rec_Sub are unconstrained subtypes.

In the Ada Reference Manual

- 3.2 Types and Subtypes[103]

4.8 Variant parts

We've introduced variant records back in the Introduction to Ada course[104]. In simple terms, a variant record is a record with discriminants that allows for varying its structure. Basically, it's a record containing a **case** statement that specifies which record components exist for each discriminant value. For example:

Listing 103: devices.ads

```
package Devices is

   type Device_State is
     (Off, On);

   type Device_Info is
   record
      V : Float;
   end record;

   type Device (State : Device_State := Off) is
   record
      case State is
         when Off =>
            null;
         when On =>
            Info : Device_Info;
      end case;
   end record;
```

(continues on next page)

103 http://www.ada-auth.org/standards/12rm/html/RM-3-2.html
104 https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_records.html#intro-ada-variant-records

(continued from previous page)

```

end Devices;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Simple_Device
MD5: 3b63a63aef1d9cb00be870c831829158
```

The Device type from this example has a variant part. Depending on the value of the State discriminant, it can be either a null record (when State is Off) or have the Info component (when State is On).

Let's look at a test application for the Devices package:

Listing 104: show_device.adb

```
with Devices; use Devices;

procedure Show_Device is
   D     : Device;
   D_Off : Device (Off);
   D_On  : Device (On);
begin
   D := D_Off;
   --  OK!

   D := D_On;
   --  OK!

   D_Off := D_On;
   --       ^^^^
   --  CONSTRAINT_ERROR!

   D_On  := D_Off;
   --       ^^^^^
   --  CONSTRAINT_ERROR!
end Show_Device;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Simple_Device
MD5: a11e2739131f435e8428a5e2a9a478e7
```

Build output

```
show_device.adb:11:09: warning: "D_On" may be referenced before it has a value [enabled by default]
show_device.adb:14:13: warning: incorrect value for discriminant "State" [enabled by default]
show_device.adb:14:13: warning: Constraint_Error will be raised at run time [enabled by default]
show_device.adb:18:13: warning: incorrect value for discriminant "State" [enabled by default]
show_device.adb:18:13: warning: Constraint_Error will be raised at run time [enabled by default]
```

Runtime output

```

raised CONSTRAINT_ERROR : show_device.adb:14 discriminant check failed
```

As we've discussed *previously* (page 197), when we set the values for the discriminants of a type in the object declaration, we're constraining the objects. If the discriminants of two objects don't match, the `Constraint_Error` exception is raised at runtime because the *discriminant check* (page 515) fails. Therefore, in the `Show_Device` procedure, because `D_Off` and `D_On` are constrained and have different values for the `State` discriminant, we cannot assign them to each other. In contrast, because `D` wasn't constrained at its declaration, we can assign objects with different discriminants (such as `D_Off` and `D_On`) to it.

Note that the variant part of a record can be more complex. For example, we could have an additional discriminant and use it in the variant part:

Listing 105: devices.ads

```
package Devices is

   type Device_State is
     (Off, On);

   type Device_Info is
   record
      V : Float;
   end record;

   type Device (State : Device_State;
                Boost : Boolean) is
   record
      case State is
         when Off =>
            null;
         when On =>
            Info : Device_Info;
            case Boost is
               when False =>
                  null;
               when True =>
                  Factor : Float;
            end case;
      end case;
   end record;

end Devices;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device_Boost
MD5: 4c5e84ccebca9e4ef5e2d6d131ba0e6a
```

In this version of the `Devices` package, we introduced a *boost button* as a discriminant (`Boost`) and an associated boost factor component (`Factor`) in the variant part.

In the remaining parts of this section, we discuss a couple of details about variant records.

In the Ada Reference Manual

- 3.8.1 Variant Parts and Discrete Choices[105]

[105] http://www.ada-auth.org/standards/12rm/html/RM-3-8-1.html

4.8.1 Discriminant type and value coverage

The subtype of discriminants used in the variant part must be of a discrete type — it cannot be of an access or a floating-point type, for example. Also, all possible values of the subtype of each discriminant must be covered in the case statement of the variant part. For example, consider the following variant record:

Listing 106: subtype_coverage.ads

```
package Subtype_Coverage is

   type Var_Rec (Value : Integer) is
   record
      case Value is
         when 0 .. 100 =>
            I : Integer;

         --  ERROR: missing values!
      end case;
   end record;

end Subtype_Coverage;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Coverage
MD5: 084a468bc8d6f63d21f804c9ddc70622
```

Build output

```
subtype_coverage.ads:5:07: error: missing case values: -16#8000_0000# .. -1
subtype_coverage.ads:5:07: error: missing case values: 101 .. 16#7FFF_FFFF#
gprbuild: *** compilation phase failed
```

This package cannot be compiled because, in the variant part, we're only covering values for the `Value` discriminant in the range between 0 and 100. To fix this compilation error, we have to cover all values instead. For example:

Listing 107: subtype_coverage.ads

```
package Subtype_Coverage is

   type Var_Rec (Value : Integer) is
   record
      case Value is
         when Integer'First .. -1 =>
            null;
         when 0 .. 100 =>
            I : Integer;
         when 101 .. Integer'Last =>
            null;
      end case;
   end record;

end Subtype_Coverage;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Coverage
MD5: 9dfa0dfc3d3e11394a79b1ab6b61bafc
```

Of course, specifying all possible values can be difficult. As an alternative, we could simplify the case statement by just using **others** as a discrete choice that encompasses all values that haven't been specified earlier in the case statement:

Listing 108: subtype_coverage.ads

```
package Subtype_Coverage is

   type Var_Rec (Value : Integer) is
   record
      case Value is
         when 0 .. 100 =>
            I : Integer;
         when others =>
            null;
      end case;
   end record;

end Subtype_Coverage;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Coverage
MD5: 0b28038d5137de702cb5b8e875fadefa
```

By using `when others => ...` in this last example, we ensure that all values have been covered.

4.8.2 Record size

When declaring an object, the values we select for the discriminants related to the variant part have an impact on the overall size of that object — in fact, it may be smaller or bigger depending on this selection. Let's see an example:

Listing 109: variant_records.ads

```
package Variant_Records is

   type Simple_Record
     (Extended : Boolean := False) is
   record
      V : Integer;
      case Extended is
         when False =>
            null;
         when True  =>
            V_Float : Float;
      end case;
   end record;

end Variant_Records;
```

Listing 110: show_variant_rec_size.adb

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

with Variant_Records; use Variant_Records;

procedure Show_Variant_Rec_Size is
   SR_No_Ext : Simple_Record
                 (Extended => False);
   SR_Ext    : Simple_Record
                 (Extended => True);
```

(continues on next page)

(continued from previous page)

```ada
   SR        : Simple_Record;
begin
   Put_Line ("SR_No_Ext'Size : "
             & SR_No_Ext'Size'Image
             & " bits");
   Put_Line ("SR_Ext'Size : "
             & SR_Ext'Size'Image
             & " bits");
   Put_Line ("SR'Size : "
             & SR'Size'Image
             & " bits");
end Show_Variant_Rec_Size;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Size
MD5: 4aaf10924e7469d000cefeb70a69a2fa
```

Build output

```
show_variant_rec_size.adb:6:04: warning: variable "SR_No_Ext" is read but never␣
↪assigned [-gnatwv]
show_variant_rec_size.adb:8:04: warning: variable "SR_Ext" is read but never␣
↪assigned [-gnatwv]
show_variant_rec_size.adb:10:04: warning: variable "SR" is read but never assigned␣
↪[-gnatwv]
```

Runtime output

```
SR_No_Ext'Size :  64 bits
SR_Ext'Size :  96 bits
SR'Size :  96 bits
```

As we can confirm when we run this application, the choice for the discriminant has an impact on the size of the object. In the case of the SR_No_Ext object, setting the Extended discriminant to **False** excludes the V_Float component. For the SR_Ext object, on the other hand, we include the V_Float component. Therefore, on a typical PC, the size of SR_No_Ext is 8 bytes (4 bytes for the Extended discriminant and 4 bytes for the V component), while the size of SR_Ext is 12 bytes (i.e., additional 4 bytes for the V_Float component).

In the case of SR, because the object isn't constrained, the size of the object is 12 bytes on a typical PC — the same size as SR_Ext. This is because SR has to account for the case when all components must be available, even though the Extended discriminant is set to **False** by default. Remember that an assignment such as SR := SR_Ext is valid, so enough memory must be available to ensure that the assignment is performed correctly.

This principle applies to more complicated variant records. For example:

Listing 111: variant_records.ads

```ada
package Variant_Records is

   type Simple_Record
     (Extended   : Boolean := False;
      Extended_2 : Boolean := False) is
   record
      V : Integer;
      case Extended is
         when False =>
            case Extended_2 is
               when False =>
```

(continues on next page)

(continued from previous page)

```
                  null;
               when True  =>
                  V_Int_2 : Integer;
                  V_Int_3 : Integer;
            end case;
         when True  =>
            V_Float : Float;
            case Extended_2 is
               when False =>
                  null;
               when True  =>
                  V_Float_2 : Float;
            end case;
      end case;
   end record;

end Variant_Records;
```

Listing 112: show_variant_rec_size.adb

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

with Variant_Records; use Variant_Records;

procedure Show_Variant_Rec_Size is
   SR : Simple_Record;
begin
   Put_Line ("SR'Size : "
             & SR'Size'Image
             & " bits");
end Show_Variant_Rec_Size;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Size
MD5: 5f0ac936a5fee50cbe88a7b863a1a550
```

Build output

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

Runtime output

```
SR'Size :  128 bits
```

In this example, the size of SR is 16 bytes on a typical PC. This accounts for 4 bytes for the discriminants Extended and Extended_2, and 4 bytes for each of the 3 components that are being taken into account for the worst case:

- components V, V_Int_2 and V_Int_3 when we set Extended => **False**, Extended_2 => **True**;
- components V, V_Float and V_Float_2 when we set Extended => **True**, Extended_2 => **True**.

Note that a memory block is shared between the V_Int_2 and V_Int_3 components from the first worst case, and V_Float and the V_Float_2 components from the second worst case. As we can see, the compiler will typically optimize the size of a record as much as possible by assessing which components are really needed for the worst case.

Also, as we discussed previously, we can use *unchecked unions* (page 121) in combination with variant records, which has an impact on the object size.

4.8.3 Ensuring valid information

We can use variant parts to prevent invalid information from being used. Let's look again at the Device type from the previous code example:

```
type Device (State : Device_State) is
record
   case State is
      when Off =>
         null;
      when On =>
         Info : Device_Info;
   end case;
end record;
```

For the sake of this example, we could say that a device that is turned off doesn't have any valuable information. Therefore, the device information stored in the Info component of the Device type is only valid if the device is turned on. Thus, if the device is turned off (i.e., Device_State = Off), we should prevent the application from processing device information that is probably incorrect. Let's extend the previous code example to accommodate this requirement:

Listing 113: devices.ads

```
package Devices is

   type Device_State is
     (Off, On);

   type Device
     (State : Device_State := Off) is
       private;

   procedure Turn_Off (D : in out Device);

   procedure Turn_On (D : in out Device);

   type Device_Info is
   record
      V : Float;
   end record;

   function Current_Info (D : Device)
                          return Device_Info;

private

   type Device (State : Device_State := Off) is
   record
      case State is
         when Off =>
            null;
         when On =>
            Info : Device_Info;
      end case;
   end record;

   Device_Off : constant Device :=
```

(continues on next page)

(continued from previous page)

```
                  (State => Off);

   Device_On  : constant Device :=
                  (State => On,
                   others => <>);

end Devices;
```

Listing 114: devices.adb

```
package body Devices is

   procedure Turn_Off (D : in out Device) is
   begin
      D := Device_Off;
   end Turn_Off;

   procedure Turn_On (D : in out Device) is
   begin
      D := Device_On;
   end Turn_On;

   function Current_Info (D : Device)
                          return Device_Info is
     (D.Info);

end Devices;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device
MD5: e03db406de3550865dd99986d2c71145
```

Let's create a test application called Show_Device that makes use of this device by turning it on and off, and by retrieving information from it:

Listing 115: show_device.adb

```
with Devices; use Devices;

procedure Show_Device is
   D : Device;
   I : Device_Info;
begin
   Turn_On (D);
   I := Current_Info (D);

   Turn_Off (D);

   --  The following call raises
   --  an exception at runtime
   --  because D is turned off.
   I := Current_Info (D);
end Show_Device;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device
MD5: cba0100ad5bbb2b6bf00d0847a700271
```

Runtime output

```
raised CONSTRAINT_ERROR : devices.adb:15 discriminant check failed
```

In this example, by using the variant part, we're preventing information retrieved by an inappropriate call to the `Current_Info` function from being used elsewhere in the application. In fact, if the device is turned off, a call to `Current_Info` raises the `Constraint_Error` exception because the `Info` component isn't accessible. We see that effect in the `Show_Device` procedure: the call to `Current_Info` *fails* (by raising an exception) when the device has just been turned off.

To avoid exceptions at runtime, we must check the device's state before calling `Current_Info`:

Listing 116: show_device.adb

```
with Devices; use Devices;

procedure Show_Device is
   D : Device;
   I : Device_Info;
begin
   Turn_On (D);

   if D.State = On then
      I := Current_Info (D);
   end if;

   Turn_Off (D);

   if D.State = On then
      I := Current_Info (D);
   end if;
end Show_Device;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device
MD5: 62230848af720b156f22c96d59f772d2
```

Now, no exception is raised, as we only retrieve information from the device when it is turned on — that is, we only call the `Current_Info` function when the `State` discriminant of the object is set to `On`.

4.8.4 Extending record types

We can use variant parts as a means to extend record types. This can be viewed as a static approach to implement type extension — similar to type extension via tagged types, but with clear differences.

Let's say we have a sensor, and we implement a package called `Sensors` that interfaces with that sensor:

Listing 117: sensors.ads

```
package Sensors is

   type Sensor is private;

   type Sensor_Info is
   record
      Info_1 : Float := 0.0;
```

(continues on next page)

(continued from previous page)

```ada
   end record;

   function Current_Info (S : Sensor)
                          return Sensor_Info;

   procedure Display (SI : Sensor_Info);

private

   type Sensor is null record;

end Sensors;
```

Listing 118: sensors.adb

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

package body Sensors is

   function Current_Info (S : Sensor)
                          return Sensor_Info is
     ((Info_1 => 4.0));
   --             ^^^^
   --  NOTE: we're returning dummy
   --        information!

   procedure Display (SI : Sensor_Info) is
   begin
      Put_Line ("Info_1 : "
                & SI.Info_1'Image);
   end Display;

end Sensors;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 140a0d9cbca023de875417409c3f67d9
```

The `Sensor` type from the `Sensors` package has two subprograms: the `Current_Info` function and the `Display` procedure. We use those subprograms in the `Show_Sensors` procedure below:

Listing 119: show_sensors.adb

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

with Sensors;     use Sensors;

procedure Show_Sensors is
   S1 : Sensor;
begin
   Display (Current_Info (S1));
end Show_Sensors;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 93aa76da463fea9b4483ed97fa8bcf64
```

Runtime output

```
Info_1 :  4.00000E+00
```

Now, let's assume that a new model of this sensor is available, and it has additional features — e.g., it provides additional information to the user. If we wanted to update the application to be able to handle this new model of the sensor without removing support for the original model, we could convert the `Sensor_Info` type to a tagged type and derive a `Sensor_Info_V2` type from it. (We would probably have to implement a `Sensor_V2` type derived from the `Sensor` type as well.)

Alternatively, we could add a variant part to the `Sensor_Info` type to store the additional information. For example:

Listing 120: sensors.ads

```ada
package Sensors is

   type Sensor_Model is (Sensor_V1,
                         Sensor_V2);

   type Sensor
     (Model : Sensor_Model := Sensor_V1) is
       private;

   type Sensor_Info
     (Model : Sensor_Model := Sensor_V1) is
   record
      Info_1 : Float := 0.0;
      case Model is
         when Sensor_V1 =>
            null;
         when Sensor_V2 =>
            Info_2 : Float := 0.0;
      end case;
   end record;

   function Current_Info (S : Sensor)
                          return Sensor_Info;

   procedure Display (SI : Sensor_Info);

private

   type Sensor
     (Model : Sensor_Model := Sensor_V1) is
       null record;

end Sensors;
```

Listing 121: sensors.adb

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

package body Sensors is

   function Current_Info (S : Sensor)
                          return Sensor_Info is
   begin
      --  Using dummy info for the information
      --  returned by the function
      case S.Model is
         when Sensor_V1 =>
```

(continues on next page)

(continued from previous page)

```
            return ((Model  => Sensor_V1,
                     Info_1 => 4.0));
         when Sensor_V2 =>
            return ((Model  => Sensor_V2,
                     Info_1 => 8.0,
                     Info_2 => 6.0));
      end case;
   end Current_Info;

   procedure Display (SI : Sensor_Info) is
   begin
      Put_Line ("Model  : "
                & SI.Model'Image);
      Put_Line ("Info_1 : "
                & SI.Info_1'Image);
      if SI.Model = Sensor_V2 then
         Put_Line ("Info_2 : "
                   & SI.Info_2'Image);
      end if;
   end Display;

end Sensors;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 74198e928e3dfa3a7a7f2786971da8a7
```

In this new version of the Sensors package, the Model discriminant was added to the Sensor_Info type. If the model is set to version 2 for a specific sensor (i.e., Model = Sensor_V2), a new component (Info_2) is available.

The Current_Info and Display subprograms have been adapted to take this new model into account. In the Current_Info function, we return information for the newer model of the sensor. In the Display procedure, we display the additional information provided by the newer model.

Note that the original test application that makes use of the sensor (Show_Sensors) doesn't require any adaptation:

Listing 122: show_sensors.adb

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

with Sensors;     use Sensors;

procedure Show_Sensors is
   S1 : Sensor;
begin
   Display (Current_Info (S1));
end Show_Sensors;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 93aa76da463fea9b4483ed97fa8bcf64
```

Runtime output

```
Info_1 :  4.00000E+00
```

Because we have a default value for the discriminant of the Sensor type, we're essentially making the type *backwards-compatible*, so that users of this type don't have to adapt their code after the update to the Sensors package. Of course, we don't have *binary backwards-compatibility* because the size of the type (Sensor_Info'Size) increases.

Of course, in our test application, we can also use the new model of that sensor:

Listing 123: show_sensors.adb

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

with Sensors;     use Sensors;

procedure Show_Sensors is
   S1 : Sensor;
   S2 : Sensor (Sensor_V2);
begin
   Display (Current_Info (S1));
   Display (Current_Info (S2));
end Show_Sensors;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 347d272cddbacf7bf2987aa23014ff0b
```

Runtime output

```
Model  : SENSOR_V1
Info_1 :  4.00000E+00
Model  : SENSOR_V2
Info_1 :  8.00000E+00
Info_2 :  6.00000E+00
```

In the updated version of the Show_Sensors procedure, we're now using both old and new versions of the sensor.

4.9 Per-Object Expressions

In record type declarations, we might want to define a component that makes use of a *name* (page 5) that refers to a *discriminant* (page 192) 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 124: rec_per_object_expressions.ads

```
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);
      --                     ^^^^^^
      --
      --                          S
      --                          ^
      --    Per-object expression
      --
      --                   (1 .. S)
      --                   ^^^^^^^^
      --     Per-object constraint

      Top : Natural := 0;
   end record;

end Rec_Per_Object_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_
↪Expression
MD5: e4012454ea886fd429d82159b8d344b7
```

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 711):

Listing 125: rec_per_object_expressions.ads

```
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
      --                 ^^^^^^^^^^
      --
```

(continues on next page)

(continued from previous page)

```
      --    Per-object expression
      --
      --              (Selected_T)
      --              ^^^^^^^^^^^^
      --     Per-object constraint
   end record;

end Rec_Per_Object_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_
↪Expression_Access_Discriminant
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 126: rec_per_object_expressions.ads

```ada
package Rec_Per_Object_Expressions is

   type T is limited private;

private

   type T_Processor (Selected_T : access T) is
     null record;

   type T is limited record
      E : T_Processor (T'Access);
      --
      --               T'Access
      --               ^^^^^^^^
      --  Per-object expression
      --
      --              (T'Access)
      --              ^^^^^^^^^^
      --   Per-object constraint
   end record;

end Rec_Per_Object_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_
↪Expression_Access_Discriminant
MD5: a67b3034008fdf2a8c5fd1b6da769128
```

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*.

In the Ada Reference Manual

- 3.8 Record Types[106]

4.9.1 Default value

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

Listing 127: rec_per_object_expressions.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_
↪Expression_Default_Value
MD5: 70454b0b116094a02b897d8d1d0080fb
```

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 248) that exist for per-object constraints, as we discuss later on.) Such expressions might include function calls or uses of any defined operator. For this reason, the following code example is accepted by the compiler:

Listing 128: rec_per_object_expressions.ads

```
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;
```

(continues on next page)

106 http://www.ada-auth.org/standards/22rm/html/RM-3-8.html

(continued from previous page)

```

      Overflow_Warning : Positive
        := S * 9 / 10;
      --   ^^^^^^^^^^
      --   Per-object expression
      --   using computation for
      --   the default expression.
   end record
     with
       Dynamic_Predicate =>
         Overflow_Warning in
           (S + 1) / 2 .. S - 1;
      --
      --   (S + 1) / 2
      --   ^^^^^^^^^^^
      --   Per-object expression
      --   using computation.
      --
      --                  S - 1
      --                  ^^^^^
      --   Per-object expression
      --   using computation.

end Rec_Per_Object_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_
↪Expression_Computation
MD5: 6783568fd3e76a85ca7c1cc65ba023c5
```

In this example, we can identify multiple per-object expressions that use a computation: `S * 9 / 10`, `(S + 1) / 2`, and `S - 1`.

4.9.2 Restrictions

There are some important restrictions on per-object constraints:

1. Per-object range constraints such as `1 .. T'Size` are not allowed.
   - For example, the following code example doesn't compile:

Listing 129: rec_per_object_expressions.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_
↪Expressions.Per_Object_Expression_Range_Constraint
MD5: c2ac9588c1d1adac8c584a0e36a81342
```

Build output

```
rec_per_object_expressions.ads:8:30: error: in a constraint the␣
↪current instance can only be used with an access attribute
gprbuild: *** compilation phase failed
```

2. Within a per-object index constraint or discriminant constraint, each per-object expression must be the name of a discriminant directly, without any further computation.

   - Therefore, we're allowed to write (`1 .. S`) — as we've seen in a previous example —. However, writing (`1 .. S - 1`) would be illegal.
   - For example, the following adaptation to the previous code example doesn't compile:

Listing 130: 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);
      --                          ^^^^^
      --  ERROR: computation in per-object
      --         expression is illegal.

      Top : Integer := -1;
   end record;

end Rec_Per_Object_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_
↪Expressions.Per_Object_Expression_Range_Computation
MD5: 1224bb63f7953743d84a258226c35c50
```

Build output

```
rec_per_object_expressions.ads:11:33: error: discriminant in␣
↪constraint must appear alone
gprbuild: *** compilation phase failed
```

   In this example, using the computation `S - 1` to specify the range of `Arr` isn't permitted. (Note that, *as we've seen before* (page 247), this restriction doesn't apply when the computation is used in a per-object expression that calculates the default value of a component.)

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

CHAPTER

FIVE

AGGREGATES

5.1 Container Aggregates

Note

This feature was introduced in Ada 2022.

A container aggregate is a list of elements — such as [1, 2, 3] — that we use to initialize or assign to a container. For example:

Listing 1: show_container_aggregate.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Simple_
↪Container_Aggregate
MD5: b54cd5800179d4016bbce5a9b10734f2
```

In this example, [1.0, 2.0, 3.0] is a container aggregate that we use to initialize a vector V.

We can specify container aggregates in three forms:

- as a null container aggregate, which indicates a container without any elements and is represented by the [] syntax;
- as a positional container aggregate, where the elements are simply listed in a sequence (such as [1, 2]);
- as a named container aggregate, where a key is indicated for each element of the list (such as [1 => 10, 2 => 15]).

Let's look at a complete example:

Listing 2: show_container_aggregate.adb

```
with Ada.Containers.Vectors;

procedure Show_Container_Aggregate is

   package Float_Vec is new
     Ada.Containers.Vectors (Positive, Float);

   --  Null container aggregate
   Null_V  : constant Float_Vec.Vector :=
               [];

   --  Positional container aggregate
   Pos_V   : constant Float_Vec.Vector :=
               [1.0, 2.0, 3.0];

   --  Named container aggregate
   Named_V : constant Float_Vec.Vector :=
               [1 => 1.0,
                2 => 2.0,
                3 => 3.0];

   pragma Unreferenced (Null_V, Pos_V, Named_V);
begin
   null;
end Show_Container_Aggregate;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Simple_
↪Container_Aggregate
MD5: f00b21da1722669ae92bd5fe4a9a3966
```

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;

while

- 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

```
with Ada.Containers.Vectors;
with Ada.Containers.Indefinite_Hashed_Maps;
with Ada.Strings.Hash;

procedure Show_Named_Container_Aggregate is

   package Float_Vec is new
     Ada.Containers.Vectors (Positive, Float);

   package Float_Hashed_Maps is new
     Ada.Containers.Indefinite_Hashed_Maps
       (Key_Type        => String,
        Element_Type    => Float,
```

(continues on next page)

(continued from previous page)

```
        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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Named_
↪Container_Aggregate
MD5: 9d117543135e75e66801628ca29e32ef
```

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[107]

5.2 Record aggregates

We've already seen record aggregates in the Introduction to Ada[108] 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:

[107] http://www.ada-auth.org/standards/22rm/html/RM-4-3-5.html
[108] https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-aggregates

Listing 4: points.ads

```
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

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

package body Points is

   procedure Display (P : Point_3D) is
   begin
      Put_Line ("(X => "
                & Integer'Image (P.X)
                & ",");
      Put_Line (" Y => "
                & Integer'Image (P.Y)
                & ",");
      Put_Line (" Z => "
                & Integer'Image (P.Z)
                & ")");
   end Display;

end Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
MD5: fd01961cf1da9b48d2a6150da30f7377
```

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

Listing 6: show_record_aggregates.adb

```
with Points; use Points;

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

   Display (P);

   --  Named component association
   P := (X => 3,
         Y => 4,
         Z => 5);

   Display (P);
end Show_Record_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
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

```ada
with Points; use Points;

procedure Show_Record_Aggregates is
   P : Point_3D;
begin
   --  Positional and named component associations
   P := (3, 4,
         Z => 5);

   Display (P);
end Show_Record_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
MD5: 493a2a87b4b28dfb0882ad73acf84710
```

Runtime output

```
(X =>  3,
 Y =>  4,
 Z =>  5)
```

In this case, only the Z component has a named association, while the other components have a positional association.

Note that a positional association cannot follow a named association, so we cannot write `P := (3, Y => 4, 5);`, for example. Once we start using a named association for a component, we have to continue using it for the remaining components.

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

Listing 8: show_record_aggregates.adb

```ada
with Points; use Points;

procedure Show_Record_Aggregates is
   P : Point_3D;
begin
   --  Multiple component selection
   P := (X | Y => 5,
         Z     => 6);
```

(continues on next page)

(continued from previous page)

```

   Display (P);
end Show_Record_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
MD5: a4fde562fb60d290caf46d86b13e694b
```

Runtime output

```
(X =>  5,
 Y =>  5,
 Z =>  6)
```

Here, we assign 5 to both X and Y.

In the Ada Reference Manual

- 4.3.1 Record Aggregates[109]

5.2.1 <>

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:

Listing 9: show_record_aggregates.adb

```
with Points; use Points;

procedure Show_Record_Aggregates is
   P : Point_3D;
begin
   P := (0, 1, 2);
   Display (P);

   --  Specifying X component.
   P := (X => 42,
         Y => <>,
         Z => <>);
   Display (P);

   --  Specifying Y and Z components.
   P := (X => <>,
         Y => 10,
         Z => 20);
   Display (P);
end Show_Record_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
MD5: 25145e7cba5a566c518ac4218e550899
```

Runtime output

[109] http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html

```
(X =>  0,
 Y =>  1,
 Z =>  2)
(X =>  42,
 Y =>  1,
 Z =>  2)
(X =>  42,
 Y =>  10,
 Z =>  20)
```

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

- when we write `(X => 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 dependent 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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
MD5: 8a716db129e6f231c4003b77d8b61ea3
```

Then, writing <> makes use of those default values we've just specified:

Listing 11: show_record_aggregates.adb

```ada
with Points; use Points;

procedure Show_Record_Aggregates is
   P : Point_3D := (0, 0, 0);
```

(continues on next page)

(continued from previous page)

```
begin
   --  Using default value for
   --  all components
   P := (X => <>,
         Y => <>,
         Z => <>);
   Display (P);
end Show_Record_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
MD5: e64c6fe4e4b3dbaa084d9b97b4fb971f
```

Runtime output

```
(X =>  10,
 Y =>  20,
 Z =>  30)
```

Now, as expected, the default values of each component (10, 20 and 30) are used when we write <>.

Similarly, we can specify a default value for the type of each component. For example, let's declare a `Point_Value` type with a default value — using the `Default_Value` aspect — and use it in the `Point_3D` record type:

Listing 12: points.ads

```
package Points is

   type Point_Value is new Float
     with Default_Value => 99.9;

   type Point_3D is record
      X : Point_Value;
      Y : Point_Value;
      Z : Point_Value;
   end record;

   procedure Display (P : Point_3D);

end Points;
```

Listing 13: points.adb

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

package body Points is

   procedure Display (P : Point_3D) is
   begin
      Put_Line ("(X => "
                & Point_Value'Image (P.X)
                & ",");
      Put_Line (" Y => "
                & Point_Value'Image (P.Y)
                & ",");
      Put_Line (" Z => "
                & Point_Value'Image (P.Z)
```

(continues on next page)

(continued from previous page)

```
                  & ")");
   end Display;

end Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregate_Default_Value
MD5: 508d7f5e7d02da1677485f7d588847f6
```

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

Listing 14: show_record_aggregates.adb

```ada
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregate_Default_Value
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 <>.

5.2.2 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);
```

(continues on next page)

(continued from previous page)

```
   Display (P);

   --  Specifying all components
   P := (others => 256);
   Display (P);
end Show_Record_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↪Rec_Aggregates
MD5: 3146363eb36ab4485c7755794fb78bbc
```

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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregates_Others
MD5: 875e470aa2cbc5fcfefae649ed5528f6
```

If we had written an aggregate such as `(others => 256)` for an object of type `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

```
with Custom_Records; use Custom_Records;

procedure Show_Record_Aggregates_Others is
   Dummy : Integer_Float;
begin
   --  ERROR: components selected by
```

(continues on next page)

(continued from previous page)

```
   --         others must be of same
   --         type.
   Dummy := (others => 256);
end Show_Record_Aggregates_Others;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregates_Others
MD5: d543ee07e24caf63384ab0d140054be2
```

Build output

```
show_record_aggregates_others.adb:9:14: error: components in "others" choice must␣
↪have same type
show_record_aggregates_others.adb:9:24: error: expected type "Standard.Float"
show_record_aggregates_others.adb:9:24: error: found type universal integer
gprbuild: *** compilation phase failed
```

We can fix this compilation error by making sure that **others** only refers to components of the same type:

Listing 18: show_record_aggregates_others.adb

```
with Custom_Records; use Custom_Records;

procedure Show_Record_Aggregates_Others is
   Dummy : Integer_Float;
begin
   --  OK: components selected by
   --      others have Integer type.
   Dummy := (Y | Z  => 256.0,
             others => 256);
end Show_Record_Aggregates_Others;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregates_Others
MD5: d01977a49e08d2c6cb6b7788581ed56f
```

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

Listing 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregates_Others
MD5: db9b72ffc933436e76305887276eeafd
```

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

5.2.3 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)
                   & ")");
      when Dim_2 =>
         Put_Line ("   (X => "
                   & Integer'Image (P.X2)
                   & ",");
         Put_Line ("    Y => "
                   & Integer'Image (P.Y2)
                  & ")");
      when Dim_3 =>
         Put_Line ("   (X => "
                   & Integer'Image (P.X3)
                   & ",");
         Put_Line ("    Y => "
                   & Integer'Image (P.Y3)
                   & ",");
         Put_Line ("    Z => "
                   & Integer'Image (P.Z3)
                   & ")");
      end case;
   end Display;

end Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregate_Discriminant
MD5: bd71322a65ca50e1eefa0aedd407931a
```

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
   P1 : constant Point := (Dim_1, 0);

   --  Named component association
   P2 : constant Point := (D  => Dim_2,
                           X2 => 3,
                           Y2 => 4);

   --  Positional / named component association
   P3 : constant Point := (Dim_3,
                           X3 => 3,
                           Y3 => 4,
                           Z3 => 5);
begin
   Display (P1);
   Display (P2);
   Display (P3);
end Show_Rec_Aggregate_Discriminant;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↪Aggregate_Discriminant
MD5: d487e0c68ea69c3e0f2adb8ac958e31d
```

Runtime output

```
DIM_1
  (X =>  0)
DIM_2
  (X =>  3,
   Y =>  4)
DIM_3
  (X =>  3,
   Y =>  4,
   Z =>  5)
```

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.

5.3 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[110].

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

Listing 23: persons.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules_Aggregates.
↪Full_Coverage_Rules
MD5: 7755bffa8b4473c425ae5075e9c478e9
```

We can create an object of the type using an aggregate:

Listing 24: show_aggregate_init.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules_Aggregates.
↪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:

110 https://www.adacore.com/gems/gem-1

Listing 25: persons.ads

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules_Aggregates.
↪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 266) and case statements, but occasionally useful for *record aggregates* (page 253)):

Listing 26: show_aggregate_init_others.adb

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

with Persons; use Persons;

procedure Show_Aggregate_Init_Others is

   X : constant Person :=
         (Name   =>
            To_Unbounded_String ("John Doe"),
          others => 25);
begin
   null;
end Show_Aggregate_Init_Others;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules_Aggregates.
↪Full_Coverage_Rules
MD5: 6d26de8dd6820682cb9150dcbb40f106
```

According to the Ada RM, **others** here means precisely the same thing as Age | Shoe_Size. But that's wrong: what **others** really means is "all the other components, including the ones we might add next week or next year". That means you shouldn't use **others** unless you're pretty sure it should apply to all the cases that haven't been invented yet.

Later on, we'll discuss *full coverage rules for limited types* (page 816).

5.4 Array aggregates

We've already discussed array aggregates in the Introduction to Ada[111] course. Therefore, this section just presents some details about this topic.

In the Ada Reference Manual

- 4.3.3 Array Aggregates[112]

5.4.1 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 *record aggregates* (page 253), array aggregates can be positional or named. Consider this package:

Listing 27: points.ads

```
package Points is

   type Point_3D is array (1 .. 3) of Integer;

   procedure Display (P : Point_3D);

end Points;
```

Listing 28: points.adb

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

package body Points is

   procedure Display (P : Point_3D) is
   begin
      Put_Line ("(X => "
                & Integer'Image (P (1))
                & ",");
      Put_Line (" Y => "
                & Integer'Image (P (2))
                & ",");
      Put_Line (" Z => "
                & Integer'Image (P (3))
                & ")");
   end Display;

end Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: d4b3becacc321d20810c3c90f4d8b7ff
```

[111] https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-array-type-declaration
[112] http://www.ada-auth.org/standards/22rm/html/RM-4-3-3.html

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

Listing 29: show_array_aggregates.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: 2d65c026639d990e7f6a99f7616d7eb4
```

Runtime output

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

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

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: d4e4d3ab4b7d538fc4ef1e92d28e47d9
```

Runtime output

```
(X =>  0,
 Y =>  1,
 Z =>  2)
(X =>  0,
 Y =>  3,
 Z =>  4)
```

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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.
↪Array_Aggregates
MD5: 16df9c01e46623ca735b84167a11a0fd
```

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.

5.4.2 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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates_2
MD5: 8e6e4951c14dcc6e8dea9b6a76064930
```

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

Listing 34: show_array_aggregates.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates_2
MD5: 188f7b006c08927f8cad83557a5e1cd9
```

Runtime output

```
Length =  0
(
)
```

In this example, when we call the Display procedure, we confirm that N doesn't have any components.

5.4.3 |, <>, others

We've seen the following syntactic elements when we were discussing *record aggregates* (page 253): |, <> and **others**. We can apply them to array aggregates as well:

Listing 35: show_array_aggregates.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: 648d68f393107b138c6390c599d3d247
```

Runtime output

```
(X =>  0,
 Y =>  0,
 Z =>  0)
(X =>  3,
 Y =>  3,
 Z =>  4)
(X =>  1101901064,
 Y =>  5,
 Z =>  5)
```

In this example, we use the `|`, `<>` and **others** elements in a very similar way as we did with record aggregates. (See the comments in the code example for more details.)

Note that, as for record aggregates, the `<>` makes use of the default value (if it is available). We discuss this topic in more details *later on* (page 280).

5.4.4 ..

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

Listing 36: show_array_aggregates.adb

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: 656f44d37ce676b24e9d512639fd0adc
```

Runtime output

```
(X =>  0,
 Y =>  0,
 Z =>  0)
(X =>  3,
 Y =>  3,
 Z =>  4)
(X => -1964943016,
 Y =>  5,
 Z =>  5)
```

This example is a variation of the previous one. However, in this case, we're using ranges instead of the | and **others** syntax.

5.4.5 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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
 ↪Aggregates
MD5: 34bbc8e8bd0bd3f8b63d07fa881233bd
```

Build output

```
show_array_aggregates.adb:6:09: warning: too few elements for type "Point_3D"␣
 ↪defined at points.ads:3 [enabled by default]
show_array_aggregates.adb:6:09: warning: expected 3 elements; found 1 element␣
 ↪[enabled by default]
show_array_aggregates.adb:6:09: warning: Constraint_Error will be raised at run␣
 ↪time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_array_aggregates.adb:6 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

```
with Points; use Points;

procedure Show_Array_Aggregates is
   P : Point_3D;
```

(continues on next page)

(continued from previous page)

```
begin
   P := [1 => 4, others => 0];
   --  OK: unspecified components have a
   --      value of zero

   Display (P);
end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: 5f1b7e3778b7d5ec990fba9558495758
```

Runtime output

```
(X =>  4,
 Y =>  0,
 Z =>  0)
```

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

Listing 39: show_array_aggregates.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates_2
MD5: e185c823ca68e9193a0b12270ffebe61
```

Build output

```
show_array_aggregates.adb:4:27: error: "others" choice not allowed here
show_array_aggregates.adb:4:27: error: qualify the aggregate with a constrained␣
↪subtype to provide bounds for it
gprbuild: *** compilation phase failed
```

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

5.4.6 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

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: 68bddcec76f8431b16d1c090b74c2500
```

Runtime output

```
(X =>  0,
 Y =>  0,
 Z =>  0)
(X =>  3,
 Y =>  3,
 Z =>  4)
(X =>  99,
 Y =>  2,
 Z =>  3)
```

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:

Listing 41: show_array_aggregates.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates
MD5: 932ebc6e51c2146a726bad68b7f2cad0
```

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 `[for E of P => E * 2]` to double the value of each component.
2. We write `[for E of P => E + 1]` to increase the value of each component by one.

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

5.4.7 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

```
package Matrices is

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

   procedure Display (M : Matrix);
```

(continues on next page)

(continued from previous page)

```

end Matrices;
```

Listing 43: matrices.adb

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

package body Matrices is

   procedure Display (M : Matrix) is

      procedure Display_Row (M : Matrix;
                             I : Integer) is
      begin
         Put_Line ("   (");
         for J in M'Range (2) loop
            Put ("      "
                 & J'Image
                 & " => "
                 & M (I, J)'Image);
            if J /= M'Last (2) then
               Put_Line (",");
            else
               New_Line;
            end if;
         end loop;
         Put ("   )");
      end Display_Row;

   begin
      Put_Line ("Length (1) = "
                & M'Length (1)'Image);
      Put_Line ("Length (2) = "
                & M'Length (2)'Image);

      Put_Line ("(");
      for I in M'Range (1) loop
         Display_Row (M, I);
         if I /= M'Last (1) then
            Put_Line (",");
         else
            New_Line;
         end if;
      end loop;
      Put_Line (")");

   end Display;

end Matrices;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Matrix_
↪Aggregates
MD5: 55573272f8cc0621eef7c924cfd6366a
```

We can assign multidimensional aggregates to a matrix `M` using positional or named component association:

Listing 44: show_array_aggregates.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Matrix_
 ↪Aggregates
MD5: fe3cb6ee62422991b444c32239f72d05
```

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:

Listing 45: string_lists.ads

```ada
package String_Lists is

   type String_List is array (Positive range <>,
                              Positive range <>)
                              of Character;

   procedure Display (SL : String_List);

end String_Lists;
```

Listing 46: string_lists.adb

```ada
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 if;
      end loop;
      Put_Line (")");
   end Display;

end String_Lists;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.String_
↪Aggregates
MD5: aacdbb9aa2f3b6146d8a36ca7581fd18
```

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
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'],
          [1 => 'D',
           2 => 'E',
           3 => 'F']];

   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: 82c28e5d8e592403d8909b8eaa1fe356
```

Runtime output

```
Length (1) =  2
Length (2) =  3
(
  (ABC),
  (DEF)
)
Length (1) =  2
Length (2) =  3
(
  (ABC),
  (DEF)
)
Length (1) =  2
Length (2) =  3
```

(continues on next page)

(continued from previous page)

```
(
  (XYZ),
  (   )
)
```

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.)

5.4.8 <> 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

```
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

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

package body Points is

   procedure Display (P : Point_3D) is
   begin
      Put ("      (X => "
           & Point_Value'Image (P.X)
           & ",");
      New_Line;
      Put ("        Y => "
           & Point_Value'Image (P.Y)
           & ",");
      New_Line;
      Put ("        Z => "
           & Point_Value'Image (P.Z)
           & ")");
   end Display;

   procedure Display (PA : Point_3D_Array) is
   begin
      Put_Line ("(");
      for I in PA'Range (1) loop
```

(continues on next page)

(continued from previous page)

```
         Put_Line ("   "
                   & Integer'Image (I)
                   & " =>");
         Display (PA (I));
         if I /= PA'Last (1) then
            Put_Line (",");
         else
            New_Line;
         end if;
      end loop;
      Put_Line (")");
   end Display;

end Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Rec_Array_
↪Aggregates
MD5: ffaf3745621a30362c6aadaec2c3cef2
```

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

Listing 50: show_record_aggregates.adb

```
with Points; use Points;

procedure Show_Record_Aggregates is
   PA : Point_3D_Array (1 .. 2);
begin
   PA := [ (X => 3,
            Y => 4,
            Z => 5),
           (X => 6,
            Y => 7,
            Z => 8) ];
   Display (PA);

   --  Array components are
   --  uninitialized.
   PA := [1 => <>,
          2 => <>];
   Display (PA);
end Show_Record_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Rec_Array_
↪Aggregates
MD5: 4575fead51e24b1a06faf4581efad112
```

Runtime output

```
(
   1 =>
      (X =>  3,
       Y =>  4,
       Z =>  5),
   2 =>
      (X =>  6,
       Y =>  7,
```

(continues on next page)

(continued from previous page)

```
       Z =>  8)
)
(
   1 =>
      (X =>  4484528,
       Y =>  0,
       Z =>  1821779848),
   2 =>
      (X =>  32764,
       Y =>  1,
       Z =>  0)
)
```

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

```ada
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

```ada
with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Aggregates is
   N : Integer_Array (1 .. 4);
begin
   N := [for I in N'Range => Value (I)];
   Display (N);

   N := [others => <>];
   Display (N);
end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates_2
MD5: 8007fb4af578397d1f07ad85e09ab354
```

Runtime output

```
Length =  4
(
   1 =>  1,
   2 =>  2,
   3 =>  3,
```

(continues on next page)

(continued from previous page)

```
   4 =>  4
)
Length =  4
(
   1 =>  99,
   2 =>  99,
   3 =>  99,
   4 =>  99
)
```

When writing an aggregate for the `Point_3D` type, any component that has `<>` gets the default value of the `Point` type (99):

For further reading...

Similarly, we could specify the `Default_Component_Value` aspect (which we discussed *earlier on* (page 70)) in the declaration of the array type:

Listing 53: integer_arrays.ads

```
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;
```

Listing 54: show_array_aggregates.adb

```
with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Aggregates is
   N : Integer_Array (1 .. 4);
begin
   N := [for I in N'Range => Value (I)];
   Display (N);

   N := [others => <>];
   Display (N);
end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates_2
MD5: 3f535bc5ce7f74ab0f0f48098a82c98a
```

Runtime output

```
Length =  4
(
   1 =>  1,
   2 =>  2,
   3 =>  3,
   4 =>  4
)
Length =  4
(
   1 =>  9999,
   2 =>  9999,
   3 =>  9999,
   4 =>  9999
)
```

In this case, when writing <> for a component, the value specified in the Default_Component_Value aspect is used.

Finally, we might want to use both Default_Value (which we discussed *previously* (page 69)) 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

```
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

```
with Integer_Arrays; use Integer_Arrays;

procedure Show_Array_Aggregates is
   N : Integer_Array (1 .. 4);
begin
   N := [for I in N'Range => Value (I)];
   Display (N);

   N := [others => <>];
   Display (N);
end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↪Aggregates_2
MD5: e58618b565874acaa99c5d494c2acaa4
```

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.

5.5 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[113]

5.5.1 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.

We can translate the description above to the following code:

```
   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.

5.5.2 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

```
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;
```

(continues on next page)

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

(continued from previous page)

```
   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

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

package body Points is

   procedure Display (P : Point_1D) is
   begin
      Put_Line ("(X => " & P.X'Image & ")");
   end Display;

   procedure Display (P : Point_2D) is
   begin
      Put_Line ("(X => " & P.X'Image
                & ", Y => " & P.Y'Image & ")");
   end Display;

   procedure Display (P : Point_3D) is
   begin
      Put_Line ("(X => " & P.X'Image
                & ", Y => " & P.Y'Image
                & ", Z => " & P.Z'Image & ")");
   end Display;

end Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↪Aggregate_Points
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;"

```

(continues on next page)

(continued from previous page)

```
   P_2D.Y := 0.7;

   Display (P_2D);
end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↪Aggregate_Points
MD5: 68ae6fa8e6f779aebea97085bd75e082
```

Runtime output

```
(X =>  5.00000E-01)
(X =>  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.

5.5.3 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:

Listing 60: 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);

   P_2D := (P_1D with Y => 0.7);
   Display (P_2D);
end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↪Aggregate_Points
MD5: 4d03f6a565126b602d6f21fe5ee6dd27
```

Runtime output

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

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);
```

5.5.4 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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↪Aggregate_Points
MD5: 2ec6831557c43f697bffce8496962b53
```

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_1 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.)

5.5.5 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

```
with Points; use Points;

procedure Show_Points is
   P_1D : Point_1D;
   P_2D : Point_2D;
   P_3D : Point_3D;
begin
   P_1D := (X => 0.5);
   P_2D := (P_1D with Y => 0.7);

   --  Initialize P_3D with P_1D and set other
   --  components to 0.6.
   --
   P_3D := (P_1D with others => 0.6);
   Display (P_3D);

   --  Initialize P_3D with P_2D, and other
   --  components with their default value.
   --
   P_3D := (P_2D with others => <>);
   Display (P_3D);
end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↪Aggregate_Points
MD5: 0594586fc59ead106258cef8682927e9
```

Runtime output

```
(X =>  5.00000E-01, Y =>  6.00000E-01, Z =>  6.00000E-01)
(X =>  5.00000E-01, Y =>  7.00000E-01, Z =>  5.93540E-39)
```

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 => <>`.

5.5.6 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;
```

Listing 64: show_points.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↪Aggregate_Points
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.

5.5.7 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:

Listing 65: show_points.adb

```
with Points; use Points;

procedure Show_Points is
   P_2D : Point_2D;
   P_3D : Point_3D;
begin
```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↪Aggregate_Points
MD5: ae5e88a36c58b1eb495d5ba8752e50e7
```

Runtime output

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

Here, we're using `Point_1D (P_3D)` to select the `Point_1D` view of an object of `Point_3D` type. At this point, we have specified the `Point_1D` part of the aggregate, so we still have to specify the remaining components of the `Point_2D` type — the Y component, to be more specific. When we do that, we get the appropriate aggregate for the `Point_2D` type. In summary, by carefully selecting the appropriate view, we're able to initialize an object of ascending type (`Point_2D`), which contains less components, using an object of a descendent type (`Point_3D`), which contains more components.

5.6 Delta Aggregates

Note

This feature was introduced in Ada 2022.

Previously, we've discussed *extension aggregates* (page 287), which are used to assign an object `Obj_From` of a tagged type to an object `Obj_To` of a descendent type.

We may want also to assign an object `Obj_From` of to an object `Obj_To` of the same type, but change some of the components in this assignment. To do this, we use delta aggregates.

5.6.1 Delta Aggregates for Tagged Records

Let's reuse the `Points` package from a previous example:

Listing 66: points.ads

```
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;
```

(continues on next page)

(continued from previous page)

```
   end record;

   procedure Display (P : Point_3D);

end Points;
```

Listing 67: points.adb

```
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 68: show_points.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↪Aggregates_Tagged
MD5: 23e9f53d626e32fc0524abfa0a437dbf
```

Runtime output

```
(X =>  5.00000E-01, Y =>  7.00000E-01, Z =>  3.00000E-01)
(X =>  1.00000E+00, Y =>  7.00000E-01, Z =>  3.00000E-01)
(X =>  2.00000E-01, Y =>  3.00000E-01, Z =>  3.00000E-01)
```

Here, we assign P1 to P2, but change the X component. Also, we assign P1 to P3, but change the X and Y components.

We can use class-wide types with delta aggregates. Consider this example:

Listing 69: show_points.adb

```
with Points; use Points;

procedure Show_Points is

```

(continues on next page)

(continued from previous page)

```ada
   P_3D : Point_3D;

   function Reset (P_2D : Point_2D'Class)
                   return Point_2D'Class is
     ((P_2D with delta X | Y => 0.0));

begin
   P_3D := (X => 0.1, Y => 0.2, Z => 0.3);
   Display (P_3D);

   P_3D := Point_3D (Reset (P_3D));
   Display (P_3D);

end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↪Aggregates_Tagged
MD5: dca144fe420dd37e224d089458f9e8a8
```

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;
```

5.6.2 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
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↪Aggregates_Non_Tagged
MD5: 1f12f33ac0a84919978c56d04f479e35
```

Runtime output

```
(X =>  5.00000E-01, Y =>  7.00000E-01, Z =>  3.00000E-01)
(X =>  1.00000E+00, Y =>  7.00000E-01, Z =>  3.00000E-01)
(X =>  2.00000E-01, Y =>  3.00000E-01, Z =>  3.00000E-01)
```

In this example, Point_3D is a non-tagged type. Note that we haven't changed anything in the Show_Points procedure: it still works as it did with tagged types.

5.6.3 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:

Listing 74: points.ads

```
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;
```

Listing 75: points.adb

```
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;
```

Listing 76: show_points.adb

```
with Points; use Points;

procedure Show_Points is
   P1, P2, P3 : Point_3D;
begin
   P1 := [0.5, 0.7, 0.3];
   Display (P1);

   P2 := [P1 with delta 1 => 1.0];
   Display (P2);

   P3 := [P1 with delta 1 => 0.2, 2 => 0.3];
   --  Alternatively:
   --  P3 := [P1 with delta 1 .. 2 => 0.2, 0.3];

   Display (P3);
end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↪Aggregates_Array
MD5: 06293882e5dd020f56fbced6bc03ccf0
```

Runtime output

```
( 1 =>  5.00000E-01 2 =>  7.00000E-01 3 =>  3.00000E-01)
( 1 =>  1.00000E+00 2 =>  7.00000E-01 3 =>  3.00000E-01)
( 1 =>  2.00000E-01 2 =>  3.00000E-01 3 =>  3.00000E-01)
```

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.

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

```ada
with Points; use Points;

procedure Show_Points is
   P1, P3 : Point_3D;
begin
   P1 := [0.5, 0.7, 0.3];
   Display (P1);

   P3 := [P1 with delta
            P3'First + 1 .. P3'Last => 0.0];
   Display (P3);
end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↪Aggregates_Array
MD5: 0a00e17b2d803f23edc728969d663c59
```

Runtime output

```
( 1 =>  5.00000E-01 2 =>  7.00000E-01 3 =>  3.00000E-01)
( 1 =>  5.00000E-01 2 =>  0.00000E+00 3 =>  0.00000E+00)
```

In this example, we're assigning P1 to P3, but resetting all components of the array starting by the second one.

Multiple components

We can also assign multiple components or slices:

Listing 78: float_arrays.ads

```ada
package Float_Arrays is

   type Float_Array is
```

(continues on next page)

(continued from previous page)

```
      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

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↪Aggregates_Array
MD5: 37063cd1c6cd46522d8e5b0df7b5741b
```

Runtime output

```
( 1 =>  1.00000E+00 2 =>  2.00000E+00 3 =>  3.00000E+00 4 =>  4.00000E+00 5 =>  5.
↪00000E+00)
( 1 =>  1.00000E+00 2 =>  0.00000E+00 3 =>  0.00000E+00 4 =>  2.00000E-01 5 =>  2.
↪00000E-01)
```

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[114]

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

CHAPTER

SIX

ARRAYS

6.1 Array constraints

Array constraints are important in the declaration of an array because they define the total size of the array. In fact, arrays must always be constrained. In this section, we start our discussion with unconstrained array types, and then continue with constrained arrays and arrays types. Finally, we discuss the differences between unconstrained arrays and vectors.

In the Ada Reference Manual

- 3.6 Array Types[115]

6.1.1 Unconstrained array types

In the Introduction to Ada course[116], 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

```
package Measurement_Defs is

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

end Measurement_Defs;
```

Listing 2: show_measurements.adb

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

with Measurement_Defs; use Measurement_Defs;

procedure Show_Measurements is
   M : Measurements (1 .. 10);
   --                ^ Providing bounds here!
begin
   Put_Line ("First index: " & M'First'Image);
   Put_Line ("Last index:  " & M'Last'Image);
end Show_Measurements;
```

115 http://www.ada-auth.org/standards/22rm/html/RM-3-6.html
116 https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-types

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Constraints.Unconstrained_
↪Array_Type
MD5: a5cdc74dd61e36476431cf675452d1d5
```

Build output

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

Runtime output

```
First index:  1
Last index:   10
```

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.

6.1.2 Constrained arrays

The Introduction to Ada course[117] 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:

```
M : Measurements (1 .. 10);
--                ^^^^^^^
--  Bounds cannot be changed!
```

6.1.3 Constrained array types

Note that we could declare constrained array types. Let's rework the previous example:

Listing 3: measurement_defs.ads

```
package Measurement_Defs is

   type Measurements is
     array (1 .. 10) of Float;
   --       ^ Bounds are of known and fixed.

end Measurement_Defs;
```

Listing 4: show_measurements.adb

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

with Measurement_Defs; use Measurement_Defs;

```

(continues on next page)

117 https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-type-instance-bound

(continued from previous page)

```
procedure Show_Measurements is
   M : Measurements;
   --             ^ We cannot change the
   --               bounds here!
begin
   Put_Line ("First index: " & M'First'Image);
   Put_Line ("Last index:  " & M'Last'Image);
end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Constraints.Constrained_
↪Array_Type
MD5: 4741986fdf4dab731baa001b6e60c345
```

Build output

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

Runtime output

```
First index:  1
Last index:   10
```

In this case, the bounds of the `Measurements` type are fixed. Now, we cannot specify the bounds (or change them) in the declaration of the `M` array, as they have already been defined in the type declaration.

Unconstrained Arrays vs. Vectors

If you need, however, the flexibility of increasing the length of an array, you could use the language-defined `Vector` type instead. This is how we could rewrite the previous example using vectors:

Listing 5: measurement_defs.ads

```
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 6: show_measurements.adb

```
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);
```

(continues on next page)

(continued from previous page)

```
   --                  ^ Creating 10-element
   --                    vector.
begin
   Put_Line ("First index: "
             & M.First_Index'Image);
   Put_Line ("Last index:  "
             & M.Last_Index'Image);

   Put_Line ("Adding element...");
   M.Append (1.0);

   Put_Line ("First index: "
             & M.First_Index'Image);
   Put_Line ("Last index:  "
             & M.Last_Index'Image);
end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Constraints.Unconstrained_
↪Array_Type_Vs_Vector
MD5: afec7a4b898392be4dd1f60e1519da88
```

Runtime output

```
First index:  1
Last index:   10
Adding element...
First index:  1
Last index:   11
```

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[118]
- A.18.2 The Generic Package Containers.Vectors[119]

6.2 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

[118] http://www.ada-auth.org/standards/22rm/html/RM-3-6.html
[119] http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html

with a comma. The following example presents the one-dimensional array A1, the two-dimensional array A2 and the three-dimensional array A3:

Listing 7: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↪Multidimensional_Arrays
MD5: 928243b293c67a078d729c3cac68bb92
```

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 8: 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): "
```

(continues on next page)

(continued from previous page)

```
             & A3'Length (3)'Image);
end Show_Multidimensional_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↪Multidimensional_Arrays
MD5: 70b9b8df7e46302b92613fa484ef71ca
```

Runtime output

```
A1'Length:      10
A1'Length (1):  10
A2'Length (1):  5
A2'Length (2):  10
A3'Length (1):  2
A3'Length (2):  5
A3'Length (3):  10
```

As this simple example shows, we can easily retrieve the length of each dimension. Also, as we've just mentioned, 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:

Listing 9: measurement_defs.ads

```
package Measurement_Defs is

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

   type Hours is range 0 .. 11;

   subtype Measurement is Float;

   type Measurements is
     array (Days, Hours) of Measurement;

   procedure Show_Indices (M : Measurements);

   procedure Show_Values (M : Measurements);

   procedure Reset (M : out Measurements);

end Measurement_Defs;
```

Listing 10: measurement_defs.adb

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

package body Measurement_Defs is
```

(continues on next page)

(continued from previous page)

```

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

      for D in M'Range (1) loop
         Put (D'Image & " ");

         for H in M'First (2) ..
                  M'Last (2) - 1
         loop
            Put (H'Image & " ");
         end loop;
         Put_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 Measurement_Defs;
```

Listing 11: show_measurements.adb

```
with Measurement_Defs; use Measurement_Defs;

procedure Show_Measurements is
   M : Measurements;
begin
   Reset (M);
   Show_Indices (M);
   Show_Values (M);
end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↪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 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
```

We recommend that you spend some time analyzing this example. Also, we'd like to highlight the following aspects:

- We access a value from a multidimensional array by using commas to separate the index values within the parentheses. For example: `M (D, H)` allows us to access the value on day `D` and hour `H` from the multidimensional array `M`.
- To loop over the multidimensional array `M`, we write `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[120]
- 3.6.2 Operations of Array Types[121]

[120] http://www.ada-auth.org/standards/22rm/html/RM-3-6.html
[121] http://www.ada-auth.org/standards/22rm/html/RM-3-6-2.html

6.2.1 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 12: multidimensional_arrays_decl.ads

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↪Unconstrained_Multidimensional_Arrays
MD5: 8637e93db355fddafa3ffa5ce453a0e1
```

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 13: 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
   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

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↪Unconstrained_Multidimensional_Arrays
MD5: 9fb007abbfe238345d80cb315bb834c9
```

Build output

```
show_multidimensional_arrays.adb:7:04: warning: variable "A1" is read but never
↪assigned [-gnatwv]
show_multidimensional_arrays.adb:8:04: warning: variable "A2" is read but never
↪assigned [-gnatwv]
show_multidimensional_arrays.adb:9:04: warning: variable "A3" is read but never
↪assigned [-gnatwv]
```

Runtime output

```
A1'Length (1):  2
A2'Length (1):  4
A2'Length (2):  11
A3'Length (1):  2
A3'Length (2):  5
A3'Length (3):  2
```

6.2.2 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:

Listing 14: array_of_arrays_decl.ads

```
package Array_Of_Arrays_Decl is

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

   type T2 is
     array (Positive range <>) of T1 (1 .. 10);
   --                                ^^^^^^^
   --                            bounds must be set!

end Array_Of_Arrays_Decl;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Of_Arrays.Array_Of_Arrays
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:

Listing 15: measurement_defs.ads

```
package Measurement_Defs is

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

   type Hours is range 0 .. 11;

   subtype Measurement is Float;

   type Hourly_Measurements is
```

(continues on next page)

(continued from previous page)

```
      array (Hours) of Measurement;

   type Measurements is
     array (Days) of Hourly_Measurements;

   procedure Show_Indices (M : Measurements);

   procedure Show_Values (M : Measurements);

   procedure Reset (M : out Measurements);

end Measurement_Defs;
```

Listing 16: measurement_defs.adb

```
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
      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 (M'First)'Range loop
         H_IO.Put (H);
      end loop;
      New_Line;

      for D in M'Range loop
         Put (D'Image & " ");
```

(continues on next page)

(continued from previous page)

```

         for H in M (D)'Range loop
            M_IO.Put (M (D) (H));
            Put (" ");
         end loop;
         New_Line;
      end loop;
   end Show_Values;

   procedure Reset (M : out Measurements) is
   begin
      M := (others => (others => 0.0));
   end Reset;

end Measurement_Defs;
```

Listing 17: show_measurements.adb

```
with Measurement_Defs; use Measurement_Defs;

procedure Show_Measurements is
   M : Measurements;
begin
   Reset (M);
   Show_Indices (M);
   Show_Values (M);
end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Of_Arrays.Multidimensional_
↪Measurements
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 (D) (H)` allows us to access the value on day `D` and hour `H` from the array of arrays `M`.

- To loop over an array of arrays M, we write **for** D **in** M'Range **loop** for the first level of M and **for** H **in** M (D)'Range **loop** for the second level of M.
- Resetting all values of an array of arrays is very similar to how we do it for multidimensional arrays. In fact, we can still use an aggregate with this form: (**others** => (**others** => 0.0)).

6.3 Derived array types and array subtypes

6.3.1 Derived array types

As expected, we can derive from array types by declaring a new type. Let's see a couple of examples based on the Measurement_Defs package from previous sections:

Listing 18: measurement_defs.ads

```
package Measurement_Defs is

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

   --
   --  New array type:
   --
   type Measurements_Derived is
     new Measurements;

   --
   --  New array type with
   --  default component value:
   --
   type Measurements_Def30 is
     new Measurements
       with Default_Component_Value => 30.0;

   --
   --  New array type with constraints:
   --
   type Measurements_10 is
     new Measurements (1 .. 10);

end Measurement_Defs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.
↪Derived_Arrays
MD5: aefef9b9a844ad820d7f16546b8ffa64
```

In this example, we're deriving Measurements_Derived from the Measurements type. In the case of the Measurements_Def30 type, we're not only deriving from the Measurements type, but also setting the *default component value* (page 70) to 30.0. Finally, in the case of the Measurements_10, we're deriving from the Measurements type and *constraining the array type* (page 300) in the range from 1 to 10.

Let's use these types in a test application:

Listing 19: show_measurements.adb

```
with Measurement_Defs; use Measurement_Defs;

```

(continues on next page)

(continued from previous page)

```
procedure Show_Measurements is
   M1, M2  : Measurements (1 .. 10)
               := (others => 0.0);

   MD      : Measurements_Derived (1 .. 10);
   MD2     : Measurements_Derived (1 .. 40);
   MD10    : Measurements_10;
begin
   M1   := M2;
   --  ^^^^^^
   --  Assignment of arrays of
   --  same type.

   MD   := Measurements_Derived (M1);
   --       ^^^^^^^^^^^^^^^^^^^^^^^^^
   --  Conversion to derived type for
   --  the assignment.

   MD10 := Measurements_10 (M1);
   --       ^^^^^^^^^^^^^^^^^^^^
   --  Conversion to derived type for
   --  the assignment.

   MD10 := Measurements_10 (MD);
   MD10 := Measurements_10 (MD2 (1 .. 10));
end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.
↪Derived_Arrays
MD5: ce37a9c17eb9e1bb3931cca82852b54a
```

Build output

```
show_measurements.adb:8:04: warning: variable "MD2" is read but never assigned [-
↪gnatwv]
```

As illustrated by this example, we can assign objects of different array types, provided that we perform the appropriate type conversions and make sure that the bounds match.

6.3.2 Array subtypes

Naturally, we can also declare subtypes of array types. For example:

Listing 20: measurement_defs.ads

```
package Measurement_Defs is

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

   --
   --  Simple subtype declaration:
   --
   subtype Measurements_Sub is Measurements;

   --
   --  Subtype with constraints:
   --
```

(continues on next page)

(continued from previous page)

```
   subtype Measurements_10 is
     Measurements (1 .. 10);

   --
   --  Subtype with dynamic predicate
   --  (array can only have 20 components
   --  at most):
   --
   subtype Measurements_Max_20 is Measurements
       with Dynamic_Predicate =>
              Measurements_Max_20'Length <= 20;

   --
   --  Subtype with constraints and
   --  dynamic predicate (first element
   --  must be 2.0).
   --
   subtype Measurements_First_Two is
     Measurements (1 .. 10)
       with Dynamic_Predicate =>
              Measurements_First_Two (1) = 2.0;

end Measurement_Defs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.Array_
↪Subtypes
MD5: fa03c836111aa2df223a38a5d04d18bc
```

Here, we're declaring subtypes of the `Measurements` type. For example, `Measurements_Sub` is a *simple* subtype of `Measurements` type. In the case of the `Measurements_10` subtype, we're constraining the type to a range from 1 to 10.

For the `Measurements_Max_20` subtype, we're specifying — via a dynamic predicate — that arrays of this subtype can only have 20 components at most. Finally, for the `Measurements_First_Two` subtype, we're constraining the type to a range from 1 to 10 and requiring that the first component must have a value of 2.0.

Note that we cannot set the default component value for array subtypes — only type declarations are allowed to use that facility.

Let's use these subtypes in a test application:

Listing 21: show_measurements.adb

```
with Measurement_Defs; use Measurement_Defs;

procedure Show_Measurements is
   M1, M2  : Measurements (1 .. 10)
               := (others => 0.0);
   MS      : Measurements_Sub (1 .. 10);
   MD10    : Measurements_10;
   M_Max20 : Measurements_Max_20 (1 .. 40);
   M_F2    : Measurements_First_Two;
begin
   MS       := M1;
   MD10     := M1;

   M_Max20  := (others => 0.0);  --  ERROR!

   MD10 (1) := 4.0;
```

(continues on next page)

(continued from previous page)

```
   M_F2     := MD10;             --  ERROR!
end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.Array_
↪Subtypes
MD5: 003ddaab65d8c163302811abd7889745
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_
↪measurements.adb:14
```

As expected, assignments to objects with different subtypes — but with the same parent type — work fine without conversion. The assignment to `M_Max_20` fails because of the predicate failure: the predicate requires that the length be 20 at most, and it's 40 in this case. Also, the assignment to `M_F2` fails because the predicate requires that the first element must be set to `2.0`, and `MD10 (1)` has the value 4.0.

CHAPTER

SEVEN

STRINGS

7.1 Character and String Literals

So far, we're already seen many examples of string literals — both in the Introduction to Ada[122] course and in the present course. In this section, we define them once more and discuss a couple of details about them.

7.1.1 Character Literals

A character literal is simply a character between apostrophes (or *single quotation marks*). For example:

Listing 1: show_character_literals.adb

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

procedure Show_Character_Literals is
   C   : Character := 'a';
   --                 ^^^
   --           Character literal
begin
   Put_Line ("Character : " & C);
end Show_Character_Literals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.
↪Character_Literals
MD5: e9bf0dee97b4c6d52937316e7f285f48
```

Runtime output

```
Character : a
```

In this example, we initialize the character variable C with the character literal 'a'.

7.1.2 String Literals

A string literal is simply a collection of characters between quotation marks. For example:

Listing 2: show_simple_string_literals.adb

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

procedure Show_Simple_String_Literals is
```

(continues on next page)

122 https://learn.adacore.com/courses/intro-to-ada/index.html#intro-ada-course-index

(continued from previous page)

```
   S1 : String := "Hello";
   --             ^^^^^^^
   --         String literal

   S2 : String := "World";
   --             ^^^^^^^
   --           String literal
begin
   Put_Line (S1 & " " & S2);
end Show_Simple_String_Literals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.Simple_
↪String_Literals
MD5: a19bfa1ab6048f8cad9858d57b9f21e1
```

Runtime output

```
Hello World
```

In this example, "Hello" and "World" are string literals.

String literals with quotation

If you want to include a quotation mark in a string literal, you have to write "" (inside that string literal):

Listing 3: show_string_literals_with_quotes.adb

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

procedure Show_String_Literals_With_Quotes is
   S1 : String := "Hello";
   S2 : String := "World";
begin
   Put_Line ("  "" " & S1
   --           ^^
   --       Quotation marks
             & " " & S2 & " ""  ");
   --                       ^^
   --       Quotation marks

   Put_Line ("""Hello World!""");
   --         ^^            ^^
   --          Quotation marks

   Put_Line ("""""");
   --         ^^^^
   --    Quotation marks
end Show_String_Literals_With_Quotes;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.String_
↪Literals_With_Quotes
MD5: 97752e289b5f58f98920407f5dedc1fb
```

Runtime output

```
  " Hello World "
"Hello World!"
""
```

In this example, we display " Hello World " to the user by adding quotation marks to the concatenated strings in the call to Put_Line.

Note that the three quotation marks at the beginning of `"""Hello World!"""` consist of the quotation mark that indicate the beginning of the string literal and the two quotation marks that represent a single quotation mark inside the string literal. (The same thing happens at the end of this string literal, but in reverse.) This string literal is displayed as "Hello World!" to the user.

Finally, the string literal `""""""` is displayed as "" to the user.

Empty string literals

An empty string is represented by quotation marks without characters in between: `""`. For example:

Listing 4: show_empty_string_literals.adb

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

procedure Show_Empty_String_Literals is
   S1 : String          := "";
   S2 : String (1 .. 0) := "";
begin
   Put_Line (S1);
   Put_Line (S2);
   Put_Line ("");
end Show_Empty_String_Literals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.Empty_
↪String_Literals
MD5: f2f7c47784f1053665db9499cb6b53d8
```

Runtime output

```



```

Note that an empty string is an array of characters without any components. This is made explicit by the declaration of S2. Here, by using the range `1 .. 0`, we're declaring an empty array.

In other languages

In C, an empty string still contains a single character: the null character (\0). In Ada, however, an empty string doesn't have any characters.

In the Ada Reference Manual

- 2.5 Character Literals[123]
- 2.6 String Literals[124]

7.2 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:

```
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:

Character Type	Width
Character	8 bits
Wide_Character	16 bits
Wide_Wide_Character	32 bits

We can see that when running this example:

Listing 5: show_wide_char_types.adb

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

procedure Show_Wide_Char_Types is
begin
   Put_Line ("Character'Size:           "
             & Integer'Image
                 (Character'Size));
   Put_Line ("Wide_Character'Size:      "
             & Integer'Image
                 (Wide_Character'Size));
   Put_Line ("Wide_Wide_Character'Size: "
             & Integer'Image
                 (Wide_Wide_Character'Size));
end Show_Wide_Char_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Wide_Char_
 ↪Types
MD5: a0e9fb9e8d43e9fa707dc8c57f7562f8
```

[123] http://www.ada-auth.org/standards/22rm/html/RM-2-5.html
[124] http://www.ada-auth.org/standards/22rm/html/RM-2-6.html

Runtime output

```
Character'Size:           8
Wide_Character'Size:      16
Wide_Wide_Character'Size:  32
```

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

Listing 6: show_wide_string_types.adb

```ada
with Ada.Text_IO;
with Ada.Wide_Text_IO;
with Ada.Wide_Wide_Text_IO;

procedure Show_Wide_String_Types is
   package TI   renames Ada.Text_IO;
   package WTI  renames Ada.Wide_Text_IO;
   package WWTI renames Ada.Wide_Wide_Text_IO;

   S   : constant String           := "hello";
   WS  : constant Wide_String      := "hello";
   WWS : constant Wide_Wide_String := "hello";
begin
   TI.Put_Line ("String:            " & S);
   TI.Put_Line ("Length:            "
                & Integer'Image (S'Length));
   TI.Put_Line ("Size:              "
                & Integer'Image (S'Size));
   TI.Put_Line ("Component_Size:    "
                & Integer'Image
                    (S'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 ("Component_Size:    "
                & Integer'Image
                    (WWS'Component_Size));
   TI.Put_Line ("------------------------");
end Show_Wide_String_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Wide_
↪String_Types
MD5: 137816c6fd78add34287a72e45cf4fb7
```

Runtime output

```
String:           hello
Length:            5
Size:              40
Component_Size:    8
------------------------
Wide string:      hello
Length:            5
Size:              80
Component_Size:    16
------------------------
Wide-wide string: hello
Length:            5
Size:              160
Component_Size:    32
------------------------
```

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[125]

7.2.1 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:

[125] http://www.ada-auth.org/standards/22rm/html/RM-3-6-3.html

Scalar Type	Text I/O Packages
`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`:

Scalar Type	Text I/O Packages
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[126]
- A.10.1 The Package Text_IO[127]
- A.10.8 Input-Output for Integer Types[128]

- A.10.9 Input-Output for Real Types[129]
- A.11 Wide Text Input-Output and Wide Wide Text Input-Output[130]

7.2.2 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[131], 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,
- Ada.Strings.Wide_Wide_Maps.

Let's look at this example[132] from the Introduction to Ada course, which we adapted for wide-wide strings:

Listing 7: show_find_words.adb

```
with Ada.Strings; use Ada.Strings;

with Ada.Strings.Wide_Wide_Fixed;
use  Ada.Strings.Wide_Wide_Fixed;

with Ada.Strings.Wide_Wide_Maps;
use  Ada.Strings.Wide_Wide_Maps;

with Ada.Wide_Wide_Text_IO;
use  Ada.Wide_Wide_Text_IO;

procedure Show_Find_Words is

   S   : constant Wide_Wide_String :=
           "Hello" & 3 * " World";
   F   : Positive;
   L   : Natural;
   I   : Natural := 1;

   Whitespace : constant
     Wide_Wide_Character_Set :=
       To_Set (' ');
begin
   Put_Line ("String: " & S);
   Put_Line ("String length: "
             & Integer'Wide_Wide_Image
                 (S'Length));

   while I in S'Range loop
```

(continues on next page)

[126] http://www.ada-auth.org/standards/22rm/html/RM-A-10.html
[127] http://www.ada-auth.org/standards/22rm/html/RM-A-10-1.html
[128] http://www.ada-auth.org/standards/22rm/html/RM-A-10-8.html
[129] http://www.ada-auth.org/standards/22rm/html/RM-A-10-9.html
[130] http://www.ada-auth.org/standards/22rm/html/RM-A-11.html
[131] https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-string-operations
[132] https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-string-operations-show-find-words

(continued from previous page)

```
      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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Wide_Wide_
↪String_Handling
MD5: 3b5a4d61e6dc5bd16e85f85580ad82ae
```

Runtime output

```
String: Hello World World World
String length:  23
Found word instance at position  1: 'Hello'
Found word instance at position  7: 'World'
Found word instance at position  13: 'World'
Found word instance at position  19: 'World'
```

In this example, we're using the Find_Token procedure to find the words from the phrase stored in the S constant. All the operations we're using here are similar to the ones for **String** type, but making use of the Wide_Wide_String type instead. (We talk about the Wide_Wide_Image attribute *later on* (page 343).)

In the Ada Reference Manual

- A.4.6 String-Handling Sets and Mappings[133]
- A.4.7 Wide_String Handling[134]
- A.4.8 Wide_Wide_String Handling[135]

7.2.3 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[136] and unbounded strings[137] — those correspond to the Bounded_String and Unbounded_String types.

[133] http://www.ada-auth.org/standards/22rm/html/RM-A-4-6.html
[134] http://www.ada-auth.org/standards/22rm/html/RM-A-4-7.html
[135] http://www.ada-auth.org/standards/22rm/html/RM-A-4-8.html
[136] https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-bounded-strings
[137] https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-unbounded-strings

Those kinds of string types are available for **Wide_String**, and Wide_Wide_String. The following table shows the available types and corresponding packages:

Type	Package
Bounded_Wide_String	Ada.Strings.Wide_Bounded
Bounded_Wide_Wide_String	Ada.Strings.Wide_Wide_Bounded
Unbounded_Wide_String	Ada.Strings.Wide_Unbounded
Unbounded_Wide_Wide_String	Ada.Strings.Wide_Wide_Unbounded

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:

Type	Text I/O Package
Bounded_Wide_String	Ada.Wide_Text_IO.Wide_Bounded_IO
Bounded_Wide_Wide_String	Ada.Wide_Wide_Text_IO.Wide_Wide_Bounded_IO
Unbounded_Wide_String	Ada.Wide_Text_IO.Wide_Unbounded_IO
Unbounded_Wide_Wide_String	Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO

Let's look at a simple example:

Listing 8: show_unbounded_wide_wide_string.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Unbounded_
↪Wide_Wide_String
MD5: 0d369270e2408b3f1cc8284c13fca806
```

Runtime output

```
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[138]
- A.4.8 Wide_Wide_String Handling[139]
- A.11 Wide Text Input-Output and Wide Wide Text Input-Output[140]

7.3 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)[141] in various versions, which vary according to the underlying precision, support for backwards-compatibility and other requirements.

In the Ada Reference Manual

- A.4.11 String Encoding[142]

7.3.1 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:

Child Package of `Ada.Strings.UTF_Encoding`	Convert from / to
`.Strings`	**String** type
`.Wide_Strings`	**Wide_String** type
`.Wide_Wide_Strings`	`Wide_Wide_String` type

Let's look at an example:

Listing 9: show_ww_utf_string.adb

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

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

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

with Ada.Strings.Wide_Wide_Unbounded;
```

(continues on next page)

[138] http://www.ada-auth.org/standards/22rm/html/RM-A-4-7.html
[139] http://www.ada-auth.org/standards/22rm/html/RM-A-4-8.html
[140] http://www.ada-auth.org/standards/22rm/html/RM-A-11.html
[141] https://unicode.org/faq/utf_bom.html#gen2
[142] http://www.ada-auth.org/standards/22rm/html/RM-A-4-11.html

(continued from previous page)

```
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 := "مرحبا يا عالم";
   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;

end Show_WW_UTF_String;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WW_UTF_String
MD5: cecfb420bb804f42e7a65b793abcbef5
```

Runtime output

```
Wide_Wide_String Length:  26
Wide_Wide_String Size:    832
---------------------------------------
Converting Wide_Wide_String to UTF-8...
UTF-8 String:        Hello World: مرحبا يا عالم
UTF-8 String Length:  37
UTF-8 String Size:    296
```

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.

7.3.2 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:

String type	Size	Length
Wide_Wide_String	832	26
UTF_8_String	296	37

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:

Listing 10: show_utf_8.adb

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

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

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

procedure Show_UTF_8 is

```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8
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[143], not the standard "x" from the Basic Latin block[144].)

7.3.3 UTF-16 encoding and decoding

So far, we've discussed the UTF-8 encoding scheme. However, other encoding schemes exist and are supported as well. In fact, the `Ada.Strings.UTF_Encoding` package defines three encoding schemes:

```
type Encoding_Scheme is (UTF_8,
                         UTF_16BE,
                         UTF_16LE);
```

For example, instead of using UTF-8 encoding, we can use UTF-16 encoding — either in the big-endian or in the little-endian version. To convert between UTF-8 and UTF-16 encoding schemes, we can make use of the conversion functions from the `Ada.Strings.UTF_Encoding.Conversions` package.

To declare a UTF-16 encoded string, we can use one of the following data types:

- the 8-bit-character based `UTF_String` type, or
- the 16-bit-character based `UTF_16_Wide_String` type.

When using the 8-bit version, though, we have to specify the input and output schemes when converting between UTF-8 and UTF-16 encoding schemes.

143 https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols
144 https://en.wikipedia.org/wiki/Basic_Latin_(Unicode_block)

Let's see a code example that makes use of both `UTF_String` and `UTF_16_Wide_String` types:

Listing 11: show_utf16_types.adb

```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
   --            UTF_8_String conversion.

   Put_Line
     ("UTF_String / UTF_16BE: "
      & Convert
          (Item          => Symbols_UTF_16BE,
           Input_Scheme  => UTF_16BE,
           Output_Scheme => UTF_8));
end Show_UTF16_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_16_Types
MD5: 905e20e83a6199fdc91a6b15bb71bb01
```

Runtime output

```
UTF_8_String:          ♥♫
UTF_16_Wide_String:    ♥♫
UTF_String / UTF_16BE: ♥♫
```

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 12: show_ww_utf16_string.adb

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

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

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

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

procedure Show_WW_UTF16_String is
   Symbols_UTF_16 : constant
     UTF_16_Wide_String :=
       Wide_Character'Val (16#2665#) &
       Wide_Character'Val (16#266B#);
   --  ^ Calling Wide_Character'Val
   --    to specify the UTF-16 BE code
   --    for "♥" and "♫".

   Symbols_WWS : constant
     Wide_Wide_String :=
       Decode (Symbols_UTF_16);
   --  ^ Calling Decode for UTF_16_Wide_String
   --    to Wide_Wide_String conversion.
begin
   Put_Line ("UTF_16_Wide_String: "
             & Convert (Symbols_UTF_16));
   --          ^ Calling Convert for the
   --            UTF_16_Wide_String to
   --            UTF_8_String conversion.

   Put_Line ("Wide_Wide_String:   "
             & Encode (Symbols_WWS));
   --          ^ Calling Encode for the
   --            Wide_Wide_String to
   --            UTF_8_String conversion.
end Show_WW_UTF16_String;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WW_UTF_16_String
MD5: 900af8f5c6aad7303c3e49c1c4a68d73
```

Runtime output

```
UTF_16_Wide_String: ♥♫
Wide_Wide_String:   ♥♫
```

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

7.4 UTF-8 applications

In this section, we take a further look into UTF-8 encoding and some real-world applications. First, we discuss the use of UTF-8 encoding in source-code files. Then, we talk about parsing UTF-8 files using *wide-wide* strings.

7.4.1 UTF-8 encoding in source-code files

In the past, it was common to use different character sets in text files when writing in different (human) languages. By default, Ada source-code files are expected to use the Latin-1 coding, which is a 8-bit character set.

Nowadays, however, using UTF-8 coding for text files — including source-code files — is very common. If your Ada code only uses standard ASCII characters, but you're saving it in a UTF-8 coded file, there's no need to worry about character sets, as UTF-8 is backwards compatible with ASCII.

However, you might want to use Unicode symbols in your Ada source code to declare constants — as we did in the previous sections — and store the source code in a UTF-8 coded file. In this case, you need be careful about how this file is parsed by the compiler.

Let's look at this source-code example:

Listing 13: show_utf_8_strings.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_Strings
MD5: fd1aaff161a33365d15adca5bea7b277
```

Runtime output

```
UTF_8_String: ♥♫
Length:        6
```

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[145] to be encoded in UTF-8 format.) Also, in this case, the UTF-8 encoding process is using two additional bytes. Therefore, the total length of the string is six, which matches what we see when running the `Show_UTF_8_Strings` procedure. In other words, the length of the `Symbols_UTF_8` string doesn't refer to those two characters ("♥♫") that we were using in the constant declaration, but the length of the encoded bytes in its UTF-8 representation.

The UTF-8 format is very useful for storing and transmitting texts. However, if we want to process Unicode symbols, it's probably better to use string types with 32-bit characters — such as `Wide_Wide_String`. For example, let's say we want to use the "♥♫" string again to initialize a constant of `Wide_Wide_String` type:

Listing 14: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WWS_Strings_W8
MD5: 1e5e38e62b412de48d3fa4271bb48bf1
```

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

[145] https://en.wikipedia.org/wiki/UTF-8

Wide_Wide_String. For this case, GNAT offers the -gnatW8 switch. Let's run the previous example using this switch:

Listing 15: 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: 1e5e38e62b412de48d3fa4271bb48bf1
```

Runtime output

```
Wide_Wide_String: ♥♫
Length:           2
```

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 16: 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_No_W8
MD5: 1e5e38e62b412de48d3fa4271bb48bf1

Runtime output

```
Wide_Wide_String: ♥♫
Length:            6
```

Now, the `"♥♫"` string is being interpreted as a string of six 8-bit characters. (In other words, the UTF-8-coded string isn't converted to the UTF-32 format.) Each of those 8-bit characters is then stored in a 32-bit character of the `Wide_Wide_String` type. This explains why the `Show_WWS_Strings` procedure reports a length of 6 components for the `Symbols_WWS` string.

Portability of UTF-8 in source-code files

In a previous code example, we were assuming that the format that we use for the source-code file is UTF-8. This allows us to simply use Unicode symbols directly in strings:

```
Symbol_UTF_8 : constant UTF_8_String := "★";
```

This approach, however, might not be portable. For example, if the compiler uses a different string encoding for source-code files, it might interpret that Unicode character as something else — or just throw a compilation error.

If you're afraid that format mismatches might happen in your compilation environment, you may want to write strings in your code in a completely portable fashion, which consists in entering the exact sequence of codes in bytes — using the `Character'Val` function — for the symbols you want to use.

We can reuse parts of the previous example and replace the UTF-8 character with the corresponding UTF-8 code:

Listing 17: show_utf_8.adb

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

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

procedure Show_UTF_8 is

   Symbol_UTF_8 : constant
     UTF_8_String :=
       Character'Val (16#e2#)
       & Character'Val (16#98#)
       & Character'Val (16#85#);

begin
   Put_Line ("UTF-8 String: "
             & Symbol_UTF_8);
end Show_UTF_8;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8
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.

7.4.2 Parsing UTF-8 files for Wide-Wide-String processing

A typical use-case is to parse a text file in UTF-8 format and use *wide-wide* strings to process the lines of that file. Before we look at the implementation that does that, let's first write a procedure that generate a text file in UTF-8 format:

Listing 18: generate_utf_8_file.adb

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

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

procedure Generate_UTF_8_File
  (Output_File_Name : String)
is
   F : File_Type;
begin
   Create (F, Out_File, Output_File_Name);
   Put_Line (F, UTF_8_String'("♥♫"));
   Put_Line
     (F,
      UTF_8_String'("مرحبا يا عالم"));
   Close (F);
end Generate_UTF_8_File;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
↪Processing
MD5: 58c7591796bc1348796afa6db6f64d22
```

Procedure `Generate_UTF_8_File` writes two strings with non-Latin characters into the UTF-8 file indicated by the `Output_File_Name` parameter.

In addition, let's implement an auxiliary procedure to display the individual characters of a *wide-wide* string:

Listing 19: put_line_utf_8_characters.adb

```ada
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 Put_Line_UTF_8_Characters
  (WSS : Wide_Wide_String)
is
```

(continues on next page)

(continued from previous page)

```
   procedure Put_Complete_UTF_8_String
     (WSS : Wide_Wide_String)
   is
      S_UTF_8 : constant UTF_8_String :=
                  Encode (WSS);
   begin
      Put_Line ("STRING: " & S_UTF_8);
      Put_Line ("Length: "
                & WSS'Length'Image
                & " characters");
      New_Line;
   end Put_Complete_UTF_8_String;

   --  This is a wrapper function of the
   --  Encode function for the
   --  Wide_Wide_Character type:
   function Encode (Item : Wide_Wide_Character)
                    return UTF_8_String
    is
       SC : constant Wide_Wide_String (1 .. 1)
            := (1 => Item);
       --  We need a 1-character string
       --  for the call to Encode.
   begin
       return Encode (SC);
   end Encode;

   procedure Put_UTF_8_Characters
     (WSS : Wide_Wide_String) is
   begin
      for I in WSS'Range loop
         Put (I'Image & ": ");
         Put (Encode (WSS (I)));
         New_Line;
      end loop;
   end Put_UTF_8_Characters;

begin
    Put_Complete_UTF_8_String (WSS);
    Put_UTF_8_Characters (WSS);
    Put_Line ("--------------------");
end Put_Line_UTF_8_Characters;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
 ↪Processing
MD5: 14fae1f2b1d3795f3cef244f60082fcc
```

Finally, let's look at a code example that parses an UTF-8 file:

Listing 20: show_utf_8.adb

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

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

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

```

(continues on next page)

(continued from previous page)

```
with Generate_UTF_8_File;
with Put_Line_UTF_8_Characters;

procedure Show_UTF_8 is

   File_Name : constant String :=
                 "utf-8_test.txt";

   procedure Read_UTF_8_File
     (Input_File_Name : String)
   is
      F : File_Type;
   begin
      Open (F, In_File, Input_File_Name);

      while not End_Of_File (F) loop
         declare
            S_UTF8 : constant UTF_8_String
                       := Get_Line (F);
            S      : constant Wide_Wide_String
                       := Decode (S_UTF8);
         begin
            Put_Line_UTF_8_Characters (S);
         end;
      end loop;
      Close (F);
   end Read_UTF_8_File;

begin
   Generate_UTF_8_File (File_Name);
   Read_UTF_8_File (File_Name);
end Show_UTF_8;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
↪Processing
MD5: 512ad5ac7c6d5936735f017bfe629aa3
```

Runtime output

```
STRING: ♥♫
Length:  2 characters

 1: ♥
 2: ♫
--------------------
STRING: مرحبا يا عالم
Length:  13 characters

 1: م
 2: ر
 3: ح
 4: ب
 5: ا
 6:  
 7: ي
 8: ا
 9:  
 10: ع
 11: ا
```

(continues on next page)

(continued from previous page)

```
12: ل
13: م
--------------------
```

The Show_UTF_8 procedure first calls the Generate_UTF_8_File procedure to generate a text file in UTF-8 format, and then calls the nested Read_UTF_8_File procedure to read from that file — this is done by reading the 8-bit UTF-8 encoded string and decoding it into a string of Wide_Wide_String type.

(Note that we call the auxiliary Put_Line_UTF_8_Characters procedure to display the characters of each line we read from the UTF-8 file.)

For completeness, we include the nested Read_Write_UTF_8_File procedure, which not only reads each line from a UTF-8 file, but also writes it into another UTF-8 file:

Listing 21: show_utf_8.adb

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

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

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

with Generate_UTF_8_File;
with Put_Line_UTF_8_Characters;

procedure Show_UTF_8 is

   File_Name_In  : constant String :=
                     "utf-8_test.txt";
   File_Name_Out : constant String :=
                     "utf-8_copy.txt";

   procedure Read_Write_UTF_8_File
     (Input_File_Name,
      Output_File_Name : String)
   is
      F_In, F_Out : File_Type;
   begin
      Open (F_In, In_File, Input_File_Name);
      Create (F_Out, Out_File, Output_File_Name);

      while not End_Of_File (F_In) loop
         declare
            S : constant Wide_Wide_String :=
                  Decode (Get_Line (F_In));
         begin
            Put_Line_UTF_8_Characters (S);
            Put_Line (F_Out, Encode (S));
         end;
      end loop;

      Close (F_In);
      Close (F_Out);
   end Read_Write_UTF_8_File;

begin
   Generate_UTF_8_File (File_Name_In);

```

(continues on next page)

(continued from previous page)

```
   Read_Write_UTF_8_File
     (Input_File_Name  => File_Name_In,
      Output_File_Name => File_Name_Out);
end Show_UTF_8;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
↪Processing
MD5: 8cd13e8a565266fa5dd854ff6a34524c
```

Runtime output

```
STRING: ♥♫
Length:  2 characters

 1: ♥
 2: ♫
--------------------
STRING: مرحبا يا عالم
Length:  13 characters

 1: م
 2: ر
 3: ح
 4: ب
 5: ا
 6:  
 7: ي
 8: ا
 9:  
 10: ع
 11: ا
 12: ل
 13: م
--------------------
```

In the nested Read_Write_UTF_8_File procedure, we see both Decode and Encode functions being called to convert from and to the UTF_8_String type, respectively.

In the GNAT toolchain

If we use the -gnatW8 switch, which we mentioned *in a previous section* (page 332), the implementation of Generate_UTF_8_File and Put_Line_UTF_8_Characters must be adapted. In addition, we can simplify the implementation of the Show_UTF_8 procedure, too. (Note, however, that the previous implementation, which makes use of the Decode and Encode functions, would work fine as well.)

Listing 22: put_line_utf_8_characters.adb

```
with Ada.Wide_Wide_Text_IO;
use  Ada.Wide_Wide_Text_IO;

procedure Put_Line_UTF_8_Characters
  (WSS : Wide_Wide_String)
is
   procedure Put_Complete_UTF_8_String
     (WSS : Wide_Wide_String)
   is
   begin
      Put_Line ("STRING: " & WSS);
```

```
      Put_Line ("Length: "
                & WSS'Length'Wide_Wide_Image
                & " characters");
      New_Line;
   end Put_Complete_UTF_8_String;

   procedure Put_UTF_8_Characters
     (WSS : Wide_Wide_String)
   is
   begin
      for I in WSS'Range loop
         Put (I'Wide_Wide_Image & ": ");
         Put (WSS (I));
         New_Line;
      end loop;
   end Put_UTF_8_Characters;

begin
    Put_Complete_UTF_8_String (WSS);
    Put_UTF_8_Characters (WSS);
    Put_Line ("--------------------");
end Put_Line_UTF_8_Characters;
```

Listing 23: generate_utf_8_file.adb

```
with Ada.Wide_Wide_Text_IO;
use  Ada.Wide_Wide_Text_IO;

procedure Generate_UTF_8_File
  (Output_File_Name : String)
is
   F : File_Type;
begin
   Create (F, Out_File, Output_File_Name);
   Put_Line (F, "♥♫");
   Put_Line (F, "مرحبا يا عالم");
   Close (F);
end Generate_UTF_8_File;
```

Listing 24: show_utf_8.adb

```
with Ada.Wide_Wide_Text_IO;
use  Ada.Wide_Wide_Text_IO;

with Generate_UTF_8_File;
with Put_Line_UTF_8_Characters;

procedure Show_UTF_8 is

   File_Name_In  : constant String :=
                     "utf-8_test.txt";
   File_Name_Out : constant String :=
                     "utf-8_copy.txt";

   procedure Read_Write_UTF_8_File
     (Input_File_Name,
      Output_File_Name : String)
   is
      F_In, F_Out : File_Type;
   begin
      Open (F_In, In_File, Input_File_Name);
      Create (F_Out, Out_File, Output_File_Name);

```

```
      while not End_Of_File (F_In) loop
         declare
            S : constant Wide_Wide_String :=
                  Get_Line (F_In);
         begin
            Put_Line_UTF_8_Characters (S);
            Put_Line (F_Out, S);
         end;
      end loop;

      Close (F_In);
      Close (F_Out);
   end Read_Write_UTF_8_File;

begin
   Generate_UTF_8_File (File_Name_In);

   Read_Write_UTF_8_File
     (Input_File_Name  => File_Name_In,
      Output_File_Name => File_Name_Out);
end Show_UTF_8;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
↪Processing
MD5: 8eeed924f6d661a0a62ecb4d94be7027
```

Runtime output

```
STRING: ♥♫
Length:  2 characters

 1: ♥
 2: ♫
--------------------
STRING: مرحبا يا عالم
Length:  13 characters

 1: م
 2: ر
 3: ح
 4: ب
 5: ا
 6:  
 7: ي
 8: ا
 9:  
 10: ع
 11: ا
 12: ل
 13: م
--------------------
```

In this version of the code, we've removed all references to the UTF_8_String type — as well as the Decode and Encode functions that we were using to convert from and to this type. In this case, all UTF-8 processing happens directly using strings of Wide_Wide_Strings type.

7.5 Image attribute

7.5.1 Overview

In the Introduction to Ada[146] course, we've seen that the Image attribute returns a string that contains a textual representation of an object. For example, we write Integer'Image (V) to get a string for the integer variable V:

Listing 25: show_simple_image.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Simple_Image
MD5: e38f6f1a0808f12bd53c1f3cf4983353
```

Runtime output

```
V:  10
```

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

Listing 26: show_simple_image.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Simple_Image
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.

[146] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-image-attribute

In the Ada Reference Manual

- Image Attributes[147]

7.5.2 Type'Image and Obj'Image

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

Listing 27: show_simple_image.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Simple_Image
MD5: c8b2e458de47b403568dd795b3d3fc24
```

Runtime output

```
V:  10
```

In this example, the `Integer'Image (V)` and `V'Image` forms are equivalent.

7.5.3 Wider versions of Image

Although we've been talking only about the Image attribute, it's important to mention that each of the wider versions of the string types also has a corresponding Image attribute. In fact, this is the attribute for each string type:

Attribute	Type of Returned String
Image	String
Wide_Image	Wide_String
Wide_Wide_Image	Wide_Wide_String

Let's see a simple example:

Listing 28: show_wide_wide_image.adb

```
with Ada.Wide_Wide_Text_IO;
use  Ada.Wide_Wide_Text_IO;

procedure Show_Wide_Wide_Image is
   F : Float;
begin
   F := 100.0;
```

(continues on next page)

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

(continued from previous page)

```
   Put_Line ("F = "
             & F'Wide_Wide_Image);
end Show_Wide_Wide_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Wide_Wide_Image
MD5: ff542ef93286529343466c27935d5c21
```

Runtime output

```
F =  1.00000E+02
```

In this example, we use the `Wide_Wide_Image` attribute to retrieve a string of `Wide_Wide_String` type for the floating-point variable F.

7.5.4 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 29: simple_records.ads

```
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 30: show_non_scalar_image.adb

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

with Simple_Records;
use  Simple_Records;

```

(continues on next page)

(continued from previous page)

```
procedure Show_Non_Scalar_Image is

   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => Rec,
        Name   => Rec_Access);

   R_A : Rec_Access :=
     new Rec'(Init);

   N_R : Null_Rec :=
     (null record);
begin
   R_A := new Rec'(Init);
   N_R := (null record);

   Put_Line ("R_A:      " & R_A'Image);
   Put_Line ("R_A.all: " & R_A.all'Image);
   Put_Line ("N_R:      " & N_R'Image);

   Free (R_A);
   Put_Line ("R_A:      " & R_A'Image);
end Show_Non_Scalar_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Non_Scalar_Image
MD5: eb48f3fbe69b70258bc26f467918717c
```

Runtime output

```
R_A:     (access 288b12c0)
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[148] section of the Ada Reference Manual, as it was only applied to those types. Now, it is part of the new Image Attributes[149] section.

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

[148] http://www.ada-auth.org/standards/22rm/html/RM-3-5.html
[149] http://www.ada-auth.org/standards/22rm/html/RM-4-10.html

Listing 31: show_array_image.adb

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

procedure Show_Array_Image is

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

   FA_3C   : Float_Array (1 .. 3);
   FA_Null : Float_Array (1 .. 0);

begin
   FA_3C   := [1.0, 3.0, 2.0];
   FA_Null := [];

   Put_Line ("FA_3C:   " & FA_3C'Image);
   Put_Line ("FA_Null: " & FA_Null'Image);
end Show_Array_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Array_Image
MD5: a24daba1d92a139ae8995bba5a81e0d6
```

Runtime output

```
FA_3C:
[ 1.00000E+00,  3.00000E+00,  2.00000E+00]
FA_Null:
[]
```

In this example, we display the values of the three components of the FA_3C array. Also, we display the null array FA_Null.

7.5.5 Image attribute for tagged types

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

Listing 32: simple_records.ads

```
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;

```

(continues on next page)

(continued from previous page)

```
   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 33: show_tagged_image.adb

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

with Simple_Records; use Simple_Records;

procedure Show_Tagged_Image is
   R       : constant Rec       := Init;
   R_Class : constant Rec'Class := Rec'(Init);
   R_C     : constant Rec_Child := Init;
begin
   Put_Line ("R:       " & R'Image);
   Put_Line ("R_Class: " & R_Class'Image);
   Put_Line ("R_A:     " & R_C'Image);
end Show_Tagged_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Tagged_Image
MD5: 496827d5f81f8b7bec3b1d4a104f550e
```

Runtime output

```
R:
(F =>  1.00000E+01,
 I =>  4)
R_Class: SIMPLE_RECORDS.REC'
(F =>  1.00000E+01,
 I =>  4)
R_A:
(F =>  1.00000E+01,
 I =>  4,
 Z => OFF)
```

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.

7.5.6 Image attribute for task and protected types

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

Listing 34: simple_tasking.ads

```
package Simple_Tasking is

   protected type Protected_Float (I : Integer) is

```

(continues on next page)

(continued from previous page)

```
   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 35: simple_tasking.adb

```
package body Simple_Tasking is

   protected body Protected_Float is

   end Protected_Float;

   protected body Protected_Null is

   end Protected_Null;

   task body T is
   begin
      accept Start;
   end T;

end Simple_Tasking;
```

Listing 36: show_protected_task_image.adb

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

with Simple_Tasking; use Simple_Tasking;

procedure Show_Protected_Task_Image is

   PF : Protected_Float (0);
   PN : Protected_Null;
   T1 : T;

begin
   Put_Line ("PF: " & PF'Image);
   Put_Line ("PN: " & PN'Image);
   Put_Line ("T1: " & T1'Image);

   T1.Start;
end Show_Protected_Task_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Protected_Task_
↪Image
MD5: feb14f17ba1cca0311420272ef91ab38
```

Runtime output

```
PF: (protected object)
PN: (protected object)
T1: (task t1_0000000019262090)
```

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

7.6 Put_Image aspect

Note

This feature was introduced in Ada 2022.

7.6.1 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:

Listing 37: show_put_image.ads

```ada
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;
```

Listing 38: show_put_image.adb

```ada
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
```

(continues on next page)

(continued from previous page)

```
      --  information
      Buffer.Put ("<custom info>");
   end Put_Image_T;

end Show_Put_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Simple_Put_Image
MD5: 45c55444f0e1825312b5eafe307ca58d
```

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 356).)

In the Ada Reference Manual

- Image Attributes[150]

7.6.2 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 39: custom_numerics.ads

```
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;
```

150 http://www.ada-auth.org/standards/22rm/html/RM-4-10.html

Listing 40: custom_numerics.adb

```
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;
```

Listing 41: show_put_image.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Put_Image_Custom_
 ↪Numerics
MD5: 1dbb5fa612b5ca86facc3e93b47977e0
```

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.)

7.6.3 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
   --  buffer as a string
   return Buffer.Get;
end Image;
```

In other words, the Image attribute basically:

- calls the Put_Image procedure specified in the Put_Image aspect of type T's declaration and passes a buffer;

and

- retrieves the contents of the buffer as a string and returns it.

If the Put_Image aspect of type T isn't specified, the default version is used. (We've seen the default version of various types *in the previous section* (page 342) about the Image attribute.)

7.6.4 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 42: untagged_put_image.ads

```
with Ada.Strings.Text_Buffers;

package Untagged_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 Untagged_Put_Image;
```

Listing 43: untagged_put_image.adb

```
package body Untagged_Put_Image is

   procedure Put_Image_T
     (Buffer : in out Root_Buffer_Type'Class;
      Arg    :        T) is
      pragma Unreferenced (Arg);
   begin
      Buffer.Wide_Wide_Put ("Put_Image_T");
   end Put_Image_T;

   procedure Put_Image_T_Derived_2
     (Buffer : in out Root_Buffer_Type'Class;
      Arg    :        T_Derived_2) is
      pragma Unreferenced (Arg);
   begin
      Buffer.Wide_Wide_Put
        ("Put_Image_T_Derived_2");
   end Put_Image_T_Derived_2;

end Untagged_Put_Image;
```

Listing 44: show_untagged_put_image.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Untagged_Put_Image
MD5: acc0c17d45e6271cb582e65bfc8a2a98
```

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.

7.6.5 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 45: tagged_put_image.ads

```
with Ada.Strings.Text_Buffers;

package Tagged_Put_Image is

   use Ada.Strings.Text_Buffers;

   type T is tagged record
      I : Integer := 0;
   end record
     with Put_Image => Put_Image_T;

   procedure Put_Image_T
     (Buffer : in out Root_Buffer_Type'Class;
      Arg    :        T);

   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'Class;
      Arg    :        T_Child_3);

end Tagged_Put_Image;
```

Listing 46: tagged_put_image.adb

```
package body Tagged_Put_Image is

   procedure Put_Image_T
     (Buffer : in out Root_Buffer_Type'Class;
      Arg    :        T) is
      pragma Unreferenced (Arg);
   begin
      Buffer.Wide_Wide_Put ("Put_Image_T");
   end Put_Image_T;

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

(continues on next page)

(continued from previous page)

```
         ("Put_Image_T_Child_3");
   end Put_Image_T_Child_3;

end Tagged_Put_Image;
```

Listing 47: show_tagged_put_image.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Tagged_Put_Image
MD5: b19214bbcbc8c0339ead744afffcdd68
```

Runtime output

```
T'Image :         Put_Image_T
--------------------
T_Child_1'Image :
(Put_Image_T with I1 =>  0)
--------------------
T_Child_2'Image :
(Put_Image_T)
--------------------
T_Child_3'Image : Put_Image_T_Child_3
--------------------
T'Class'Image :   TAGGED_PUT_IMAGE.T_CHILD_1'
(Put_Image_T with I1 =>  0)
```

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.

7.7 Universal text buffer

In the *previous section* (page 349), 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.

7.7.1 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[151]

7.7.2 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.

[151] http://www.ada-auth.org/standards/22rm/html/RM-A-4-12.html

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 48: custom_numerics.ads

```
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 49: custom_numerics.adb

```
package body Custom_Numerics is

   procedure Put_Float_Integer
     (Buffer : in out Root_Buffer_Type'Class;
      Arg    :        Float_Integer) is
   begin
      Buffer.Wide_Wide_Put ("(");
      Buffer.New_Line;

      Buffer.Increase_Indent;

      Buffer.Wide_Wide_Put
        ("F : "
         & Arg.F'Wide_Wide_Image);
      Buffer.New_Line;

      Buffer.Wide_Wide_Put
        ("I : "
        & Arg.I'Wide_Wide_Image);

      Buffer.Decrease_Indent;
      Buffer.New_Line;

      Buffer.Wide_Wide_Put (")");
   end Put_Float_Integer;

end Custom_Numerics;
```

Listing 50: show_put_image.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Universal_Text_Buffer.Put_Image_
↪Custom_Numerics
MD5: e976a2ade2ad4a10033924e19bc84159
```

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.

CHAPTER

EIGHT

NUMERICS

8.1 Numeric Literals

8.1.1 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 1: real_integer_literals.adb

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

procedure Real_Integer_Literals is
   Integer_Literal : constant := 365;
   Real_Literal    : constant := 365.2564;
begin
   Put_Line ("Integer Literal: "
             & Integer_Literal'Image);
   Put_Line ("Real Literal:    "
             & Real_Literal'Image);
end Real_Integer_Literals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Real_Integer_
↪Literals
MD5: ba1cc348cad054f3ab86c05e051b40fa
```

Runtime output

```
Integer Literal:  365
Real Literal:     3.65256400000000000E+02
```

Another classification takes the use of a base indicator into account. (Remember that, when writing a literal such as 2#1011#, the base is the element before the first # sign.) So here we distinguish between decimal literals and based literals. For example:

Listing 2: decimal_based_literals.adb

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

procedure Decimal_Based_Literals is

   package F_IO is new
     Ada.Text_IO.Float_IO (Float);

```

(continues on next page)

(continued from previous page)

```
   --
   --  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);

   Put ("Dec_Real:           ");
   F_IO.Put (Item => Dec_Real);
   New_Line;

   Put ("Dec_Real_Exp:       ");
   F_IO.Put (Item => Dec_Real_Exp);
   New_Line;

   Put_Line ("Based_Integer:     "
             & Based_Integer'Image);
   Put_Line ("Based_Integer_Exp: "
             & Based_Integer_Exp'Image);

   Put ("Based_Real:         ");
   F_IO.Put (Item => Based_Real);
   New_Line;

   Put ("Based_Real_Exp:     ");
   F_IO.Put (Item => Based_Real_Exp);
   New_Line;
end Decimal_Based_Literals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Decimal_Based_
↪Literals
MD5: bde8f422c3844826819348d18fb48a33
```

Runtime output

```
Dec_Integer:        365
Dec_Real:           365.2564
Dec_Real_Exp:       365.2564
```

(continues on next page)

(continued from previous page)

```
Based_Integer:      365
Based_Integer_Exp:  365
Based_Real:         365.2564
Based_Real_Exp:     365.2564
```

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.

8.1.2 Features and Flexibility

Note

This section was originally written by Franco Gasperoni and published as Gem #7: The Beauty of Numeric Literals in Ada[152].

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:

Listing 3: ada_numeric_literals.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Pi_Literals
MD5: 8f6516730fa98f08234b159488431aaf
```

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:

[152] https://www.adacore.com/gems/ada-gem-7

Listing 4: ada_numeric_literals.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Based_Literals
MD5: 0959ec5e4aafcde245c5a15597ac9b7e
```

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:

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

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

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

Here's the complete example:

Listing 5: ada_numeric_literals.adb

```
with Ada.Text_IO;

procedure Ada_Numeric_Literals is
   Lights_On  : constant := 2#1000_1000#;
   Lights_Off : constant := 2#0111_0111#;

   type Byte is mod 256;
   Output_Devices : Byte := 0;

   --  for Output_Devices'Address
   --    use 16#DEAD_BEEF#;
   --  ^^^^^^^^^^^^^^^^^^^^^^^^^^
   --  Memory mapped Output

   use Ada.Text_IO;
begin
   Output_Devices := Output_Devices or
                     Lights_On;

   Put_Line ("Output_Devices (lights on ) = "
             & Byte'Image (Output_Devices));

   Output_Devices := Output_Devices and
                     Lights_Off;

   Put_Line ("Output_Devices (lights off) = "
             & Byte'Image (Output_Devices));
end Ada_Numeric_Literals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literal_Lights
MD5: c3e72b25366d8d815a1f425f2695ad0b
```

Runtime output

```
Output_Devices (lights on ) =  136
Output_Devices (lights off) =  0
```

Of course, we can also use *records with representation clauses* (page 103) to do the above, which is even more elegant.

The notion of base in Ada allows for exponents, which is particularly pleasant. For instance, we can write:

Listing 6: literal_binaries.ads

```
package Literal_Binaries is
   Kilobyte  : constant := 2#1#e+10;
   Megabyte  : constant := 2#1#e+20;
   Gigabyte  : constant := 2#1#e+30;
   Terabyte  : constant := 2#1#e+40;
   Petabyte  : constant := 2#1#e+50;
   Exabyte   : constant := 2#1#e+60;
   Zettabyte : constant := 2#1#e+70;
   Yottabyte : constant := 2#1#e+80;
end Literal_Binaries;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literal_Binary
MD5: 98d971e0f170db570069f8868e442d6d
```

In based literals, the exponent — like the base — uses the regular decimal notation and specifies the power of the base that the based literal should be multiplied with to obtain the final value. For instance `2#1#e+10` = 1 x 2^{10} = `1_024` (in base 10), whereas `16#F#e+2` = 15 x 16^{2} = 15 x 256 = `3_840` (in base 10).

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

```
One_Third : constant := 3#0.1#;
--                      ^^^^^^
--                  same as 1.0/3
```

Whether we write `3#0.1#` or `1.0 / 3`, or even `3#1.0#e-1`, Ada allows us to specify exactly rational numbers for which decimal literals cannot be written.

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 :=
  0.33333333333333333333333333333;
Zero_Approx      : constant :=
  1.0 - 3.0 * One_Third_Approx;
```

where `Zero_Approx` is really `1.0e-29` — and that will show up in your numerical computations. The above is quite handy when we want to write fractions without any loss of precision. Here's the complete code:

Listing 7: 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 :=
     0.33333333333333333333333333333;
   Zero_Approx      : constant :=
     1.0 - 3.0 * One_Third_Approx;
```

(continues on next page)

(continued from previous page)

```

   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

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literals
MD5: ee604245b34e8cb878a8ebdb21cd564e
```

Runtime output

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

Along these same lines, we can write:

Listing 8: ada_numeric_literals.adb

```ada
with Ada.Text_IO;

with Literal_Binaries; use Literal_Binaries;

procedure Ada_Numeric_Literals is

   Big_Sum : constant := 1         +
                         Kilobyte  +
                         Megabyte  +
                         Gigabyte  +
                         Terabyte  +
                         Petabyte  +
                         Exabyte   +
                         Zettabyte;

   Result : constant := (Yottabyte - 1) /
                        (Kilobyte - 1);

   Nil    : constant := Result - Big_Sum;
   pragma Assert (Nil = 0);

   use Ada.Text_IO;

begin
   Put_Line ("Nil         = "
             & Integer'Image (Nil));
end Ada_Numeric_Literals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literal_Binary
MD5: 7bda6442e68271d12bdb827b63f0d702
```

Runtime output

```
Nil         =  0
```

and be guaranteed that Nil is equal to zero.

8.2 Universal Numeric Types

Previously, we introduced the concept of *universal types* (page 32). Three of them are numeric types: universal real, universal integer and universal fixed types. In this section, we discuss these universal numeric types in more detail.

8.2.1 Universal Real and Integer

Universal real and integer types are mainly used in the declaration of *named numbers* (page 13):

Listing 9: show_universal_real_integer.ads

```
package Show_Universal_Real_Integer is

   Pi : constant := 3.1415926535;
   --               ^^^^^^^^^^^^
   --            universal real type

   N  : constant := 10;
   --               ^^
   --        universal integer type

end Show_Universal_Real_Integer;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Universal_Real_Integer
MD5: 3cfa52af66185c693ede07f3b0e689e6
```

The type of a named number is implied by the type of the *numeric literal* (page 359) and the type of any named numbers that we use in the *static expression* (page 366). (We discuss static expressions next.) In this specific example, we declare `Pi` using a real literal, which implies that it's a named number of universal real type. Likewise, `N` is of universal integer type because we use an integer literal in its declaration.

In the Ada Reference Manual

- 3.3.2 Number Declarations[153]

Static expressions

As we've just seen, we can use an expression in the declaration of a named number. This expression is static, as it's always evaluated at compile time. Therefore, we must use the keyword **constant** in the declaration of named numbers.

If all components of the static expression are of universal integer type, then the named number is of universal integer type. Otherwise, the static expression is of universal real type. For example, if the first element of a static expression is of universal integer type, but we have a constant of universal real type in the same expression, then the type of the whole static expression is universal real:

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

Listing 10: static_expressions.ads

```
package Static_Expressions is

   Two_Pi : constant := 2 * 3.1415926535;
   --                   ^
   --              universal integer type
   --
   --                       ^^^^^^^^^^^^
   --                 universal real type
   --
   --      => result: universal real type

end Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_
↪Expressions
MD5: 3429db9e1a7c4d4fe7d94e82159c3cb8
```

In this example, the static expression is of universal real type because of the real literal (3.1415926535) — even though we have the universal integer 2 in the expression.

Likewise, if we use a constant of universal real type in the static expression, the result is of universal real type:

Listing 11: static_expressions.ads

```
package Static_Expressions is

   Pi     : constant := 3.1415926535;
   --                   ^^^^^^^^^^^^
   --               universal real type

   Two_Pi : constant := 2 * Pi;
   --                   ^
   --              universal integer type
   --
   --                       ^^
   --                 universal real type
   --
   --      => result: universal real type

end Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_
↪Expressions
MD5: 599494102c4e5e5979a6e0071412da78
```

In this example, the result of the static expression is of universal real type because of we're using the named number Pi, which is of universal real type.

Complexity of static expressions

The operations that we use in static expressions may be arbitrarily complex. For example:

Listing 12: static_expressions.ads

```
package Static_Expressions is

   C1 : constant := 300_453.5;
   C2 : constant := 455_233.5 * C1;
   C3 : constant := 872_922.5 * C2;
   C4 : constant := 155_277.5 * C1 + C2 / C3;
   C5 : constant := 2.0 * C1 +
                    3.0 * (C2 / (C4 * C3)) +
                    4.0 * (C1 / (C2 * C2)) +
                    5.0 * (C3 / (C1 * C1));

end Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_
↪Expressions
MD5: ebdd5b1c64ad1944931a962756e72291
```

As we can see in this example, we may create a chain of dependencies, where the result of a static expression depends on the result of previously evaluated static expressions. For instance, C5 depends on the evaluation of C1, C2, C3, C4.

Accuracy of static expressions

The accuracy and range of numeric literals used in static expressions may be arbitrarily high as well:

Listing 13: static_expressions.ads

```
package Static_Expressions is

   Pi : constant :=
      3.14159_26535_89793_23846_26433_83279_50288;

   Seed : constant :=
      143_574_786_272_784_656_928_283_872_972_764;

   Super_Seed : constant :=
      Seed * Seed * Seed * Seed * Seed * Seed;

end Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_
↪Expressions
MD5: 777574a29ffa6da8bffb4287dee45be8
```

In this example, Super_Seed has a value that is above the typical range of integer constants. This might become challenging when using such named numbers in actual computations, as we *discuss soon* (page 372).

Another example is when the result of the expression is a repeating decimal[154]:

[154] https://en.wikipedia.org/wiki/Repeating_decimal

Listing 14: repeating_decimals.ads

```
package Repeating_Decimals is

   One_Over_Three : constant :=
      1.0 / 3.0;

end Repeating_Decimals;
```

Listing 15: show_repeating_decimals.adb

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

with Repeating_Decimals;
use  Repeating_Decimals;

procedure Show_Repeating_Decimals is
   F_1_3    : constant Float           :=
                One_Over_Three;
   LF_1_3   : constant Long_Float      :=
                One_Over_Three;
   LLF_1_3  : constant Long_Long_Float :=
                One_Over_Three;
begin
   Put_Line (F_1_3'Image);
   Put_Line (LF_1_3'Image);
   Put_Line (LLF_1_3'Image);
end Show_Repeating_Decimals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Repeating_Decimal
MD5: 4fc38ef6482e403d655b4662d4199abb
```

Runtime output

```
 3.33333E-01
 3.33333333333333E-01
 3.33333333333333333E-01
```

In this example, as expected, we see that the accuracy of the value we display increases if we use a type with higher precision. This wouldn't be possible if we had used a floating-point type with limited precision for the One_Over_Three constant:

Listing 16: repeating_decimals.ads

```
package Repeating_Decimals is

   One_Over_Three : constant Long_Float :=
      1.0 / 3.0;
   --                        ^^^^^^^^^^
   --          using Long_Float instead of
   --              universal real type

end Repeating_Decimals;
```

Listing 17: show_repeating_decimals.adb

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

```

(continues on next page)

(continued from previous page)

```
with Repeating_Decimals;
use  Repeating_Decimals;

procedure Show_Repeating_Decimals is
   F_1_3    : constant Float           :=
                Float (One_Over_Three);
   LF_1_3   : constant Long_Float      :=
                Long_Float (One_Over_Three);
   LLF_1_3  : constant Long_Long_Float :=
                Long_Long_Float (One_Over_Three);
begin
   Put_Line (F_1_3'Image);
   Put_Line (LF_1_3'Image);
   Put_Line (LLF_1_3'Image);
end Show_Repeating_Decimals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Repeating_Decimal
MD5: d0fa105d679cc246e2e8baf37cbe48c4
```

Runtime output

```
3.33333E-01
3.33333333333333E-01
3.33333333333333315E-01
```

Because we're using the **Long_Float** type for the One_Over_Three constant instead of the universal real type, the accuracy doesn't increase when we use the **Long_Long_Float** type — as we see in the value of the LLF_1_3 constant — even though this type has a higher precision.

For further reading...

When using *big numbers* (page 407), you could simply assign the named number One_Over_Three to a big real:

Listing 18: repeating_decimals.ads

```
package Repeating_Decimals is

   One_Over_Three : constant :=
      1.0 / 3.0;

end Repeating_Decimals;
```

Listing 19: show_repeating_decimals.adb

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

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

with Repeating_Decimals;
use  Repeating_Decimals;

procedure Show_Repeating_Decimals is
   BR_1_3 : constant Big_Real := One_Over_Three;
begin
   Put_Line ("BR: "
             & To_String (Arg   => BR_1_3,
                          Fore  => 2,
                          Aft   => 31,
                          Exp   => 0));
end Show_Repeating_Decimals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Repeating_Decimal
MD5: 4f1981b785baa35704c85e7e688c8ce4
```

Runtime output

```
BR:  0.3333333333333333333333333333333
```

Another approach is to use the division operation directly:

Listing 20: show_repeating_decimals.adb

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

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

with Repeating_Decimals;
use  Repeating_Decimals;

procedure Show_Repeating_Decimals is
   BR_1_3   : constant Big_Real := 1 / 3;
begin
   Put_Line ("BR: "
             & To_String (Arg   => BR_1_3,
                          Fore  => 2,
                          Aft   => 31,
                          Exp   => 0));
end Show_Repeating_Decimals;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Repeating_Decimal
MD5: 5fc195f9fbab3b1ec74c507780a44ec8
```

Runtime output

```
BR:  0.3333333333333333333333333333333
```

We talk more about *big real and quotients* (page 421) later on.

Conversion of universal real and integer

Although a named number exists as an numeric representation form in Ada, the value it represents cannot be used directly at runtime — even if we *just* display the value of the constant at runtime, for example. In fact, a conversion to a non-universal type is required in order to use the named number anywhere else other than a static expression:

Listing 21: static_expressions.ads

```
package Static_Expressions is

   Pi : constant :=
      3.14159_26535_89793_23846_26433_83279_50288;

   Seed : constant :=
      143_574_786_272_784_656_928_283_872_972_764;

   Super_Seed : constant :=
      Seed * Seed * Seed * Seed * Seed * Seed;

end Static_Expressions;
```

Listing 22: show_static_expressions.adb

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

with Static_Expressions;
use  Static_Expressions;

procedure Show_Static_Expressions is
begin
   Put_Line (Pi'Image);
   --  Same as:
   --  Put_Line (Float (Pi)'Image);

   Put_Line (Seed'Image);
   --  Same as:
   --  Put_Line (
   --     Long_Long_Long_Integer (Seed)'Image);
end Show_Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Conversion_To_Non_Universal_Types
MD5: e50641737f970b935e853ac249dd83d8
```

Runtime output

```
 3.14159265358979324E+00
 143574786272784656928283872972764
```

As we see in this example, the named number Pi is converted to **Float** before being used as an actual parameter in the call to Put_Line. Similarly, Seed is converted to Long_Long_Long_Integer.

When we use the Image attribute, the compiler automatically selects a numeric type which has a suitable range for the named number. In the example above, we wouldn't be able to represent the value of Seed with **Integer**, so the compiler selected Long_Long_Long_Integer. Of course, we could have also specified the type by using explicit *type conversions* (page 49) or a *qualified expressions* (page 68):

Listing 23: show_static_expressions.adb

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

with Static_Expressions;
use  Static_Expressions;

procedure Show_Static_Expressions is
begin
   Put_Line (Long_Long_Float (Pi)'Image);
   Put_Line (Long_Long_Float'(Pi)'Image);
end Show_Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Conversion_To_Non_Universal_Types
MD5: 18bcc3bffd51ebc1bc98976ed1597f01
```

Runtime output

```
 3.14159265358979324E+00
 3.14159265358979324E+00
```

Now, we're explicitly converting to **Long_Long_Float** in the first call to Put_Line and using a qualified expression in the second call to Put_Line.

A conversion is also performed when we use a named number in an object declaration:

Listing 24: show_static_expressions.adb

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

with Static_Expressions;
use  Static_Expressions;

procedure Show_Static_Expressions is
   Two_Pi : constant Float := 2.0 * Pi;
   --  Same as:
   --  Two_Pi: constant Float :=
   --            2.0 * Float (Pi);

   Two_Pi_More_Precise :
     constant Long_Long_Float := 2.0 * Pi;
   --  Same as:
   --  Two_Pi_More_Precise :
   --    constant Long_Long_Float :=
   --      2.0 * Long_Long_Float (Pi);
begin
   Put_Line (Two_Pi'Image);
   Put_Line (Two_Pi_More_Precise'Image);
end Show_Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Conversion_To_Non_Universal_Types
MD5: c918cdcb4e927cfbc1fbe6ffb0277178
```

Runtime output

```
6.28319E+00
6.28318530717958648E+00
```

In this example, Pi is converted to **Float** in the declaration of Two_Pi because we use the **Float** type in its declaration. Likewise, Pi is converted to **Long_Long_Float** in the declaration of Two_Pi_More_Precise because we use the **Long_Long_Float** type in its declaration. (Actually, the same conversion is performed for each instance of the real literal 2.0 in this example.)

Note that the range of the type we select might not be suitable for the named number we want to use. For example:

Listing 25: show_static_expressions.adb

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

with Static_Expressions;
use  Static_Expressions;

procedure Show_Static_Expressions is
   Initial_Seed : constant
     Long_Long_Long_Integer :=
       Super_Seed;
begin
   Put_Line (Initial_Seed'Image);
end Show_Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Conversion_To_Non_Universal_Types
MD5: 2f8e26fbcd0b5defd94ffef570c0f087
```

Build output

```
show_static_expressions.adb:9:08: error: value not in range of type "Standard.Long_
↪Long_Long_Integer"
show_static_expressions.adb:9:08: error: static expression fails Constraint_Check
gprbuild: *** compilation phase failed
```

In this example, we get a compilation error because the range of the Long_Long_Long_Integer type isn't enough to store the value of the Super_Seed.

For further reading...

To circumvent the compilation error in the code example we've just seen, the best alternative to use *big numbers* (page 407) — we discuss this topic later on in this chapter:

Listing 26: show_static_expressions.adb

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

with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;

with Static_Expressions;
use  Static_Expressions;

procedure Show_Static_Expressions is
   Initial_Seed : constant
     Big_Integer :=
       Super_Seed;
```

```
begin
   Put_Line (Initial_Seed'Image);
end Show_Static_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↪Conversion_To_Non_Universal_Types
MD5: bf1511f1b8bf3965baa86b953c56c406
```

Runtime output

```
␣
↪875929334140942197322254642884166058556995648205079468601338759565501436645446643560818669877
```

By changing the type from Long_Long_Long_Integer to Big_Integer, we get rid of the compilation error. (The value of Super_Seed — stored in Initial_Seed — is displayed at runtime.)

8.2.2 Universal Fixed

For fixed-point types, we also have a corresponding universal type. However, in contrast to the universal real and integer types, universal fixed types aren't an abstraction used in static expressions, but rather a concept that permeates actual fixed-point types. In fact, for *fixed-point types* (page 399), some operations are accomplished via universal fixed types — for example, the conversion between fixed-point types and the multiplication and division operations.

Let's start by analyzing how floating-point and integer types associate their operations to the specific type of an object. For example, if we have an object A of type **Float** in a multiplication, we cannot just write A * B if we want to multiply A by an object B of another floating-point type — if B is of type **Long_Float**, for example, writing A * B triggers a compilation error. (Otherwise, which precision should be used for the result?) Therefore, we have to convert one of the objects to have matching types:

Listing 27: show_float_multiplication_mismatch.adb

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

procedure Show_Float_Multiplication_Mismatch is
   F  : Float      := 0.25;
   LF : Long_Float := 0.50;
begin
   F := F * LF;
   Put_Line ("F = " & F'Image);
end Show_Float_Multiplication_Mismatch;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Float_
↪Multiplication
MD5: 88ce3a0f29e2bd31ddfc491557d7f0e3
```

Build output

```
show_float_multiplication_mismatch.adb:7:11: error: invalid operand types for␣
↪operator "*"
show_float_multiplication_mismatch.adb:7:11: error: left operand has type
↪"Standard.Float"
show_float_multiplication_mismatch.adb:7:11: error: right operand has type
↪"Standard.Long_Float"
gprbuild: *** compilation phase failed
```

This code example fails to compile because of the F * LF operation. (We could correct the code by writing F * Float (LF), for example.)

In contrast, for fixed-point types, we can mix objects of different types in a multiplication or division. (In this case, mixing is allowed for the convenience of the programmer.) For example:

Listing 28: normalized_fixed_point_types.ads

```ada
package Normalized_Fixed_Point_Types is

   type TQ31 is
     delta 2.0 ** (-31)
     range -1.0 .. 1.0 - 2.0 ** (-31);

   type TQ15 is
     delta 2.0 ** (-15)
     range -1.0 .. 1.0 - 2.0 ** (-15);

end Normalized_Fixed_Point_Types;
```

Listing 29: show_fixed_multiplication.adb

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

with Normalized_Fixed_Point_Types;
use  Normalized_Fixed_Point_Types;

procedure Show_Fixed_Multiplication is
   A : TQ15 := 0.25;
   B : TQ31 := 0.50;
begin
   A := A * B;
   Put_Line ("A = " & A'Image);
end Show_Fixed_Multiplication;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Fixed_Point_
↪Multiplication
MD5: a4cefdc29a562fbec30b6864b6ec2602
```

Runtime output

```
A =  0.12500
```

In this example, the A * B is accepted by the compiler, even though A and B have different types. This is only possible because the multiplication operation of fixed-point types makes use of the universal fixed type. In other words, the multiplication operation in this code example doesn't operate directly on the fixed-point type TQ31. Instead, it converts A and B to the universal fixed type, performs the operation using this type, and converts back to the original type — TQ15 in this case.

In addition to the multiplication operation, other operations such as the conversion between fixed-point types and the division operations make use of universal fixed types:

Listing 30: custom_decimal_types.ads

```ada
package Custom_Decimal_Types is

  type T3_D3 is delta 10.0 ** (-3) digits 3;
  type T3_D6 is delta 10.0 ** (-3) digits 6;
```

(continues on next page)

(continued from previous page)

```
  type T6_D6 is delta 10.0 ** (-6) digits 6;

end Custom_Decimal_Types;
```

Listing 31: show_universal_fixed.adb

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

with Custom_Decimal_Types;
use  Custom_Decimal_Types;

procedure Show_Universal_Fixed is
  Val_T3_D3 : T3_D3;
  Val_T3_D6 : T3_D6;
  Val_T6_D6 : T6_D6;
begin
   Val_T3_D3 := 0.65;

   Val_T3_D6 := T3_D6 (Val_T3_D3);
   --             ^^^^^^^^^^^^^^^^^
   --        type conversion using
   --         universal fixed type

   Val_T6_D6 := T6_D6 (Val_T3_D6);
   --             ^^^^^^^^^^^^^^^^^
   --        type conversion using
   --         universal fixed type

   Put_Line ("Val_T3_D3 = "
             & Val_T3_D3'Image);
   Put_Line ("Val_T3_D6 = "
             & Val_T3_D6'Image);
   Put_Line ("Val_T6_D6 = "
             & Val_T3_D6'Image);
   Put_Line ("-----------------");

   Val_T3_D6 := Val_T6_D6 * 2.0;
   --             ^^^^^^^^^^^^^^^^
   --      using universal fixed type for
   --        the multiplication operation
   Put_Line ("Val_T3_D6 = "
             & Val_T3_D6'Image);

   Val_T3_D6 := Val_T6_D6 / Val_T3_D3;
   --             ^^^^^^^^^^^^^^^^^^^^^
   --        different fixed-point types:
   --      using universal fixed type for
   --             the division operation
   Put_Line ("Val_T3_D6 = "
             & Val_T3_D6'Image);

end Show_Universal_Fixed;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Universal_Fixed
MD5: 1e253d8a39576f817b2130aa35929d96
```

Runtime output

```
Val_T3_D3 =  0.650
Val_T3_D6 =  0.650
Val_T6_D6 =  0.650
-----------------
Val_T3_D6 =  1.300
Val_T3_D6 =  1.000
```

In this example, the conversion from the fixed-point type T3_D3 to the T3_D6 and T6_D6 types is performed via universal fixed types.

Similarly, the multiplication operation `Val_T6_D6 * 2.0` uses universal fixed types. Here, we're actually multiplying a variable of type T6_D6 by two and assigning it to a variable of type Val_T3_D6. Although these variable have different fixed-point types, no explicit conversion (e.g.: `Val_T3_D6 := T3_D6 (Val_T6_D6 * 2.0);`) is required in this case because the result of the operation is of universal fixed type, so that it can be assigned to a variable of any fixed-point type.

Finally, in the `Val_T3_D6 := Val_T6_D6 / Val_T3_D3` statement, we're using three fixed-point types: we're dividing a variable of type T6_D6 by a variable of type T3_D3, and assigning it to a variable of type T3_D6. All these operations are only possible without explicit type conversions because the underlying types for the fixed-point division operation are universal fixed types.

For further reading...

It's possible to implement custom `*` and `/` operators for fixed-point types. However, those operators do **not** override the corresponding operators for universal fixed-point types. For example:

Listing 32: normalized_fixed_point_types.ads

```ada
package Normalized_Fixed_Point_Types is

   type TQ63 is
     delta 2.0 ** (-63)
     range -1.0 .. 1.0 - 2.0 ** (-63);

   type TQ31 is
     delta 2.0 ** (-31)
     range -1.0 .. 1.0 - 2.0 ** (-31);

   overriding
   --  ^^^^^^
   --  "+" operator is overriding!
   function "+" (L, R : TQ31)
                 return TQ31;

   not overriding
   --  ^^^^^^^^^^^
   --  "*" operator is NOT overriding!
   function "*" (L, R : TQ31)
                 return TQ31;

   type TQ15 is
     delta 2.0 ** (-15)
     range -1.0 .. 1.0 - 2.0 ** (-15);

end Normalized_Fixed_Point_Types;
```

Listing 33: normalized_fixed_point_types.adb

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

package body Normalized_Fixed_Point_Types is

   function "+" (L, R : TQ31)
                 return TQ31 is
   begin
      Put_Line
        ("=> Overriding '+'");
      return TQ31 (TQ63 (L) + TQ63 (R));
   end "+";

   function "*" (L, R : TQ31)
                 return TQ31 is
   begin
      Put_Line
        ("=> Custom "
         & "non-overriding '*'");
      return TQ31 (TQ63 (L) * TQ63 (R));
   end "*";

end Normalized_Fixed_Point_Types;
```

Listing 34: show_fixed_multiplication.adb

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

with Normalized_Fixed_Point_Types;
use  Normalized_Fixed_Point_Types;

procedure Show_Fixed_Multiplication is
   Q31_A : TQ31 := 0.25;
   Q31_B : TQ31 := 0.50;
   Q15_A : TQ15 := 0.25;
   Q15_B : TQ15 := 0.50;
begin
   Q31_A := Q31_A * Q31_B;
   Put_Line ("Q31_A = " & Q31_A'Image);

   Q15_A := Q15_A * Q15_B;
   Put_Line ("Q15_A = " & Q31_A'Image);

   Q15_A := TQ15 (Q31_A) * Q15_B;
   --        ^^^^^^^^^^^^
   --  A conversion is required because of
   --  the multiplication operator of
   --  TQ15.
   Put_Line ("Q31_A = " & Q31_A'Image);
end Show_Fixed_Multiplication;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Fixed_Point_
↪Custom_Multiplication
MD5: 954ada297ac676ab1f11447083d87882
```

Runtime output

```
=> Custom non-overriding '*'
Q31_A =  0.1250000000
Q15_A =  0.1250000000
Q31_A =  0.1250000000
```

In this example, we're declaring a custom multiplication operator for the TQ31 type. As we can see in the declaration, we specify that it's **not overriding** the * operator. (Removing the **not** keyword triggers a compilation error.) In contrast, for the + operator, we're indeed overriding the default + operator of the TQ31 type in the Normalized_Fixed_Point_Types because the addition operator is associated with its corresponding fixed-point type, not with the universal fixed-point type. In the Q31_A := Q31_A * Q31_B statement, we see at runtime (through the "=> Custom non-overriding '*'" message) that the custom multiplication is being used.

However, because of this custom * operator, we cannot mix objects of this type with objects of other fixed-point types in multiplication or division operations. Therefore, for a statement such as Q15_A := Q31_A * Q15_B, we have to convert Q31_A to the TQ15 type before multiplying it by Q15_B.

In the Ada Reference Manual

- 4.5.5 Multiplying Operators[155]

8.3 Attributes of Modular Types

In the Introduction to Ada course, we've seen that Ada has two kinds of integer type: signed[156] and modular[157] types. For example:

Listing 35: num_types.ads

```
package Num_Types is

   type Signed_Integer is range 1 .. 1_000_000;
   type Modular is mod 2**32;

end Num_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_1
MD5: 2dff9fe22c6bbe52f964befccf68debf
```

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[158]

8.3.1 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:

155 http://www.ada-auth.org/standards/22rm/html/RM-4-5-5.html
156 https://learn.adacore.com/courses/intro-to-ada/chapters/strongly_typed_language.html#intro-ada-integers
157 https://learn.adacore.com/courses/intro-to-ada/chapters/strongly_typed_language.html#intro-ada-unsigned-types
158 http://www.ada-auth.org/standards/22rm/html/RM-3-5-4.html

Listing 36: show_modular.adb

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

with Num_Types;   use Num_Types;

procedure Show_Modular is
   Modulus_Value : constant := Modular'Modulus;
begin
   Put_Line (Modulus_Value'Image);
end Show_Modular;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_1
MD5: 336254ebc8c09ee9921633f6919994fe
```

Runtime output

```
4294967296
```

When we run this example, we get 4294967296, which is equal to 2**32.

8.3.2 Mod Attribute

Note

This section was originally written by Robert A. Duff and published as Gem #26: The Mod Attribute[159].

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:

```
type Signed_Integer is range 1 .. 1_000_000;
```

The base range of Signed_Integer is the range of Signed_Integer'Base, which is chosen by the compiler, but is likely to be something like -2**31 .. 2**31 - 1. (Note: we discussed the Base attribute *in this section* (page 20).)

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

Listing 37: show_modular.adb

```
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;
```

[159] https://www.adacore.com/gems/gem-26

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_1
MD5: e9ac61d2e43585f002fe2b79544ef9d7
```

Runtime output

```
 1
4294967295
```

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 38: show_modular.adb

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

with Num_Types;   use Num_Types;

procedure Show_Modular is
   I : Integer := -1;
   X : Modular := 1;
begin
   X := Modular (I);  --  raises Constraint_Error
   Put_Line (X'Image);
end Show_Modular;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_1
MD5: e8e1a1924efcbe770c719c29547bb863
```

Build output

```
show_modular.adb:9:09: warning: value not in range of type "Modular" defined at
 ↪num_types.ads:4 [enabled by default]
show_modular.adb:9:09: warning: Constraint_Error will be raised at run time
 ↪[enabled by default]
```

Runtime output

```

raised CONSTRAINT_ERROR : show_modular.adb:9 range check failed
```

To solve this problem, we can use the **Mod** attribute:

Listing 39: show_modular.adb

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_1
MD5: 572a753de946b7578c5f1b6a795ede98
```

Runtime output

```
4294967295
```

The **Mod** attribute will correctly convert from any integer type to a given modular type, using wraparound semantics.

> **Historically**
>
> In older versions of Ada — such as Ada 95 —, the only way to do this conversion is to use `Unchecked_Conversion`, which is somewhat uncomfortable. Furthermore, if you're trying to convert to a generic formal modular type, how do you know what size of signed integer type to use? Note that `Unchecked_Conversion` might malfunction if the source and target types are of different sizes.
>
> The **Mod** attribute was added to Ada 2005 to solve this problem. Also, we can now safely use this attribute in generics. For example:

Listing 40: mod_attribute.ads

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

Listing 41: mod_attribute.adb

```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`.

8.3.3 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 42: crc_defs.ads

```
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 43: crc_defs.adb

```
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
```

(continues on next page)

(continued from previous page)

```
   begin
      for E of A loop
         Byte_IO.Put (E);
         Put (", ");
      end loop;
      New_Line;
   end Display;

begin
   Byte_IO.Default_Width := 1;
   Byte_IO.Default_Base  := 16;
   Crc_IO.Default_Width  := 1;
   Crc_IO.Default_Base   := 16;
end Crc_Defs;
```

Listing 44: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Mod_Crc_CCITT_Ada
MD5: 9c66abfadcce92231295cbccad087912
```

Runtime output

```
Input array: 16#0#, 16#20#, 16#30#,
CRC-CCITT: 16#21B9#
```

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.

8.4 Attributes of 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[160]
- A.5.3 Attributes of Floating Point Types[161]

8.4.1 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 45: show_machine_radix.adb

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

procedure Show_Machine_Radix is
begin
   Put_Line
     ("Float'Machine_Radix:           "
      & Float'Machine_Radix'Image);
   Put_Line
     ("Long_Float'Machine_Radix:      "
      & Long_Float'Machine_Radix'Image);
   Put_Line
     ("Long_Long_Float'Machine_Radix: "
      & Long_Long_Float'Machine_Radix'Image);
end Show_Machine_Radix;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Machine_
↪Radix
MD5: 88680df680f1db4ff803912850370551
```

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 46: show_machine_mantissa.adb

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

procedure Show_Machine_Mantissa is
begin
   Put_Line
     ("Float'Machine_Mantissa:           "
      & Float'Machine_Mantissa'Image);
   Put_Line
     ("Long_Float'Machine_Mantissa:      "
      & Long_Float'Machine_Mantissa'Image);
   Put_Line
     ("Long_Long_Float'Machine_Mantissa: "
```

(continues on next page)

[160] http://www.ada-auth.org/standards/22rm/html/RM-3-5-8.html
[161] http://www.ada-auth.org/standards/22rm/html/RM-A-5-3.html

(continued from previous page)

```
      & Long_Long_Float'Machine_Mantissa'Image);
end Show_Machine_Mantissa;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Machine_
↪Mantissa
MD5: da946a90a454c6e8f68cbff1ec54c7d3
```

Runtime output

```
Float'Machine_Mantissa:           24
Long_Float'Machine_Mantissa:      53
Long_Long_Float'Machine_Mantissa: 64
```

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:

Listing 47: show_machine_emin_emax.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

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Machine_
↪Emin_Emax
MD5: 9766e06faaf1fc2ca48dd0bc0461b247
```

Runtime output

```
Float'Machine_Emin:              -125
Float'Machine_Emax:               128
```

(continues on next page)

(continued from previous page)

```
Long_Float'Machine_Emin:          -1021
Long_Float'Machine_Emax:           1024
Long_Long_Float'Machine_Emin:     -16381
Long_Long_Float'Machine_Emax:      16384
```

On a typical desktop PC, the value of Float'Machine_Emin and Float'Machine_Emax is -125 and 128, respectively.

To get the actual minimum and maximum value of the exponent for a specific type, we need to use the Machine_Radix attribute that we've seen previously. Let's calculate the minimum and maximum value of the exponent for the Float type on a typical PC:

- Value of minimum exponent: Float'Machine_Radix ** Float'Machine_Emin.
  - In our target platform, this is 2^{-125} = 2.35098870164457501594 x 10^{-38}.
- Value of maximum exponent: Float'Machine_Radix ** Float'Machine_Emax.
  - In our target platform, this is 2^{128} = 3.40282366920938463463 x 10^{38}.

Attribute: Digits

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

Listing 48: show_digits.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Digits
MD5: cd1c88054f7d54703760a852d08acb6d
```

Runtime output

```
Float'Digits:            6
Long_Float'Digits:       15
Long_Long_Float'Digits:  18
```

Here, the requested decimal precision of the Float type is six digits.

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

Listing 49: show_base_digits.adb

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

procedure Show_Base_Digits is
   type Float_D3 is new Float digits 3;
```

(continues on next page)

(continued from previous page)

```
begin
   Put_Line ("Float_D3'Digits:            "
             & Float_D3'Digits'Image);
   Put_Line ("Float_D3'Base'Digits:       "
             & Float_D3'Base'Digits'Image);
end Show_Base_Digits;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Base_Digits
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[162].
- 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 50: show_boolean_attributes.adb

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

procedure Show_Boolean_Attributes is
begin
   Put_Line
     ("Float'Denorm:           "
      & Float'Denorm'Image);
   Put_Line
     ("Long_Float'Denorm:      "
      & Long_Float'Denorm'Image);
   Put_Line
     ("Long_Long_Float'Denorm: "
      & Long_Long_Float'Denorm'Image);
   Put_Line
     ("Float'Signed_Zeros:           "
      & Float'Signed_Zeros'Image);
   Put_Line
     ("Long_Float'Signed_Zeros:      "
      & Long_Float'Signed_Zeros'Image);
   Put_Line
```

(continues on next page)

162 https://en.wikipedia.org/wiki/Subnormal_number

(continued from previous page)

```
      ("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);
    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

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Machine_
↪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).

8.4.2 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 51: show_exponent_fraction_compose.adb

```
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

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Exponent_
↪Fraction
MD5: d2e61b6b9a7a50861145f6b65e9fac39
```

Runtime output

```
Float'Fraction (1.0):      5.00000E-01
Float'Fraction (0.25):     5.00000E-01
Float'Fraction (1.0e-25):  9.67141E-01
Float'Exponent (1.0):      1
Float'Exponent (0.25):    -1
Float'Exponent (1.0e-25): -83
Float'Compose (5.00000e-01, 1):    1.00000E+00
```

(continues on next page)

(continued from previous page)

```
Float'Compose (5.00000e-01, -1):   2.50000E-01
Float'Compose (9.67141E-01, -83):  1.00000E-25
```

To understand this code example, we have to take this formula into account:

Value = Fraction x Machine_Radix^{Exponent}

Considering that the value of `Float'Machine_Radix` on a typical PC is two, we see that the value 1.0 is composed by a fraction of 0.5 and a machine exponent of one. In other words:

0.5 x 2^1 = 1.0

For the value 0.25, we get a fraction of 0.5 and a machine exponent of -1, which is the result of 0.5 x 2^{-1} = 0.25. We can use the `Compose` attribute to perform this calculation. For example, `Float'Compose (0.5, -1) = 0.25`.

Note that `Fraction` is always between 0.5 and 0.999999 (i.e < 1.0), except for denormalized numbers, where it can be < 0.5.

Attribute: Scaling

`Scaling` is an attribute that scales a floating-point value based on the machine radix and a machine exponent passed to the function. For example:

Listing 52: show_scaling.adb

```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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Scaling
MD5: 9fa821d32911b74ee4b4fde3f3adafd8
```

Runtime output

```
Float'Scaling (0.25, 1):  5.00000E-01
Float'Scaling (0.25, 2):  1.00000E+00
Float'Scaling (0.25, 3):  2.00000E+00
```

The scaling is calculated with this formula:

scaling = value x Machine_Radix$^{machine\ exponent}$

For example, on a typical PC with a machine radix of two, `Float'Scaling (0.25, 3) = 2.0` corresponds to

0.25 x 2^3 = 2.0

Round-up and round-down attributes

Floor and Ceiling are attributes that returned the rounded-down or rounded-up value, respectively, of a floating-point value. For example:

Listing 53: show_floor_ceiling.adb

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

procedure Show_Floor_Ceiling is
begin
   Put_Line ("Float'Floor (0.25):    "
             & Float'Floor (0.25)'Image);
   Put_Line ("Float'Ceiling (0.25): "
             & Float'Ceiling (0.25)'Image);
end Show_Floor_Ceiling;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floor_
↪Ceiling
MD5: 1344d54ae86b9fd4831d5f078eb655d4
```

Runtime output

```
Float'Floor (0.25):    0.00000E+00
Float'Ceiling (0.25):  1.00000E+00
```

As we can see in this example, the rounded-down value (floor) of 0.25 is 0.0, while the rounded-up value (ceiling) of 0.25 is 1.0.

Round-to-nearest attributes

In this section, we discuss three attributes used for rounding: Rounding, Unbiased_Rounding, Machine_Rounding In all cases, the rounding attributes return the nearest integer value (as a floating-point value). For example, the rounded value for 4.8 is 5.0 because 5 is the closest integer value.

Let's see a code example:

Listing 54: show_roundings.adb

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

procedure Show_Roundings is
begin
   Put_Line
     ("Float'Rounding (0.5):  "
      & Float'Rounding (0.5)'Image);
   Put_Line
     ("Float'Rounding (1.5):  "
      & Float'Rounding (1.5)'Image);
   Put_Line
     ("Float'Rounding (4.5):  "
      & Float'Rounding (4.5)'Image);
   Put_Line
     ("Float'Rounding (-4.5): "
      & Float'Rounding (-4.5)'Image);
   Put_Line
     ("Float'Unbiased_Rounding (0.5): "
      & Float'Unbiased_Rounding (0.5)'Image);
   Put_Line
```

(continues on next page)

(continued from previous page)

```
      ("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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Rounding
MD5: 3f78165f092a163339cb9593ff15a50d
```

Runtime output

```
Float'Rounding (0.5):   1.00000E+00
Float'Rounding (1.5):   2.00000E+00
Float'Rounding (4.5):   5.00000E+00
Float'Rounding (-4.5): -5.00000E+00
Float'Unbiased_Rounding (0.5):  0.00000E+00
Float'Unbiased_Rounding (1.5):  2.00000E+00
Float'Machine_Rounding (0.5):  1.00000E+00
Float'Machine_Rounding (1.5):  2.00000E+00
```

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, `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 x divisor + remainder = dividend". For the previous example, this means 2 x 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 x 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 x 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 55: show_truncation_remainder_adjacent.adb

```
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;
```

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 56: show_copy_sign_leading_part_machine.adb

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

procedure Show_Copy_Sign_Leading_Part_Machine is
begin
   Put_Line
     ("Float'Copy_Sign (1.0, -10.0): "
      & Float'Copy_Sign (1.0, -10.0)'Image);
   Put_Line
     ("Float'Copy_Sign (-1.0, -10.0): "
      & Float'Copy_Sign (-1.0, -10.0)'Image);
   Put_Line
     ("Float'Copy_Sign (1.0,  10.0): "
      & Float'Copy_Sign (1.0,  10.0)'Image);
   Put_Line
     ("Float'Copy_Sign (1.0, -0.0):  "
      & Float'Copy_Sign (1.0, -0.0)'Image);
   Put_Line
     ("Float'Copy_Sign (1.0,  0.0):  "
      & Float'Copy_Sign (1.0,  0.0)'Image);
   Put_Line
     ("Float'Leading_Part (1.75, 1): "
      & Float'Leading_Part (1.75, 1)'Image);
   Put_Line
     ("Float'Leading_Part (1.75, 2): "
      & Float'Leading_Part (1.75, 2)'Image);
   Put_Line
     ("Float'Leading_Part (1.75, 3): "
```

(continues on next page)

(continued from previous page)

```
      & Float'Leading_Part (1.75, 3)'Image);
end Show_Copy_Sign_Leading_Part_Machine;
```

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 x 10^{15} (or 1,000,000,000,000,000):

Listing 57: show_float_value.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Float_Value
MD5: a7f80f7584ebaf39f2d5f9564c9c7d64
```

Runtime output

```
1.0E+15 = 999999986991000.0
```

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 x 10^{15} isn't directly representable on this machine, so it has to be modified to a value that is actually representable (on the machine).

This *automatic* modification we've just described is actually hidden, so to say, in the assignment. However, we can make it more visible by using the `Machine (X)` attribute, which returns a version of X that is representable on the target machine. The `Machine (X)` attribute rounds (or truncates) X to either one of the adjacent machine numbers for the specific floating-point type of X. (Of course, if the real value of X is directly representable on the target machine, no modification is performed.)

In fact, we could rewrite the `V := 1.0E+15` assignment of the code example as `V := Float'Machine (1.0E+15)`, as we're never assigning a real value directly to a floating-pointing variable — instead, we're first converting it to a version of the real value that is representable on the target machine. In this case, 999999986991000.0 is a representable version of the real value 1.0 x 10^{15}. Of course, writing `V := 1.0E+15` or `V := Float'Machine (1.0E+15)` doesn't make any difference to the actual value that is assigned to V (in the case of this specific target architecture), as the conversion to a representable value happens automatically during the assignment to V.

There are, however, instances where using the `Machine` attribute does make a difference in the result. For example, let's say we want to calculate the difference between the original real value in our example (1.0 x 10^{15}) and the actual value that is assigned to V. We can do this by using the Machine attribute in the calculation:

Listing 58: show_machine_attribute.adb

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

procedure Show_Machine_Attribute is
   package F_IO is new
     Ada.Text_IO.Float_IO (Float);

   V : Float;
begin
   F_IO.Default_Fore := 3;
   F_IO.Default_Aft  := 1;
   F_IO.Default_Exp  := 0;

   Put_Line
     ("Original value: 1_000_000_000_000_000.0");

   V := 1.0E+15;
   Put ("Machine value:  ");
   F_IO.Put (Item => V);
   New_Line;

   V := 1.0E+15 - Float'Machine (1.0E+15);
   Put ("Difference:     ");
   F_IO.Put (Item => V);
   New_Line;

end Show_Machine_Attribute;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Machine_
↪Attribute
MD5: c2db2cca028dc5811068f9b7f1bc209d
```

Runtime output

```
Original value: 1_000_000_000_000_000.0
Machine value:  999999986991000.0
Difference:     13008896.0
```

When we run this example on a typical PC, we see that the difference is roughly 1.3009 x 10^7. (Actually, the value that we might see is 1.3008896 x 10^7, which is a version of 1.3009 x 10^7 that is representable on the target machine.)

When we write `1.0E+15 - Float'Machine (1.0E+15)`:

- the first value in the operation is the universal real value 1.0 x 10^{15}, while
- the second value in the operation is a version of the universal real value 1.0 x 10^{15} that is representable on the target machine.

This also means that, in the assignment to V, we're actually writing `V := Float'Machine (1.0E+15 - Float'Machine (1.0E+15))`, so that:

1. we first get the intermediate real value that represents the difference between these values; and then
2. we get a version of this intermediate real value that is representable on the target machine.

This is the reason why we see 1.3008896 x 10^7 instead of 1.3009 x 10^7 when we run this application.

8.5 Attributes of 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[163]
- A.5.4 Attributes of Fixed Point Types[164]

8.5.1 Attributes of ordinary and decimal fixed-point types

Attribute: Machine_Radix

Machine_Radix is an attribute that returns the radix of the hardware representation of a type. For example:

Listing 59: show_fixed_machine_radix.adb

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

procedure Show_Fixed_Machine_Radix is
   type T3_D3 is delta 10.0 ** (-3) digits 3;

   D : constant := 2.0 ** (-31);
   type TQ31 is delta D range -1.0 .. 1.0 - D;
begin
   Put_Line ("T3_D3'Machine_Radix: "
             & T3_D3'Machine_Radix'Image);
   Put_Line ("TQ31'Machine_Radix:  "
             & TQ31'Machine_Radix'Image);
end Show_Fixed_Machine_Radix;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Fixed_Machine_
↪Radix
MD5: a09d060a58f76550e948a8245ffb5fde
```

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:

163 http://www.ada-auth.org/standards/22rm/html/RM-3-5-10.html
164 http://www.ada-auth.org/standards/22rm/html/RM-A-5-4.html

- 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 60: show_boolean_attributes.adb

```
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 61: fixed_point_defs.ads

```
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 62: show_fixed_type_info.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Large_Small_
↪Attribute
MD5: 89672950b355060d250e0f5d7e2d40cb
```

Runtime output

```
Size :  32
Small :  1.58456325028528675E+29
Delta :  1.58456325028528675E+29
First : -340282366920938463463374607431768211456.0
Last :  340282366762482138434845932244680310784.0
```

In this example, the *small* of the Fixed type is actually quite large: 1.58456325028528675^{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:

- 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'Small <= T'Delta`.
- If we want, however, to specify the *small*, we can do that by using the `Small` aspect. In this case, it doesn't need to be a power of two.

For decimal fixed-point types, we cannot specify the *small*. In this case, it's automatically selected by the compiler, and it's always equal to the *delta*.

Let's see an example:

Listing 63: fixed_small_delta.ads

```
package Fixed_Small_Delta is
   D3 : constant := 10.0 ** (-3);

   type T3_D3 is delta D3 digits 3;

```

(continues on next page)

(continued from previous page)

```
   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 64: show_fixed_small_delta.adb

```
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 ("TQ32'Size: "
             & TQ31'Size'Image);
   Put_Line ("--------------------");

   Put_Line ("TQ15'Small: "
             & TQ15'Small'Image);
   Put_Line ("TQ15'Delta: "
             & TQ15'Delta'Image);
   Put_Line ("TQ15'Size: "
             & TQ15'Size'Image);
end Show_Fixed_Small_Delta;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Fixed_Small_
↪Delta
MD5: 0e811c7c0b92f05483b0ac7c3489dc3d
```

Runtime output

```
T3_D3'Small:  1.00000000000000000E-03
T3_D3'Delta:  1.00000000000000000E-03
```

(continues on next page)

(continued from previous page)

```
T3_D3'Size:  11
--------------------
TD3'Small:  9.76562500000000000E-04
TD3'Delta:  1.00000000000000000E-03
TD3'Size:  11
--------------------
TQ31'Small:  4.65661287307739258E-10
TQ31'Delta:  4.65661287307739258E-10
TQ32'Size:  32
--------------------
TQ15'Small:  4.65661287307739258E-10
TQ15'Delta:  3.05175781250000000E-05
TQ15'Size:  32
```

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:

Listing 65: show_fixed_small_delta.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Fixed_Small_
↪Delta
MD5: f2a71db911913d6fbf5343671599c0ae
```

Runtime output

```
V'Size:  32
V:  0.00000
V:  0.00003
```

In the first assignment, we assign TQ15'Small (2^{-31}) to V. This value is smaller than the type's *delta* (2^{-15}). Even though V'Size is 32 bits, V'Delta indicates 16-bit precision, and TQ15'Small requires 32-bit precision to be represented correctly. As a result, V has a value of zero after this assignment.

In contrast, after the second assignment — where we assign TQ15'Delta (2^{-15}) to V — we see, as expected, that V has the same value as the *delta*.

Attributes: Fore and Aft

The Fore and Aft attributes indicate the number of characters or digits needed for displaying a value in decimal representation. To be more precise:

- The Fore attribute refers to the digits before the decimal point and it returns the number of digits plus one for the sign indicator (which is either - or space), and it's always at least two.
- The Aft attribute returns the number of decimal digits that is needed to represent the delta after the decimal point.

Let's see an example:

Listing 66: show_fixed_fore_aft.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Fixed_Fore_Aft
MD5: d031f74d967a96dee1c6a83ff4bd14cf
```

Runtime output

```
T3_D3'Fore:  2
T3_D3'Aft:   3
TQ31'Fore:  2
TQ31'Aft:   10
----
Dec: -0.123
Fix: -0.0000000005
```

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".

8.5.2 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 67: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Decimal_Digits
MD5: d46e67bd0f8b369918e7ab9ab4413ae7
```

Runtime output

```
T3_D6'Digits:  6
T3_D2'Digits:  2
```

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 68: show_decimal_scale.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Decimal_Scale
MD5: 56a99848cf31a9c69fe6d91ead73375a
```

Runtime output

```
TM3_D6'Scale: -3
T3_D6'Scale:  3
T9_D12'Scale:  9
```

In this example, we get the following values for the scales:

- `TM3_D6'Scale = -3`,
- `T3_D6'Scale = 3`,
- `T9_D12 = 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 69: show_decimal_round.adb

```
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);
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("T3_D3'Round (0.2777): "
             & T3_D3'Round (0.2777)'Image);
end Show_Decimal_Round;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Decimal_Round
MD5: 153d9dae52fee750da30dd9152a03c37
```

Runtime output

```
T3_D3'Round (0.2774):  0.277
T3_D3'Round (0.2777):  0.278
```

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.

8.6 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[165]. 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:

Category	Package
Big Integers	Ada.Numerics.Big_Numbers.Big_Integers
Big Reals	Ada.Numerics.Big_Numbers.Big_Real

In the Ada Reference Manual

- Big Numbers[166]
- Big Integers[167]
- Big Reals[168]

8.6.1 Overview

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

165 https://en.wikipedia.org/wiki/arbitrary-precision_arithmetic
166 http://www.ada-auth.org/standards/22rm/html/RM-A-5-5.html
167 http://www.ada-auth.org/standards/22rm/html/RM-A-5-6.html
168 http://www.ada-auth.org/standards/22rm/html/RM-A-5-7.html

Listing 70: show_simple_big_numbers.adb

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

with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;

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

procedure Show_Simple_Big_Numbers is
   BI : Big_Integer;
   BR : Big_Real;
begin
   BI := 12345678901234567890;
   BR := 2.0 ** 1234;

   Put_Line ("BI: " & BI'Image);
   Put_Line ("BR: " & BR'Image);

   BI := BI + 1;
   BR := BR + 1.0;

   Put_Line ("BI: " & BI'Image);
   Put_Line ("BR: " & BR'Image);
end Show_Simple_Big_Numbers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Numbers
MD5: b6a5e9ad170b09cbbabeb3ce06cc958c
```

Runtime output

```
BI:  12345678901234567890
BR:␣
↪2958112246080986290600446957161035907863396871353729922395562070506573507962389242610538372483
↪000
BI:  12345678901234567891
BR:␣
↪2958112246080986290600446957161035907863396871353729922395562070506573507962389242610538372483
↪000
```

In this example, we're declaring the big integer BI and the big real BR, and we're incrementing them by one.

Naturally, we're not limited to using the + operator (such as in this example). We can use the same operators on big numbers that we can use with normal numeric types. In fact, the common unary operators (+, -, **abs**) and binary operators (+, -, *, /, **, Min and Max) are available to us. For example:

Listing 71: show_simple_big_numbers_operators.adb

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

with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;

procedure Show_Simple_Big_Numbers_Operators is
   BI : Big_Integer;
begin
   BI := 12345678901234567890;
```

(continues on next page)

(continued from previous page)

```

   Put_Line ("BI: " & BI'Image);

   BI := -BI + BI / 2;
   BI :=  BI - BI * 2;

   Put_Line ("BI: " & BI'Image);
end Show_Simple_Big_Numbers_Operators;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Numbers_
↪Operators
MD5: 198708787bfcd6e16ec4fba718706af6
```

Runtime output

```
BI:  12345678901234567890
BI:  6172839450617283945
```

In this example, we're applying the four basic operators (+, -, *, /) on big integers.

8.6.2 Factorial

A typical example is the factorial[169]: a sequence of the factorial of consecutive small numbers can quickly lead to big numbers. Let's take this implementation as an example:

Listing 72: factorial.ads

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

Listing 73: factorial.adb

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

   return Fact;
end Factorial;
```

Listing 74: 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;
```

[169] https://en.wikipedia.org/wiki/Factorial

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Factorial_Integer
MD5: 9b20469533706ef025a03b506a07b920
```

Runtime output

```
 1! =  1
 2! =  2
 3! =  6
 4! =  24
 5! =  120
 6! =  720
 7! =  5040
 8! =  40320
 9! =  362880
 10! =  3628800
 11! =  39916800
 12! =  479001600
 13! =  6227020800
 14! =  87178291200
 15! =  1307674368000
 16! =  20922789888000
 17! =  355687428096000
 18! =  6402373705728000
 19! =  121645100408832000
 20! =  2432902008176640000

raised CONSTRAINT_ERROR : factorial.adb:6 overflow check failed
```

Here, we're using **Long_Long_Integer** for the computation and return type of the `Factorial` function. (We're using **Long_Long_Integer** because its range is probably the biggest possible on the machine, although that is not necessarily so.) The last number we're able to calculate before getting an exception is *20!*, which basically shows the limitation of standard integers for this kind of algorithm. If we use big integers instead, we can easily display all numbers up to *50!* (and more!):

Listing 75: factorial.ads

```ada
with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;

function Factorial (N : Integer)
                    return Big_Integer;
```

Listing 76: factorial.adb

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

   return Fact;
end Factorial;
```

Listing 77: show_big_number_factorial.adb

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

(continues on next page)

(continued from previous page)

```

with Factorial;

procedure Show_Big_Number_Factorial is
begin
   for I in 1 .. 50 loop
      Put_Line (I'Image & "! = "
                & Factorial (I)'Image);
   end loop;
end Show_Big_Number_Factorial;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Factorial_Big_Numbers
MD5: d1f6464a3232d574d01f7ac14b822731
```

Runtime output

```
 1! =  1
 2! =  2
 3! =  6
 4! =  24
 5! =  120
 6! =  720
 7! =  5040
 8! =  40320
 9! =  362880
 10! =  3628800
 11! =  39916800
 12! =  479001600
 13! =  6227020800
 14! =  87178291200
 15! =  1307674368000
 16! =  20922789888000
 17! =  355687428096000
 18! =  6402373705728000
 19! =  121645100408832000
 20! =  2432902008176640000
 21! =  51090942171709440000
 22! =  1124000727777607680000
 23! =  25852016738884976640000
 24! =  620448401733239439360000
 25! =  15511210043330985984000000
 26! =  403291461126605635584000000
 27! =  10888869450418352160768000000
 28! =  304888344611713860501504000000
 29! =  8841761993739701954543616000000
 30! =  265252859812191058636308480000000
 31! =  8222838654177922817725562880000000
 32! =  263130836933693530167218012160000000
 33! =  8683317618811886495518194401280000000
 34! =  295232799039604140847618609643520000000
 35! =  10333147966386144929666651337523200000000
 36! =  371993326789901217467999448150835200000000
 37! =  13763753091226345046315979581580902400000000
 38! =  523022617466601111760007224100074291200000000
 39! =  20397882081197443358640281739902897356800000000
 40! =  815915283247897734345611269596115894272000000000
 41! =  33452526613163807108170062053440751665152000000000
 42! =  1405006117752879898543142606244511569936384000000000
 43! =  60415263063373835637355132068513997507264512000000000
```

(continues on next page)

(continued from previous page)

```
44! =  2658271574788448768043625811014615890319638528000000000
45! =  119622220865480194561963161495657715064383733760000000000
46! =  5502622159812088949850305428800254892961651752960000000000
47! =  258623241511168180642964355153611979969197632389120000000000
48! =  12413915592536072670862289047373375038521486354677760000000000
49! =  608281864034267560872252163321295376887552831379210240000000000
50! =  30414093201713378043612608166064768844377641568960512000000000000
```

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.

8.6.3 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 a actual value of a big number is valid:

Type	Subtype for valid values
Big Integers	Valid_Big_Integer
Big Reals	Valid_Big_Real

These subtypes include a contract for this check. For example, this is the definition of the Valid_Big_Integer subtype:

```
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:

```
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:

Category	To big number	From big number
Big Integers	• To_Big_Integer	• To_Integer (Integer) • From_Big_Integer (other integer types)
Big Reals	• To_Big_Real (floating-point types or fixed-point types)	• From_Big_Real
	• To_Big_Real (Valid_Big_Integer) • To_Real (Integer)	• Numerator, Denominator (Integer)

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:

Listing 78: show_simple_big_integer_conversion.adb

```ada
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;
   I  : Integer := 10000;
begin
   BI := To_Big_Integer (I);
   Put_Line ("BI: " & BI'Image);

   I := To_Integer (BI + 1);
   Put_Line ("I:  " & I'Image);
end Show_Simple_Big_Integer_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Integer_
↪Conversion
MD5: 83effc9da9835d92f4c49ed03d7ed84a
```

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:

Listing 79: show_arbitrary_big_integer_conversion.adb

```
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;

   BI   : Big_Integer;
   LLI  : Long_Long_Integer := 10000;
   U_32 : Mod_32_Bit        := 2 ** 32 + 1;

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Arbitrary_Big_
↪Integer_Conversion
MD5: a89b42ff012c8729770eefa2d2b1f6c1
```

Runtime output

```
BI:    10000
LLI:   10001
BI:    1
U_32:  2
```

In this examples, we declare the Long_Long_Integer_Conversions and the Mod_32_Bit_Conversions to be able to convert between big integers and the **Long_Long_Integer** and the Mod_32_Bit types, respectively.

Note that, when converting from big integer to integer, we used the To_Integer function, while, when using the instances of the generic packages, the function is named From_Big_Integer.

Big real to floating-point types

When converting between big real and floating-point types, we have to instantiate the generic Float_Conversions package:

Listing 80: show_big_real_floating_point_conversion.adb

```
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;

   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

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Floating_
↪Point_Conversion
MD5: 4ccb570b964d11d215660f5929f2709c
```

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:

Listing 81: show_big_real_fixed_point_conversion.adb

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

   package TQ31_Conversions is new
     Fixed_Conversions (TQ31);
   use TQ31_Conversions;

   BR   : Big_Real;
   FQ31 : TQ31 := 0.25;

begin
   BR := To_Big_Real (FQ31);
   Put_Line ("BR:   " & BR'Image);

   FQ31 := From_Big_Real (BR * 2.0);
   Put_Line ("FQ31: " & FQ31'Image);

end Show_Big_Real_Fixed_Point_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Fixed_Point_
↪Conversion
MD5: 49f03e130ec34842cbac7a728a280821
```

Runtime output

```
BR:    0.250
FQ31:  0.5000000000
```

In this example, we declare the TQ31_Conversions to be able to convert between big reals and the custom fixed-point type TQ31 type. Again, we use the To_Big_Real and the From_Big_Real functions for the conversions.

Note that there's no direct way to convert between decimal fixed-point types and big real types. (Of course, you could perform this conversion indirectly by using a floating-point or an ordinary fixed-point type in between.)

Big reals to (big) integers

We can also convert between big reals and big integers (or standard integers):

Listing 82: show_big_real_big_integer_conversion.adb

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

with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;
```

(continues on next page)

(continued from previous page)

```

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

procedure Show_Big_Real_Big_Integer_Conversion
is
   I  : Integer;
   BI : Big_Integer;
   BR : Big_Real;

begin
   I  := 12345;
   BR := To_Real (I);
   Put_Line ("BR (from I):  " & BR'Image);

   BI := 123456;
   BR := To_Big_Real (BI);
   Put_Line ("BR (from BI): " & BR'Image);

end Show_Big_Real_Big_Integer_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Big_Integer_
↪Conversion
MD5: 26bf2a4704ce98709587eedab3391119
```

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 83: show_big_number_string_conversion.adb

```
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: "
```

(continues on next page)

(continued from previous page)

```
             & To_String (Arg   => BR,
                          Fore  => 2,
                          Aft   => 6,
                          Exp   => 18));
end Show_Big_Number_String_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Number_String_
↪Conversion
MD5: aa1f19af04b0b901a086ac86151693a7
```

Runtime output

```
BI:  2#1010101101010100101010011000110011101011000111110000101011010010#
BR: 12.345678E+18
```

In this example, we use the From_String to convert a string to a big number. Note that the From_String function is actually called when converting a literal — because of the corresponding aspect for user-defined literals in the definitions of the Big_Integer and the Big_Real types.

For further reading...

Big numbers are implemented using *user-defined literals* (page 74), 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.

8.6.4 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:

```
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:

Listing 84: show_big_positive_natural.adb

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

with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;

procedure Show_Big_Positive_Natural is
   BI, D, N : Big_Integer;
begin
   D  := 3;
   N  := 2;
   BI := Big_Natural (D / Big_Positive (N));

   Put_Line ("BI: " & BI'Image);
end Show_Big_Positive_Natural;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Positive_Natural
MD5: 844b41f001c9aed9cb99decb221d93fd
```

Runtime output

```
BI:  1
```

By using the `Big_Natural` and `Big_Positive` subtypes in the calculation above (in the assignment to `BI`), we ensure that we don't perform a division by zero, and that the result of the calculation is a natural number.

8.6.5 Other operators for big integers

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

Listing 85: show_big_integer_rem_mod.adb

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Integer_Rem_Mod
MD5: 7347b617c51a3782921d997b3cfd5d37
```

Runtime output

```
BI (mod): -5
BI (rem):  1
```

In this example, we use the **mod** and **rem** operators in the assignments to `BI`.

Moreover, there's a `Greatest_Common_Divisor` function for big integers which, as the name suggests, calculates the greatest common divisor of two big integer values:

Listing 86: show_big_integer_greatest_common_divisor.adb

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Integer_Greatest_
↪Common_Divisor
MD5: 27e2f7b4cbe20ec979b672f3e7edfdb7
```

Runtime output

```
BI:  5
```

In this example, we retrieve the greatest common divisor of 145 and 25 (i.e.: 5).

8.6.6 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 87: show_big_real_quotient_conversion.adb

```
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;
   --  Same as:
   --  BR := From_Quotient_String ("2 / 3");

   Put_Line ("BR:    " & BR'Image);

   Put_Line ("Q:     "
             & To_Quotient_String (BR));

   Put_Line ("Q numerator:     "
             & Numerator (BR)'Image);
   Put_Line ("Q denominator:   "
             & Denominator (BR)'Image);
end Show_Big_Real_Quotient_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Quotient_
↪Conversion
MD5: 97d78457d3f6d5e1810e461c2c7cd172
```

Runtime output

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

In this example, we assign `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.

8.6.7 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 88: show_big_numbers_in_range.adb

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

with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;

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

procedure Show_Big_Numbers_In_Range is

   BI : Big_Integer;
   BR : Big_Real;

   BI_From : constant Big_Integer := 0;
   BI_To   : constant Big_Integer := 1024;

   BR_From : constant Big_Real := 0.0;
   BR_To   : constant Big_Real := 1024.0;

begin
   BI := 1023;
   BR := 1023.9;

   if In_Range (BI, BI_From, BI_To) then
      Put_Line ("BI ("
                & BI'Image
                & ") is in the "
                & BI_From'Image
                & " .. "
                & BI_To'Image
                & " range");
   end if;

   if In_Range (BR, BR_From, BR_To) then
      Put_Line ("BR ("
                & BR'Image
                & ") is in the "
                & BR_From'Image
                & " .. "
                & BR_To'Image
                & " range");
   end if;

end Show_Big_Numbers_In_Range;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Numbers_In_Range
MD5: ded52ef7e9ef13a83264940ff9d8bcb3
```

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

CHAPTER

NINE

EXPRESSIONS

9.1 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[170] 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[171]

9.1.1 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;

   Dummy :=
       M1 in Active_Mode
```

(continues on next page)

170 http://www.ada-auth.org/standards/22rm/html/RM-4-4.html
171 http://www.ada-auth.org/standards/22rm/html/RM-4-4.html

(continued from previous page)

```
                and then M2 in Off | A;
   --
   --   ^^^^^^^^^^^^^^^^^ relation
   --
   --                     ^^^^^^^^^^^^^^ relation
   --   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   --                                 expression

   Dummy :=
       M1 in Active_Mode;
   --  ^^ name
   --  ^^ primary
   --  ^^ factor
   --  ^^ term
   --  ^^ simple expression
   --
   --        ^^^^^^^^^^^ membership choice
   --        ^^^^^^^^^^^ membership choice list
   --
   --  ^^^^^^^^^^^^^^^^^ relation
   --  ^^^^^^^^^^^^^^^^^ expression

   Dummy :=
       M2 in Off | A;
   --  ^^ name
   --  ^^ primary
   --  ^^ factor
   --  ^^ term
   --  ^^ simple expression
   --
   --        ^^^ membership choice
   --              ^ membership choice
   --        ^^^^^^^ membership choice list
   --
   --  ^^^^^^^^^^^^^ relation
   --  ^^^^^^^^^^^^^ expression

end Show_Expression_Elements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.
↪Expression_Elements
MD5: a22e6f2d2bc181ce77097a1de204eb62
```

Build output

```
show_expression_elements.adb:9:08: warning: variable "M2" is read but never␣
↪assigned [-gnatwv]
```

In this code example, we see three expressions. As we mentioned earlier, every expression has a type; here, the type of each expression is **Boolean**.

The first expression (`M1 in Active_Mode and then M2 in Off | A`) consists of two relations: `M1 in Active_Mode` and `M2 in Off | A`. Let's discuss some of the details.

The `M1 in Active_Mode` relation consists of the simple expression `M1` and the membership choice list `Active_Mode`. (Here, the **in** keyword is part of the relation definition.) Also, as we see in the comments of the source code, the simple expression `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 251);
  - a name;
  - an allocator (like `new Integer`);
  - *a parenthesized expression* (page 431);
  - *a conditional expression* (page 433);
  - *a quantified expression* (page 436);
  - *a declare expression* (page 440).

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:

```
simple_expression ::=
  [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
  numeric_literal | null | string_literal | aggregate
| name | allocator | (expression)
| (conditional_expression) | (quantified_expression)
| (declare_expression)
```

Later on in this chapter, we discuss *conditional expressions* (page 433), *quantified expressions* (page 436) and *declare expressions* (page 440) in more details.

In the relation `M2 in 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:

```
expression ::=
     relation {and relation}
   | relation {and then relation}
   | relation {or relation}
   | relation {or else relation}
   | relation {xor relation}
```

```
relation ::=
     simple_expression
       [relational_operator simple_expression]
   | simple_expression [not] in
       membership_choice_list
   | raise_expression
```

Again, for more details, please refer to the section on expressions[172] of the Ada Reference Manual.

In the Ada Reference Manual

- 4.4 Expressions[173]
- 4.5.2 Relational Operators and Membership Tests[174]

9.1.2 Numeric expressions

The expressions we've seen so far had the **Boolean** type. Although much of the grammar described in the Manual exists exclusively for Boolean operations, we can also write numeric expressions such as the following one:

Listing 2: show_numeric_expressions.adb

```
procedure Show_Numeric_Expressions is
   C1    : constant Integer := 5;
   Dummy :          Integer;
begin
   Dummy :=
       -2 ** 4 + 3 * C1 ** 8;
   --                      ^ numeric literal
   --                      ^ primary
   --                ^^      name
   --                ^^      primary
   --                ^^^^^^^ factor
   --              ^ multiplying operator
   --            ^           numeric literal
   --            ^           primary
   --            ^           factor
   --            ^^^^^^^^^^^ term
   --
   --        ^ numeric literal
   --        ^ primary
   --    ^ numeric literal
   --    ^ primary
   --    ^^^^^^               factor
   --    ^^^^^^               term
   --          ^ binary adding operator
   --   ^ unary adding operator
   --
   --   ^^^^^^^^^^^^^^^^^^^^^^ simple expression
   --
   --   ^^^^^^^^^^^^^^^^^^^^^^ expression
end Show_Numeric_Expressions;
```

Code block metadata

[172] http://www.ada-auth.org/standards/22rm/html/RM-4-4.html
[173] http://www.ada-auth.org/standards/22rm/html/RM-4-4.html
[174] http://www.ada-auth.org/standards/22rm/html/RM-4-5-2.html

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.
↪Numeric_Expressions
MD5: a3c902c7aa5b0afe30ae220256c3306a
```

In this code example, the expression `- 2 ** 4 + 3 * C1 ** 8` consists of just a single simple expression. (Note that simple expressions do not have to be "simple".) This simple expression consists of two terms: `2 ** 4` and `3 * C1 ** 8`. While the `2 ** 4` term is also a single factor, the `3 * C1 ** 8` term consists of two factors: `3` and `C1 ** 8`. Both the `2 ** 4` and the `C1 ** 8` factors consists of two primaries each:

- the `2 ** 4` factor has the primaries `2` and `4`,
- the `C1 ** 8` factor has the primaries `C1` and `8`.

In the Ada Reference Manual

- 4.4 Expressions[175]

9.1.3 Other expressions

Expressions aren't limited to the **Boolean** type or to numeric types. Indeed, expressions can be of any type, and the definition of primaries we've seen earlier on already hints in this direction — as it includes elements such as allocators. Because expressions are very flexible, covering all possible variations and combinations in this section is out of scope. Again, please refer to the section on expressions[176] of the Ada Reference Manual for further details.

9.1.4 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:

Listing 3: show_parenthesized_expressions.adb

```
procedure Show_Parenthesized_Expressions is
   C1 : constant Integer := 4;
   C2 : constant Integer := 5;

   Dummy : Integer;
begin
   Dummy :=
       (2 + C1) * C2;
   --       ^^       name
   --       ^^       primary
   --       ^^       factor
   --       ^^       term
   --
   --   ^            numeric literal
   --   ^            primary
   --   ^            factor
   --   ^            term
   --
   --     ^          binary adding operator
   --  ^^^^^^^^      simple expression
   --
```

(continues on next page)

[175] http://www.ada-auth.org/standards/22rm/html/RM-4-4.html
[176] http://www.ada-auth.org/standards/22rm/html/RM-4-4.html

(continued from previous page)

```
   --  ^^^^^^^^      expression
   --  ^^^^^^^^      primary
   --  ^^^^^^^^      factor
   --
   --            ^^ factor
   --  ^^^^^^^^^^^^^ term
   --
   --  ^^^^^^^^^^^^^ simple expression
   --
   --  ^^^^^^^^^^^^^ expression
end Show_Parenthesized_Expressions;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.
↪Parenthesized_Expressions
MD5: 5871d2b0cd33e4f562b96381e0f0d293
```

In this example, we first start with the single expression `(2 + C1) * 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:

Listing 4: show_name_in_expression.adb

```
procedure Show_Name_In_Expression is
   type Mode is (Off, A, B, C, D);

   M1 : Mode;
begin
   M1 := A;

   case M1 is
     when Off | D   =>
       null;
     when A | B | C =>
       M1 := D;
   end case;

end Show_Name_In_Expression;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.Name_
↪In_Expression
```

(continues on next page)

(continued from previous page)

```
MD5: ec8fcbc511e6a372da4f0ad99d2619a5
```

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 458).

Let's look at another example, this time with a subprogram call:

Listing 5: increment_by_one.ads

```ada
procedure Increment_By_One (I : in out Integer);
```

Listing 6: increment_by_one.adb

```ada
procedure Increment_By_One (I : in out Integer) is
begin
   I := I + 1;
end Increment_By_One;
```

Listing 7: show_name_in_expression.adb

```ada
with Increment_By_One;

procedure Show_Name_In_Expression is
   V : Integer := 0;
begin
   Increment_By_One ((V));
end Show_Name_In_Expression;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.Name_
↪In_Expression
MD5: 4805df49dc702e5cb365252e58742dd2
```

Build output

```
show_name_in_expression.adb:6:23: error: actual for "I" must be a variable
gprbuild: *** compilation phase failed
```

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.)

9.2 Conditional Expressions

As we've seen before, we can write simple expressions such as I = 0 or D.Valid. A conditional expression, as the name implies, is an expression that contains a condition. This might be an "if-expression" (in the **if** ... **then** ... **else** form) or a "case-expression" (in the **case** ... **is** **when** => 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

```
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

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
↪Conditional_If_Expressions_1
MD5: 7a99711afecc0b481557f9874dfbf4de
```

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

```
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

```
package body Expr_Func is

   function Toggled (S : State) return State is
   begin
      if S = Off then
         return On;
```

(continues on next page)

(continued from previous page)

```
      elsif S = On then
         return Off;
      else
         return S;
      end if;
   end Toggled;

end Expr_Func;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
↪Conditional_If_Expressions_2
MD5: 9e6cdf53c9c934f37e5717e1d230615a
```

If we compare the if-block of this code example to the if-expression of the previous example, we notice that the if-expression is just a simplified version without the **return** keyword and the **end if**;. In fact, converting an if-block to an if-expression is quite straightforward.

We could also replace the if-expression used in the Toggled function above with a case-expression. For example:

Listing 12: expr_func.ads

```
package Expr_Func is

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

   function Toggled (S : State) return State is
     (case S is
       when Off    => On,
       when On     => Off,
       when others => S);

end Expr_Func;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
↪Conditional_Case_Expressions_1
MD5: 0dd3a86f0872d1e8c3a81f7a17c44bd5
```

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

```
package Expr_Func is

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

   function Toggled (S : State) return State;

end Expr_Func;
```

Listing 14: expr_func.adb

```
package body Expr_Func is

   function Toggled (S : State) return State is
   begin
```

(continues on next page)

(continued from previous page)

```
      case S is
         when Off    => return On;
         when On     => return Off;
         when others => return S;
      end case;
   end Toggled;

end Expr_Func;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
↪Conditional_Case_Expressions_2
MD5: db6a0737e3931c83c31f53e4da3d8a2b
```

If we compare the case block of this code example to the case-expression of the previous example, we notice that the case-expression is just a simplified version of the case block without the **return** keyword and the **end case**;, and with alternatives separated by commas instead of semicolons.

In the Ada Reference Manual

- 4.5.7 Conditional Expressions[177]

9.3 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:

Listing 15: int_arrays.ads

```
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);
```

(continues on next page)

[177] http://www.ada-auth.org/standards/22rm/html/RM-4-5-7.html

(continued from previous page)

```

   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;
```

Listing 16: int_arrays.adb

```
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;
```

Listing 17: test_int_arrays.adb

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

with Int_Arrays;  use Int_Arrays;

procedure Test_Int_Arrays is
   A : Integer_Arr := (0, 0, 1);
begin
   Display_Array (A, "A");
   Put_Line ("Is_Zero: "
             & Boolean'Image (Is_Zero (A)));
   Put_Line ("Has_Zero: "
             & Boolean'Image (Has_Zero (A)));

   A := (0, 0, 0);

   Display_Array (A, "A");
   Put_Line ("Is_Zero: "
             & Boolean'Image (Is_Zero (A)));
   Put_Line ("Has_Zero: "
             & Boolean'Image (Has_Zero (A)));
end Test_Int_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Quantified_Expression.
↪Quantified_Expression_1
MD5: 4bbda8a3830272748500f797f23f76fc
```

Runtime output

```
A:  0  0  1
Is_Zero: FALSE
Has_Zero: TRUE
A:  0  0  0
Is_Zero: TRUE
Has_Zero: TRUE
```

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;

   procedure Display_Array (A    : Integer_Arr;
                            Name : String) is
```

(continues on next page)

(continued from previous page)

```
   begin
      Put (Name & ": ");
      for E of A loop
         Put (E'Image & " ");
      end loop;
      New_Line;
   end Display_Array;

end Int_Arrays;
```

Listing 20: test_int_arrays.adb

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

with Int_Arrays;  use Int_Arrays;

procedure Test_Int_Arrays is
   A : Integer_Arr := (0, 0, 1);
begin
   Display_Array (A, "A");
   Put_Line ("Is_Zero: "
             & Boolean'Image (Is_Zero (A)));
   Put_Line ("Has_Zero: "
             & Boolean'Image (Has_Zero (A)));

   A := (0, 0, 0);

   Display_Array (A, "A");
   Put_Line ("Is_Zero: "
             & Boolean'Image (Is_Zero (A)));
   Put_Line ("Has_Zero: "
             & Boolean'Image (Has_Zero (A)));
end Test_Int_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Quantified_Expression.
↪Quantified_Expression_2
MD5: a957a8fd60e1849248efe1a84eae6afa
```

Runtime output

```
A:  0  0  1
Is_Zero: FALSE
Has_Zero: TRUE
A:  0  0  0
Is_Zero: TRUE
Has_Zero: TRUE
```

So far, we've seen quantified expressions using indices — e.g. `for all I in A'Range => ...`. We could avoid indices in quantified expressions by simply using the `E of A` form. In this case, we can just write `for all E of A => ...`. Let's adapt the implementation of `Is_Zero` and `Has_Zero` using this form:

Listing 21: int_arrays.ads

```
package Int_Arrays is

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

(continues on next page)

(continued from previous page)

```

   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Quantified_Expression.
↪Quantified_Expression_3
MD5: 059d12a6529483ebcc5db23dc6262896
```

Here, we're checking the components E of the array A and comparing them against zero.

In the Ada Reference Manual

- 4.5.8 Quantified Expressions[178]

9.4 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

```
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

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Simple_
↪Declare_Expression
MD5: c4773c3749eea045ac5db147fbac594b
```

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:

[178] http://www.ada-auth.org/standards/22rm/html/RM-4-5-8.html

Listing 23: integer_arrays.ads

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

with Integer_Arrays; use Integer_Arrays;

procedure Show_Integer_Arrays is
   Arr : constant Integer_Array := [1, 2, 3];
begin
   Put_Line ("Sum: "
             & Sum (Arr)'Image);
   Put_Line ("Avg: "
             & Avg (Arr)'Image);
end Show_Integer_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Integer_
↪Arrays
MD5: 30a035038508549822c819b60638133d
```

Runtime output

```
Sum:  6
Avg:  2.00000E+00
```

In this example, the Avg function is implemented using a declare expression. In this expression, A renames the Arr array, and S is a constant initialized with the value returned by the Sum function.

In the Ada Reference Manual

- 4.5.9 Declare Expressions[179]

9.4.1 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

```ada
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)
                 return Float is
     (declare
         A  : Integer_Array renames Arr;

         S1 : aliased constant Integer := Sum (A);
```

(continues on next page)

[179] http://www.ada-auth.org/standards/22rm/html/RM-4-5-9.html

(continued from previous page)

```
         --  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;
```

Listing 27: integer_arrays.adb

```
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;

   function Sum (Arr : Integer_Array)
                 return Integer_Sum is
     (Integer_Sum (Integer'(Sum (Arr))));

end Integer_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Integer_
  ↪Arrays_Error
MD5: ea38f5067c849b85685d70ffc386f7a7
```

Build output

```
integer_arrays.ads:26:10: error: "aliased" not allowed in declare_expression
integer_arrays.ads:29:10: error: object renaming or constant declaration expected
integer_arrays.ads:30:10: error: object renaming or constant declaration expected
integer_arrays.ads:33:10: error: object renaming or constant declaration expected
integer_arrays.ads:38:19: error: "Unchecked_Access" attribute cannot occur in a␣
  ↪declare_expression
```

(continues on next page)

(continued from previous page)

```
integer_arrays.ads:42:15: error: anonymous access type not allowed in declare_
↪expression
gprbuild: *** compilation phase failed
```

In this version of the Avg function, we see many errors in the declarative part of the declare expression. If we convert the declare expression into an actual function implementation, however, those declarations won't trigger compilation errors. (Feel free to try this out!)

9.5 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 * 3 * 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 *) 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 an the initial value, we get the reduction expression: 0 + 2 + 3 + 4 = 9. This can be implemented using an array:

Listing 28: show_reduction_expression.adb

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

procedure Show_Reduction_Expression is
   A : array (1 .. 3) of Integer;
   I : Integer;
begin
   A := [2, 3, 4];
   I := A'Reduce ("+", 0);

   Put_Line ("A = "
             & A'Image);
   Put_Line ("I = "
             & I'Image);
end Show_Reduction_Expression;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↪Simple_Reduction_Expression
MD5: 63c85aeff33e9ab3bf37bcb62559e0b2
```

Runtime output

```
A =
[ 2,  3,  4]
I =  9
```

Here, we have the array A with a list of values. The A'Reduce ("+", 0) expression reduces the list of values of A into a single value — in this case, an integer value that is stored in I. This statement is equivalent to:

```
I := 0;
for E of A loop
   I := I + E;
end loop;
```

Naturally, we can reduce the array using the * operator:

Listing 29: show_reduction_expression.adb

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

procedure Show_Reduction_Expression is
   A : array (1 .. 3) of Integer;
   I : Integer;
begin
   A := [2, 3, 4];
   I := A'Reduce ("*", 1);

   Put_Line ("A = "
             & A'Image);
   Put_Line ("I = "
             & I'Image);
end Show_Reduction_Expression;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↪Simple_Reduction_Expression
MD5: 98e1de10863eed4bd12cc6ab1d7ce7ef
```

Runtime output

```
A =
[ 2,  3,  4]
I =  24
```

In this example, we call A'Reduce ("*", 1) to reduce the list. (Note that we use an initial value of one because it is the identity element[180] of a multiplication, so the complete operation is: 1 * 2 * 3 * 4 = 24.)

In the Ada Reference Manual

- Reduction Expressions[181]

9.5.1 Value sequences

In addition to arrays, we can apply reduction expression to value sequences, which consist of an iterated element association — for example, [for I in 1 .. 3 => I + 1]. We can simply *append* the reduction expression to a value sequence:

[180] https://en.wikipedia.org/wiki/Identity_element
[181] http://www.ada-auth.org/standards/22rm/html/RM-4-5-10.html

Listing 30: show_reduction_expression.adb

```
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

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↪Reduction_Expression_Value_Sequences
MD5: 25b75869e53aa3c8a8f8c821a05718c5
```

Runtime output

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

In this example, we create the value sequence `[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.)

9.5.2 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

```
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);

```

(continues on next page)

(continued from previous page)

```
   Put_Line ("A = "
             & A'Image);
   Put_Line ("I = "
             & I'Image);
end Show_Reduction_Expression;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↪Custom_Reducer_Procedure
MD5: 1ed7cd1f3f5d5b8acda36b04afa955f0
```

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:

```
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

```
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
```

(continues on next page)

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↪Custom_Reducer_Function
MD5: 3bfc9b59e4667490e40921770990f52b
```

Runtime output

```
A =
[ 2,  3,  4]
I =  9
```

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

9.5.3 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

```
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
```

(continues on next page)

(continued from previous page)

```
     (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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↪Reducer_Integer_Accumulator
MD5: 95d61e18e7b719d0a25dc35cdbff6af2
```

Runtime output

```
Acc =
(VALUE =>  9,
 COUNT =>  3)
Avg =  3.00000E+00
```

In this example, we're using the Integer_Accumulator record type in our reducer — the Accumulate function. In this case, we're not only accumulating the values, but also counting the number of elements in the list. (Of course, we could have used A'Length for that as well.)

Also, we're not limited to numeric types: we can also create a reducer using strings as the accumulator type. In fact, we can display the initial value and the elements of the list by using unbounded strings:

Listing 34: show_reduction_expression.adb

```
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;

```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↪Reducer_String_Accumulator
MD5: 557416f08f28a48110c0fa6909086629
```

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

TEN

STATEMENTS

10.1 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:

Category	Examples
Simple statements	Null statement, assignment, subprogram call, etc.
Compound statements	If statement, case statement, loop statement, block statement

In the Ada Reference Manual

- 5.1 Simple and Compound Statements - Sequences of Statements[182]

10.2 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

```
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

182 http://www.ada-auth.org/standards/22rm/html/RM-5-1.html

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Simple_Labels
MD5: 820f5963b476af5c04314fd4373d2286
```

Runtime output

```
Hello World!
This is a test.
====================
```

Here, we're labeling each statement. For example, we use the `Show_Hello` label to identify the `Put_Line ("Hello World!");` statement. Note that we can use multiple labels a single statement. In this code example, we use the `Show_Separator` and `Show_Block_Separator` labels for the same statement.

In the Ada Reference Manual

- 5.1 Simple and Compound Statements - Sequences of Statements[183]

10.2.1 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

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Label_Goto
MD5: 0ce06582bbefae818d4da3b7d2d3436b
```

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

10.2.2 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
```

(continues on next page)

183 http://www.ada-auth.org/standards/22rm/html/RM-5-1.html

(continued from previous page)

```
     (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

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Label_Continue_1
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 not Is_Further_Processing_Needed (E) then
         goto Continue;
      end if;

      --  Do more stuff...

      <<Continue>>
   end loop;
```

(continues on next page)

(continued from previous page)

```
end Show_Continue;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Label_Continue_2
MD5: 260b52ead782adf76eee5cf3c4e8332b
```

Here, we use a `Continue` label at the end of the loop and jump to it in the case that no further processing is needed. Note that, in this example, we don't have a statement after the `Continue` label because the label itself is at the end of a statement — to be more specific, at the end of the loop statement. In such cases, there's an implicit **null** statement.

Historically

Since Ada 2012, we can simply write:

```
loop
   --  Some statements...

   <<Continue>>
end loop;
```

If a label is used at the end of a sequence of statements, a **null** statement is implied. In previous versions of Ada, however, that is not the case. Therefore, when using those versions of the language, we must write at least a **null** statement:

```
loop
   --  Some statements...

   <<Continue>> null;
end loop;
```

10.2.3 Labels and compound statements

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

Listing 5: show_statement_identifier.adb

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

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

   Arr   : constant array (1 .. 5) of Integer :=
             (1, 4, 6, 42, 49);
   Found : Boolean := False;
begin
   <<Find_42>> for E of Arr loop
      if E = 42 then
         Found := True;
         exit;
      end if;
   end loop;

   Put_Line ("Found: " & Found'Image);
end Show_Statement_Identifier;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Loop_Label
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**

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Loop_Statement_
↪Identifier
MD5: e52cb5eea9427addf3cabe64dd73bc2d
```

> **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.

10.3 Exit loop statement

We've introduced bare loops back in the Introduction to Ada course[184]. In this section, we'll briefly discuss loop names and exit loop statements.

A bare loop has this form:

```ada
loop
    exit when Some_Condition;
end loop;
```

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

[184] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-bare-loops

```
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

```
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

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Exit_Loop_Statement.Exit_
↪Named_Loop
MD5: b77353f6ed98f8ddb32c73c47d249020
```

Runtime output

```
Vector elements are:
Element:  20
Element:  10
Element:  0
Element:  13
```

Naming a loop is particularly useful when we have nested loops and we want to exit directly from the inner loop:

Listing 8: show_inner_loop_exit.adb

```
procedure Show_Inner_Loop_Exit is
   pragma Warnings (Off);
```

(continues on next page)

(continued from previous page)

```

   Cond : Boolean := True;
begin

   Outer_Processing : loop

      Inner_Processing : loop
         exit Outer_Processing when Cond;
      end loop Inner_Processing;

   end loop Outer_Processing;

end Show_Inner_Loop_Exit;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Exit_Loop_Statement.Inner_
↪Loop_Exit
MD5: b5c7434f1bf23c2cb8f81e4c13a31386
```

Here, we indicate that we exit from the `Outer_Processing` loop in case a condition `Cond` is met, even if we're actually within the inner loop.

In the Ada Reference Manual

- 5.7 Exit Statements[185]

10.4 If, case and loop statements

In the Introduction to Ada course, we talked about if statements[186], loop statements[187], and case statements[188]. 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;
   Increment : Boolean := True;
   Val       : Integer := 0;
begin
   --
   --  If statement
   --
   if Reset then
      Val := 0;
   elsif Increment then
      Val := Val + 1;
   else
      Val := Val - 1;
   end if;
```

(continues on next page)

[185] http://www.ada-auth.org/standards/22rm/html/RM-5-7.html
[186] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-if-statement
[187] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-loop-statement
[188] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-case-statement

(continued from previous page)

```

   --
   --  Loop statement
   --
   for I in 1 .. 5 loop
      Val := Val * 2 - I;
   end loop;

   --
   --  Case statement
   --
   case Val is
      when 0 .. 5 =>
         null;
      when others =>
         Val := 5;
   end case;

end Show_If_Case_Loop_Statements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.If_Case_Loop_Statements.
↪Example
MD5: 4fdc7f00e5218ed59d9eb050339567f1
```

In this section, we'll look into a more advanced detail about the case statement.

In the Ada Reference Manual

- 5.3 If Statements[189]
- 5.4 Case Statements[190]
- 5.5 Loop Statements[191]

10.4.1 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 431), 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

```
package Scales is

   type Satisfaction_Scale is (Very_Dissatisfied,
                               Dissatisfied,
                               OK,
                               Satisfied,
                               Very_Satisfied);
```

(continues on next page)

[189] http://www.ada-auth.org/standards/22rm/html/RM-5-3.html
[190] http://www.ada-auth.org/standards/22rm/html/RM-5-4.html
[191] http://www.ada-auth.org/standards/22rm/html/RM-5-5.html

(continued from previous page)

```

   type Scale is range 0 .. 10;

   function To_Satisfaction_Scale
     (S : Scale)
      return Satisfaction_Scale;

end Scales;
```

Listing 11: scales.adb

```
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

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

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.If_Case_Loop_Statements.Case_
↪Statement_Expression
MD5: 353ff771291e0c994ec052e818f9720c
```

Build output

```
scales.adb:9:07: error: missing case values: -128 .. -1
scales.adb:9:07: error: missing case values: 11 .. 127
gprbuild: *** compilation phase failed
```

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[192], every possible value for the choice expression needs to be covered by a unique branch of the case statement. In principle, it *seems* that we're actually covering all possible values of the `Scale` type, which ranges from 0 to 10. However, we've written `case (S) is` instead of `case S is`. Because of the parentheses, `(S)` is evaluated as an expression. In this case, the expected range of the case statement is not `Scale'Range`, but the range of its *base type* (page 20) `Scale'Base'Range`.

In other languages

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

Listing 13: main.c

```c

#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");
   }
}
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.If_Case_Loop_Statements.
↪Case_Statement_C
MD5: 64ef6b15f1bdf14ca9273964ec5e1755
```

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.

10.5 Block Statements

We've introduced block statements back in the Introduction to Ada course[193]. They have this simple form:

[192] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-case-statement
[193] https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-block-statement

Listing 14: show_block_statement.adb

```
procedure Show_Block_Statement is
   pragma Warnings (Off);
begin

   --  BLOCK STARTS HERE:
   declare
      I : Integer;
   begin
      I := 0;
   end;

end Show_Block_Statement;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Block_Statements.Simple_
↪Block_Statement
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

```
procedure Show_Block_Statement is
   pragma Warnings (Off);
begin

   Simple_Block : declare
      I : Integer;
   begin
      I := 0;
   end Simple_Block;

end Show_Block_Statement;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Block_Statements.Block_
↪Statement_Identifier
MD5: b327b7675931d9b994637671c806c7c3
```

Note that we must write `end Simple_Block;` when we use the `Simple_Block` identifier.

Block statement identifiers are useful:

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

> **In the Ada Reference Manual**
>
> - 5.6 Block Statements[194]

[194] http://www.ada-auth.org/standards/22rm/html/RM-5-6.html

10.6 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

```
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

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Extended_Return_Statements.
↪Simple_Return
MD5: 16e85a8cba869802f912627c40a64c20
```

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:

Listing 17: show_extended_return.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Extended_Return_Statements.
↪Extended_Return
```

(continues on next page)

(continued from previous page)

```
MD5: d6d6edaf800a0e346ff8ede13cbbe100
```

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[195]

10.6.1 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[196].

While the `extended_return_statement` was added to the language specifically to support *limited constructor functions* (page 823), it comes in handy whenever you want a local name for the function result:

Listing 18: show_string_construct.adb

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

procedure Show_String_Construct is

   function Make_String
     (S          : String;
      Prefix     : String;
      Use_Prefix : Boolean) return String
   is
      Length : Natural := S'Length;
   begin
      if Use_Prefix then
         Length := Length + Prefix'Length;
      end if;

      return Result : String (1 .. Length) do

         --  fill in the characters
         if Use_Prefix then
            Result
              (1 .. Prefix'Length) := Prefix;

            Result
              (Prefix'Length + 1 .. Length) := S;
         else
            Result := S;
         end if;

      end return;
   end Make_String;

```

(continues on next page)

[195] http://www.ada-auth.org/standards/22rm/html/RM-6-5.html
[196] https://www.adacore.com/gems/ada-gem-10

(continued from previous page)

```
   S1 : String := "Ada";
   S2 : String := "Make_With_";
begin
   Put_Line ("No prefix:   "
             & Make_String (S1, S2, False));
   Put_Line ("With prefix: "
             & Make_String (S1, S2, True));
end Show_String_Construct;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Statements.Extended_Return_Statements.
↪Extended_Return_Other_Usages
MD5: a2b26ceed06a0ab66aff6c2b59c02003
```

Runtime output

```
No prefix:   Ada
With prefix: Make_With_Ada
```

In this example, we first calculate the length of the string and store it in Length. We then use this information to initialize the return object of the Make_String function.

CHAPTER

ELEVEN

SUBPROGRAMS

11.1 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[197]
- 6.4.1 Parameter Associations[198]

11.1.1 Formal Parameter Modes

We already discussed formal parameter modes in the Introduction to Ada[199] course:

in	Parameter can only be read, not written
out	Parameter can be written to, then read
in out	Parameter can be both read and written

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.

11.1.2 By-copy and by-reference

In the Introduction to Ada[200] 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:

197 http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
198 http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
199 https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#intro-ada-parameter-modes
200 https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#intro-ada-parameter-modes

Listing 1: check_param_passing.ads

```
with System;

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

Listing 2: check_param_passing.adb

```
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

```
with System;

package Machine_X is

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

end Machine_X;
```

Listing 4: machine_x.adb

```
with Check_Param_Passing;

package body Machine_X is

   procedure Update_Value
     (V  : in out Integer;
      AV :        System.Address) is
   begin
      V := V + 1;
      Check_Param_Passing (Formal => V'Address,
                           Actual => AV);
   end Update_Value;

end Machine_X;
```

Listing 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↪Associations.By_Copy_By_Ref_Params
MD5: e437d3432703124496f0a217177959eb
```

Runtime output

```
Formal parameter at 00007FFCF12583FC
Actual parameter at 00007FFCF125841C
Parameter is passed by copy.
```

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 127)).

As we know, when a parameter is passed by copy, it is first copied to a temporary object. In the case of a parameter with **in out** mode, the temporary object is copied back to the original (actual) parameter at the end of the subprogram call. In our example, the temporary object indicated by V is copied back to A at the end of the call to `Update_Value`.

In Ada, it's not the parameter mode that determines whether a parameter is passed by copy or by reference, but rather its type. We can distinguish between three categories:

1. By-copy types;
2. By-reference types;
3. *Unspecified* types.

Obviously, parameters of by-copy types are passed by copy and parameters of by-reference type are passed by reference. However, if a category isn't specified — i.e. when the type is neither a by-copy nor a by-reference type —, the decision is essentially left to the compiler.

As a rule of thumb, we can say that;

- elementary types — and any type that is essentially elementary, such as a private type whose full view is an elementary type — are passed by copy;
- tagged and explicitly limited types — and other types that are essentially tagged, such as task types — are passed by reference.

The following table provides more details:

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

Note that, for parameters of limited types, only those parameters whose type is *explicitly* limited are always passed by reference. We discuss this topic in more details *in another chapter* (page 832).

Let's see an example:

Listing 6: machine_x.ads

```
with System;

package Machine_X is

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

   type Rec is record
      A : Integer;
   end record;

   type Rec_Array is record
      A   : Integer;
      Arr : Integer_Array (1 .. 100);
   end record;

   type Tagged_Rec is tagged record
      A : Integer;
   end record;

   procedure Update_Value
     (R  : in out Rec;
```

(continues on next page)

(continued from previous page)

```
      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
   TR : Tagged_Rec := (A   => 5);
   R  : Rec        := (A   => 5);
   RA : Rec_Array  := (A   => 5,
                       Arr => (others => 0));
```

(continues on next page)

(continued from previous page)

```
begin
   Put_Line ("Tagged record");
   Update_Value (TR, TR'Address);

   Put_Line ("Untagged record");
   Update_Value (R,  R'Address);

   Put_Line ("Untagged record with array");
   Update_Value (RA, RA'Address);
end Show_By_Copy_By_Ref_Params;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↪Associations.By_Copy_By_Ref_Params
MD5: 3ca46380c4df36af9393041181ff2f17
```

Runtime output

```
Tagged record
Formal parameter at 00007FFED4943AF0
Actual parameter at 00007FFED4943AF0
Parameter is passed by reference.
Untagged record
Formal parameter at 00007FFED494393C
Actual parameter at 00007FFED4943AEC
Parameter is passed by copy.
Untagged record with array
Formal parameter at 00007FFED4943950
Actual parameter at 00007FFED4943950
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.)

11.1.3 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

```
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

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

with Check_Param_Passing;

package body Machine_X is

   procedure Update_Value
     (R  : in out Rec;
      AR :        System.Address)
   is
      procedure Show_Vars is
      begin
         Put_Line ("Global_R.A: "
                   & Integer'Image (Global_R.A));
         Put_Line ("R.A:        "
                   & Integer'Image (R.A));
      end Show_Vars;
   begin
      Check_Param_Passing (Formal => R'Address,
                           Actual => AR);

      Put_Line ("Incrementing Global_R.A...");
      Global_R.A := Global_R.A + 1;
      Show_Vars;

      Put_Line ("Incrementing R.A...");
      R.A := R.A + 5;
      Show_Vars;
   end Update_Value;

end Machine_X;
```

Listing 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
```

(continues on next page)

(continued from previous page)

```
begin
   Put_Line ("Calling Update_Value...");
   Update_Value (Global_R,  Global_R'Address);

   Put_Line ("After call to Update_Value...");
   Put_Line ("Global_R.A: "
             & Integer'Image (Global_R.A));
end Show_By_Copy_By_Ref_Params;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↪Associations.By_Copy_By_Ref_Params
MD5: 96be7054b7ff64a304705edf6b15f031
```

Runtime output

```
Calling Update_Value...
Formal parameter at 00007FFF3AAD695C
Actual parameter at 00000000004473BC
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.

11.1.4 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 634) and *aliased parameters* (page 643) 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

```
with System;

package Machine_X is
```

(continues on next page)

(continued from previous page)

```

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

end Machine_X;
```

Listing 13: machine_x.adb

```
with Check_Param_Passing;

package body Machine_X is

   procedure Update_Value
     (V  : aliased in out Integer;
      AV :                System.Address)
   is
   begin
      V := V + 1;
      Check_Param_Passing (Formal => V'Address,
                           Actual => AV);
   end Update_Value;

end Machine_X;
```

Listing 14: show_by_copy_by_ref_params.adb

```
with Machine_X; use Machine_X;

procedure Show_By_Copy_By_Ref_Params is
   A : aliased Integer := 5;
begin
   Update_Value (A, A'Address);
end Show_By_Copy_By_Ref_Params;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↪Associations.By_Copy_By_Ref_Params
MD5: c066af3a7081815d0a7598733f9e6aec
```

Runtime output

```
Formal parameter at 00007FFFC788D9FC
Actual parameter at 00007FFFC788D9FC
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

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
 ↪Associations.By_Copy_By_Ref_Params
MD5: 9e6586e0b771de68040131cae81799b8
```

Build output

```
show_by_copy_by_ref_params.adb:6:18: error: actual for aliased formal "V" must be␣
 ↪aliased object
gprbuild: *** compilation phase failed
```

Again, we discuss more about *aliased parameters* (page 643) and *aliased objects* (page 636) later on in the context of access types.

11.1.5 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[201]

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;
```

Listing 17: operations.adb

```ada
package body Operations is

   procedure Add (Left  : in out Integer;
                  Right :        Float := 1.0) is
   begin
      Left := Left + Integer (Right);
   end Add;

end Operations;
```

Listing 18: show_param_association.adb

```ada
with Operations; use Operations;

```

(continues on next page)

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

(continued from previous page)

```
procedure Show_Param_Association is
   A : Integer := 5;
begin
   --  Positional association
   Add (A, 2.0);

   --  Positional association
   --  (using default value)
   Add (A);

   --  Named association
   Add (Left  => A,
        Right => 2.0);

   --  Named association (inversed order)
   Add (Right => 2.0,
        Left  => A);

   --  Mixed positional / named association
   Add (A, Right => 2.0);
end Show_Param_Association;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↪Associations.Param_Association_1
MD5: 64d3f44ac2bf72317fae22658f6d218e
```

This code snippet has examples of positional and name parameter association. Also, it has an example of mixed positional / named parameter association. In most cases, the actual A parameter is associated with the formal `Left` parameter, and the actual 2.0 parameter is associated with the formal `Right` parameter.

In addition to that, parameters can have default values, so, when we write `Add (A)`, the A variable is associated with the `Left` parameter and the default value (1.0) is associated with the `Right` parameter.

Also, when we use named parameter association, the parameter order is irrelevant: we can, for example, write the last parameter as the first one. Therefore, we can write `Add (Right => 2.0, Left => A)` instead of `Add (Left => A, Right => 2.0)`.

Ambiguous calls

Ambiguous calls can be detected by the compiler during parameter association. For example, when we have both default values in parameters and subprogram overloading, the compiler might be unable to decide which subprogram we're calling:

Listing 19: operations.ads

```
package Operations is

   procedure Add (Left  : in out Integer);

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

end Operations;
```

Listing 20: operations.adb

```
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;
```

Listing 21: show_param_association.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↪Associations.Param_Association_1
MD5: 2725517f82d4068b669028eca1815079
```

Build output

```
show_param_association.adb:6:04: error: ambiguous expression (cannot resolve "Add")
show_param_association.adb:6:04: error: possible interpretation at operations.ads:5
show_param_association.adb:6:04: error: possible interpretation at operations.ads:3
gprbuild: *** compilation phase failed
```

As we see in this example, the Add procedure is overloaded. The first instance has one parameter, and the second instance has two parameters, where the second parameter has a default value. When we call Add with just one parameter, the compiler cannot decide whether we intend to call

- the first instance of Add with one parameter

or

- the second instance of Add using the default value for the second parameter.

In this specific case, there are multiple options to solve the issue, but all of them involve redesigning the package specification:

- we could just rename one of Add procedures (thereby eliminating the subprogram overloading);
- we could rename the first parameter of one of the Add procedures and use named parameter association in the call to the procedure;
  - For example, we could rename the parameter to Value and call `Add (Value => A)`.

- remove the default value from the second parameter of the second instance of Add.

Overlapping actual parameters

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

Listing 22: machine_x.ads

```ada
package Machine_X is

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

end Machine_X;
```

Listing 23: machine_x.adb

```ada
package body Machine_X is

   procedure Update_Value (V1 : in out Integer;
                           V2 :    out Integer) is
   begin
      V1 := V1 + 1;
      V2 := V2 + 1;
   end Update_Value;

end Machine_X;
```

Listing 24: show_by_copy_by_ref_params.adb

```ada
with Machine_X; use Machine_X;

procedure Show_By_Copy_By_Ref_Params is
   A : Integer := 5;
begin
   Update_Value (A, A);
end Show_By_Copy_By_Ref_Params;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↪Associations.Illegal_Calls
MD5: d18a7056463fee9298dd1fdef0a31daf
```

Build output

```
show_by_copy_by_ref_params.adb:6:18: error: writable actual for "V1" overlaps with␣
↪actual for "V2"
gprbuild: *** compilation phase failed
```

In this case, we're using 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.

11.2 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 `A := A + B + C;` when we want to add three integer variables. A hypothetical, non-intuitive version of this operation could be `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:

Category	Operators
Logical	**and**, **or**, **xor**
Relational	=, /=, <, <=, >, >=
Unary adding	+, -
Binary adding	+, -, &
Multiplying	*, /, **mod**, **rem**
Highest precedence	**, **abs**, **not**

In the Ada Reference Manual

- 4.5 Operators and Expression Evaluation[202]

11.2.1 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:

Listing 25: integer_arrays.ads

```
package Integer_Arrays is

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

end Integer_Arrays;
```

Listing 26: show_array_concatenation.adb

```
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;

```

(continues on next page)

202 http://www.ada-auth.org/standards/22rm/html/RM-4-5.html

(continued from previous page)

```
   for E of R loop
      Put (E'Image & ' ');
   end loop;
   New_Line;
end Show_Array_Concatenation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Integer_Arrays_
↪Concat
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:

Listing 27: integer_arrays.ads

```
package Integer_Arrays is

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

end Integer_Arrays;
```

Listing 28: 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

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Integer_Arrays_
↪Addition
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:

```
(¦) :: Int -> Int -> Int
a ¦ b = a + a + b

main = putStrLn $ show (2 ¦ 3)
```

This is not possible in Ada.

Let's define a custom addition operator that adds individual components of the Integer_Array type:

Listing 29: integer_arrays.ads

```ada
package Integer_Arrays is

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

   function "+" (Left, Right : Integer_Array)
                 return Integer_Array
     with Post =>
       (for all I in "+"'Result'Range =>
          "+"'Result (I) = Left (I) + Right (I));

end Integer_Arrays;
```

Listing 30: integer_arrays.adb

```ada
package body Integer_Arrays is

   function "+" (Left, Right : Integer_Array)
                 return Integer_Array
   is
      R : Integer_Array (Left'Range);
   begin
      for I in Left'Range loop
         R (I) := Left (I) + Right (I);
      end loop;

      return R;
   end "+";

end Integer_Arrays;
```

Listing 31: show_array_addition.adb

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

(continues on next page)

(continued from previous page)

```
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

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Integer_Arrays_
↪Addition
MD5: 6f50fa47270d97d3fb50379b6275777d
```

Runtime output

```
 7  9  11  13  15
```

Now, the `R := A + B` line doesn't trigger a compilation error anymore because the `+` operator is defined for the `Integer_Array` type.

In the implementation of the `+`, we return an array with the range of the `Left` array where each component is the sum of the `Left` and `Right` arrays. In the declaration of the `+` operator, we're defining the expected behavior in the postcondition. Here, we're saying that, for each index of the resulting array (`for all I in "+"'Result'Range`), the value of each component of the resulting array at that specific index is the sum of the components from the `Left` and `Right` arrays at the same index (`"+"'Result (I) = Left (I) + Right (I)`). (`for all` denotes a *quantified expression* (page 436).)

Note that, in this implementation, we assume that the range of `Right` is a subset of the range of `Left`. If that is not the case, the `Constraint_Error` exception will be raised at runtime in the loop. (You can test this by declaring `B` as `Integer_Array (5 .. 10)`, for example.)

We can also define custom operators for record types. For example, we could declare two `+` operators for a record containing the name and address of a person:

Listing 32: addresses.ads

```
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

```

(continues on next page)

(continued from previous page)

```
   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

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

package body Addresses is

   function "+" (Name    : String;
                 Address : String)
                 return Person
   is
   begin
      return (Name    =>
                Head (Name,
                      Name_String'Length),
              Address =>
                Head (Address,
                      Address_String'Length));
   end "+";

   function "+" (Left, Right : Person)
                 return Person
   is
   begin
      return (Name    => Left.Name,
              Address => Right.Address);
   end "+";

   procedure Display (P : Person) is
   begin
      Put_Line ("Name:    " & P.Name);
      Put_Line ("Address: " & P.Address);
      New_Line;
   end Display;

end Addresses;
```

Listing 34: show_address_addition.adb

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

procedure Show_Address_Addition is
   John : Person := "John" + "4 Main Street";
   Jane : Person := "Jane" + "7 High Street";
begin
   Display (John);
   Display (Jane);
   Put_Line ("----------------");

```

(continues on next page)

(continued from previous page)

```
   Jane := Jane + John;
   Display (Jane);
end Show_Address_Addition;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Rec_Operator
MD5: c69ff43ed5a80a0c62bad87eada14301
```

Runtime output

```
Name:    John
Address: 4 Main Street

Name:    Jane
Address: 7 High Street

----------------
Name:    Jane
Address: 4 Main Street
```

In this example, the first + operator takes two strings — with the name and address of a person — and returns an object of Person type. We use this operator to initialize the John and Jane variables.

The second + operator in this example brings two people together. Here, the person on the left side of the + operator moves to the home of the person on the right side. In this specific case, Jane is moving to John's house.

As a small remark, we usually expect that the + operator is commutative. In other words, changing the order of the elements in the operation doesn't change the result. However, in our definition above, this is *not* the case, as we can confirm by comparing the operation in both orders:

Listing 35: show_address_addition.adb

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

procedure Show_Address_Addition is
   John : constant Person :=
            "John" + "4 Main Street";
   Jane : constant Person :=
            "Jane" + "7 High Street";
begin
   if Jane + John = John + Jane then
      Put_Line ("It's commutative!");
   else
      Put_Line ("It's not commutative!");
   end if;
end Show_Address_Addition;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Rec_Operator
MD5: 2af6e1a31100a1d0fa786d42cc93c09b
```

Runtime output

```
It's not commutative!
```

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[203]

11.3 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_
↪Expression_Function_1
MD5: 44779999566f764279e1c2f292226f95
```

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

```
package Expr_Func is

   function Is_Zero (I : Integer)
                     return Boolean;

end Expr_Func;
```

Listing 38: expr_func.adb

```
package body Expr_Func is

   function Is_Zero (I : Integer)
                     return Boolean is
```

(continues on next page)

[203] http://www.ada-auth.org/standards/22rm/html/RM-6-1.html

(continued from previous page)

```
   begin
      return I = 0;
   end Is_Zero;

end Expr_Func;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_
↪Expression_Function_2
MD5: 4d90b1c63928cbaf9c86a6cc6421bb61
```

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_
↪Expression_Function_3
MD5: 919f9c101b3224006e1302130eba8dd2
```

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)
```

(continues on next page)

(continued from previous page)

```
                    return Boolean is
    (I = 0);

end Expr_Func;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_
↪Expression_Function_4
MD5: 491a491da92636a35579f870969aaf08
```

In addition, we can use expression functions in the private part of a package specification. For example, the following code declares the Is_Valid function in the specification of the My_Data package, while its implementation is an expression function in the private part of the package specification:

Listing 42: my_data.ads

```
package My_Data is

   type Data is private;

   function Is_Valid (D : Data)
                      return Boolean;

private

   type Data is record
      Valid : Boolean;
   end record;

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

end My_Data;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.
↪Private_Expression_Function_1
MD5: beb57eca67b3954097e0f7ac00ea70c9
```

Naturally, we could write the function implementation in the package body instead:

Listing 43: my_data.ads

```
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

```
package body My_Data is

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

end My_Data;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.
↪Private_Expression_Function_2
MD5: 3c6e2a3c53c7c8e1a7b86efccdc3bf8d
```

In the Ada Reference Manual

- 6.8 Expression functions[204]

11.4 Overloading

Note

This section was originally written by Robert A. Duff and published as Gem #50: Overload Resolution[205].

Ada allows overloading of subprograms, which means that two or more subprogram declarations with the same name can be visible at the same place. Here, "name" can refer to operator symbols, like "+". Ada also allows overloading of various other notations, such as literals and aggregates.

In most languages that support overloading, overload resolution is done "bottom up" — that is, information flows from inner constructs to outer constructs. As usual, computer folks draw their trees upside-down, with the root at the top. For example, if we have two procedures Print:

Listing 45: show_overloading.adb

```
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
```

(continues on next page)

204 http://www.ada-auth.org/standards/22rm/html/RM-6-8.html
205 https://www.adacore.com/gems/gem-50

(continued from previous page)

```

   --  Compiler selects Print (S : Sequence)
   Print (X);
end Show_Overloading;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
MD5: 020c4f04285c80c1050d8edbaf2dbcae
```

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

```
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

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
MD5: 3b776a3efdee3d7e583ddbf5159c9a1b
```

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:

Listing 47: show_overloading_error.adb

```
procedure Show_Overloading_Error is

   package Types is
```

(continues on next page)

(continued from previous page)

```
      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;
begin
   Print (Empty);  -- Illegal!
end Show_Overloading_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
MD5: 5182c517a1afff4568ab2404ac66fda8
```

Build output

```
show_overloading_error.adb:24:04: error: ambiguous expression (cannot resolve
↪"Print")
show_overloading_error.adb:24:04: error: possible interpretation at line 16
show_overloading_error.adb:24:04: error: possible interpretation at line 13
show_overloading_error.adb:24:11: error: ambiguous call to "Empty"
show_overloading_error.adb:24:11: error: interpretation at line 10
show_overloading_error.adb:24:11: error: interpretation at line 7
gprbuild: *** compilation phase failed
```

The call is ambiguous, and therefore illegal, because there are two possible meanings. One way to resolve the ambiguity is to use a qualified expression to say which type we mean:

```
Print (Sequence'(Empty));
```

Note that we're now using both bottom-up and top-down overload resolution: `Sequence'` determines which `Empty` is meant (top down) and which `Print` is meant (bottom up). You can qualify an expression, even if it is not ambiguous according to Ada rules — you might want to clarify the type because it might be ambiguous for human readers.

Of course, you could instead resolve the `Print (Empty)` example by modifying the source code so the names are unique, as in the earlier examples. That might well be the best solution, assuming you can modify the relevant sources. Too much overloading can be confusing. How much is "too much" is in part a matter of taste.

Ada really needs to have top-down overload resolution, in order to resolve literals. In some languages, you can tell the type of a literal by looking at it, for example appending L (letter 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

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Literal_
↪Resolution
MD5: f428b6b4c642c44ede6bc21e7522c532
```

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

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Literal_
↪Resolution_Error
MD5: 4789d8eea9b82649ba8e453bb861688a
```

Build output

```
show_literal_resolution_error.adb:9:04: error: ambiguous expression (cannot␣
↪resolve "Slice")
show_literal_resolution_error.adb:9:04: error: possible interpretation at line 7
show_literal_resolution_error.adb:9:04: error: possible interpretation at line 4
gprbuild: *** compilation phase failed
```

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.

Listing 50: show_record_resolution_error.adb

```
1  procedure Show_Record_Resolution_Error is
2
3     type Complex is record
4        Re, Im : Float;
5     end record;
6
7     procedure Grind (X : Complex) is null;
8     procedure Grind (X : String) is null;
9  begin
10    Grind (X => (Re => 1.0, Im => 1.0));
11    --   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
12    --   Illegal!
13 end Show_Record_Resolution_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Record_
 ↪Resolution_Error
MD5: e3dd1f1d0c403bcf672f4bab881b8ef9
```

Build output

```
show_record_resolution_error.adb:10:04: error: ambiguous expression (cannot␣
 ↪resolve "Grind")
show_record_resolution_error.adb:10:04: error: possible interpretation at line 8
show_record_resolution_error.adb:10:04: error: possible interpretation at line 7
gprbuild: *** compilation phase failed
```

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.

11.5 Operator Overloading

We've seen *previously* (page 478) that we can define custom operators for any type. We've also seen that subprograms can be *overloaded* (page 487). 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:

```
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[206]
- G.1.1 Complex Types[207]

11.6 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:

Listing 51: fixed_point.ads

```
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

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

procedure Show_Sat_Op is
   A, B, C : TQ31;
begin
```

(continues on next page)

[206] http://www.ada-auth.org/standards/22rm/html/RM-6-6.html
[207] http://www.ada-auth.org/standards/22rm/html/RM-G-1-1.html

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operator_Overriding.Fixed_Point_Exception
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

```
package Fixed_Point is

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

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

end Fixed_Point;
```

Listing 54: fixed_point.adb

```
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
```

(continues on next page)

(continued from previous page)

```
      Res := L + R;

      if Res > TQ31_2 (TQ31'Last) then
         return TQ31'Last;
      elsif Res < TQ31_2 (TQ31'First) then
         return TQ31'First;
      else
         return TQ31 (Res);
      end if;
   end "+";

end Fixed_Point;
```

Listing 55: show_sat_op.adb

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

procedure Show_Sat_Op is
   A, B, C : TQ31;
begin
   A := TQ31'Last;
   B := TQ31'Last;
   C := A + B;

   Put_Line (A'Image   & " + "
             & B'Image & " = "
             & C'Image);

   A := TQ31'First;
   B := TQ31'First;
   C := A + B;

   Put_Line (A'Image   & " + "
             & B'Image & " = "
             & C'Image);

end Show_Sat_Op;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operator_Overriding.Fixed_
↪Point_Operator_Overloading
MD5: 6317bcf9c278c01f86dbdcb761d86240
```

Runtime output

```
 0.9999999995 +  0.9999999995 =  0.9999999995
-1.0000000000 + -1.0000000000 = -1.0000000000
```

In the implementation of the overridden + operator of the TQ31 type, we declare another type (TQ31_2) with a wider range than TQ31. We use variables of the TQ31_2 type to perform the actual addition, and then we verify whether the result is still in TQ31's range. If it is, we simply convert the result *back* to the TQ31 type. Otherwise, we saturate it — using either the first or last value of the TQ31 type.

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).

11.7 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:

Listing 56: servers.ads

```ada
package Servers is

   procedure Run_Server
     with No_Return;

end Servers;
```

Listing 57: servers.adb

```ada
package body Servers is

   procedure Run_Server is
   begin
      pragma Warnings
        (Off,
         "implied return after this statement");
      while True loop
         --  Processing happens here...
         null;
      end loop;
   end Run_Server;

end Servers;
```

Listing 58: show_endless_loop.adb

```ada
with Servers; use Servers;

procedure Show_Endless_Loop is
begin
   Run_Server;
end Show_Endless_Loop;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Nonreturning_Procedures.
↪Server_Proc
MD5: 3f859b6e2aca8e31367658632e84126c
```

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

```
package Loggers is

   Logged_Failure : exception;

   procedure Log_And_Raise (Msg : String)
     with No_Return;

end Loggers;
```

Listing 60: loggers.adb

```
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

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

procedure Show_No_Return_Exception is
   Check_Passed : constant Boolean := False;
begin
   if not Check_Passed then
      Log_And_Raise ("Check failed!");
      Put_Line ("This line will not be reached!");
   end if;
end Show_No_Return_Exception;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Nonreturning_Procedures.Log_
↪Exception
MD5: 10b4933d8c862d14ade54935cbd2b541
```

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:

Listing 62: loggers.ads

```
package Loggers is

```

(continues on next page)

(continued from previous page)

```
   Logged_Failure : exception;

   procedure Log_And_Raise (Msg : String)
     with No_Return;

end Loggers;
```

Listing 63: loggers.adb

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

package body Loggers is

   procedure Log_And_Raise (Msg : String) is
   begin
      Put_Line (Msg);
   end Log_And_Raise;

end Loggers;
```

Listing 64: show_no_return_exception.adb

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

procedure Show_No_Return_Exception is
   Check_Passed : constant Boolean := False;
begin
   if not Check_Passed then
      Log_And_Raise ("Check failed!");
      Put_Line ("This line will not be reached!");
   end if;
end Show_No_Return_Exception;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Nonreturning_Procedures.
↪Erroneous_Log_Exception
MD5: e44fd8df0529dda5749e85b9e300a999
```

Build output

```
show_no_return_exception.adb:9:07: warning: unreachable code [enabled by default]
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[208]

11.8 Inline subprograms

Inlining[209] 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 agressively.

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

```
package Float_Arrays is

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

   function Average (Data : Float_Array)
                     return Float
     with Inline;

end Float_Arrays;
```

Listing 66: float_arrays.adb

```
package body Float_Arrays is

   function Average (Data : Float_Array)
                     return Float
   is
      Total : Float := 0.0;
   begin
      for Value of Data loop
         Total := Total + Value;
      end loop;
      return Total / Float (Data'Length);
   end Average;
```

(continues on next page)

208 http://www.ada-auth.org/standards/22rm/html/RM-6-5-1.html
209 https://en.wikipedia.org/wiki/Inline_expansion

(continued from previous page)

```

end Float_Arrays;
```

Listing 67: compute_average.adb

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

with Float_Arrays; use Float_Arrays;

procedure Compute_Average is
   Values        : constant Float_Array :=
                     (10.0, 11.0, 12.0, 13.0);
   Average_Value : Float;
begin
   Average_Value := Average (Values);
   Put_Line ("Average = "
             & Float'Image (Average_Value));
end Compute_Average;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Inline_Subprograms.Inlining_
↪Float_Arrays
MD5: 246bc11e8a969d69873f416f583f450e
```

Runtime output

```
Average =  1.15000E+01
```

When compiling this example, the compiler will most probably inline `Average` in the `Compute_Average` procedure. Note, however, that the `Inline` aspect is just a *recommendation* to the compiler. Sometimes, the compiler might not be able to follow this recommendation, so it won't inline the subprogram.

These are some examples of situations where the compiler might not be able to inline a subprogram:

- when the code is too large,
- when it's too complicated — for example, when it involves exception handling —, or
- when it contains tasks, etc.

In the GNAT toolchain

In order to effectively use the `Inline` aspect, we need to set the optimization level to at least `-O1` and use the `-gnatn` switch, which instructs the compiler to take the `Inline` aspect into account.

In addition to the `Inline` aspect, in GNAT, we also have the (implementation-defined) `Inline_Always` aspect. In contrast to the former aspect, however, the `Inline_Always` aspect isn't primarily related to performance. Instead, it should be used when the functionality would be incorrect if inlining was not performed by the compiler. Examples of this are procedures that insert Assembly instructions that only make sense when the procedure is inlined, such as memory barriers.

Similar to the `Inline` aspect, there might be situations where a subprogram has the `Inline_Always` aspect, but the compiler is unable to inline it. In this case, we get a compilation error from GNAT.

Note that we can use the Inline aspect for generic subprograms as well. When we do this, we indicate to the compiler that we wish it inlines all instances of that generic subprogram.

In the Ada Reference Manual

- 6.3.2 Inline Expansion of Subprograms[210]

11.9 Null Procedures

Null procedures are procedures that don't have any effect, as their body is empty. We declare a null procedure by simply writing **is null** in its declaration. For example:

Listing 68: null_procs.ads

```
package Null_Procs is

   procedure Do_Nothing (Msg : String) is null;

end Null_Procs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Null_Proc_1
MD5: a8a801e6c71d8177db61e4aa131b8832
```

As expected, calling a null procedure doesn't have any effect. For example:

Listing 69: show_null_proc.adb

```
with Null_Procs; use Null_Procs;

procedure Show_Null_Proc is
begin
   Do_Nothing ("Hello");
end Show_Null_Proc;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Null_Proc_1
MD5: 274eed0b0952b9aa7e422933ece42d86
```

Null procedures are equivalent to implementing a procedure with a body that only contains **null**. Therefore, the Do_Nothing procedure above is equivalent to this:

Listing 70: null_procs.ads

```
package Null_Procs is

   procedure Do_Nothing (Msg : String);

end Null_Procs;
```

Listing 71: null_procs.adb

```
package body Null_Procs is

   procedure Do_Nothing (Msg : String) is
```

(continues on next page)

210 http://www.ada-auth.org/standards/22rm/html/RM-6-3-2.html

(continued from previous page)

```
   begin
      null;
   end Do_Nothing;

end Null_Procs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Null_Proc_1
MD5: d0c9dc9265ebbaa9603681182dee1d92
```

11.9.1 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

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Simple_
↪Storage_Model
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_Null_Proc is
   S : Storage_Model;
begin
   Put_Line ("Setting 24...");
   Set (S, "24");
   Display (S);
end Show_Null_Proc;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Simple_
↪Storage_Model
MD5: 523b3e7239e683f2d879caa9139106ca
```

Runtime output

```
Setting 24...
```

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;

end Simple_Storage;
```

Listing 76: show_null_proc.adb

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

procedure Show_Null_Proc is
   S : Integer_Storage;
begin
   Put_Line ("Setting 24...");
   Set (S, "24");
   Display (S);
end Show_Null_Proc;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Simple_
↪Storage_Model
MD5: 55d491d1ef72fb7be2bf0d2a212f335b
```

Runtime output

```
Setting 24...
Value:  24
```

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 183) contains an extended example that makes use of null procedures.

In the Ada Reference Manual

- 6.7 Null Procedures[211]

[211] http://www.ada-auth.org/standards/22rm/html/RM-6-7.html

CHAPTER

TWELVE

EXCEPTIONS

12.1 Classification of Errors

When we talk about errors and erroneous behavior in Ada, we can classify them in one of the four categories:

- compilation errors — i.e. errors that an Ada compiler must detect at compilation time;
- runtime errors — i.e. errors that are detected by an Ada-based application using checks at runtime;
- bounded errors;
- erroneous execution.

In this section, we discuss each of these categories.

In the Ada Reference Manual

- 1.1.5 Classification of Errors[212]

12.1.1 Compilation errors

In the category of compilation errors, the goal is to prevent compilers from accepting illegal programs. Here, any program that doesn't follow the rules described in the Ada Reference Manual is considered illegal. Those rules include not only simple syntax errors, but also more complicated semantic rules, such as the ones concerning *accessibility levels* (page 645) for access types.

Note that Ada — in contrast to many programming languages, which can be quite permissive — tries to prevent as many errors as possible at compilation time because of its focus on safety. However, even though a wide range of errors can be detected at compilation time, this doesn't mean that a legal Ada program is free of errors. Therefore, using methods such as static analysis or unit testing is important.

12.1.2 Runtime errors

When a rule cannot be verified at compilation time, a common strategy is to have the compiler insert runtime checks into the resulting application. We see details about these checks later on when we discuss *checks and exceptions* (page 513).

A typical example is an *overflow check* (page 519). Consider a calculation using variables: if this calculation leads to a result that isn't representable with the underlying data types, we cannot possibly store a value into a register or memory that can be considered correct — so we have to detect this situation. Unfortunately, because we're using variables, we

[212] http://www.ada-auth.org/standards/12rm/html/RM-1-1-5.html

obviously cannot verify the result of the calculation at compilation time, so we have to verify it at runtime.

As we've mentioned before, Ada strives for detecting as many erroneous conditions as possible, while other programming language would allow errors such as overflow errors to remain undetected — which would likely lead the application to misbehave. Those checks raise an exception if an erroneous condition is detected, so the programmer has the means — and the responsibility — to catch that exception and handle the situation properly (Note, however, that some of the runtime checks can be deactivated. We will discuss this topic later on.)

12.1.3 Bounded errors

For certain kinds of errors, the compiler might not be able to detect the error — neither at compilation time, nor with checks at runtime. Such errors are called bounded errors because their possible effects are *bounded*. In fact, the Ada Reference Manual describes each bounded error and its possible effects — one of those effects is raising the `Program_Error` exception.

Just as an example, consider the bounded error described in section 13.9.1 Data Validity[213], paragraphs 9:

> If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an invalid representation. It is a bounded error to evaluate the value of such an object. If the error is detected, either `Constraint_Error` or `Program_Error` is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations.

Let's see a code example:

Listing 1: show_bounded_error.adb

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

procedure Show_Bounded_Error is
   subtype Int_1_10 is
     Integer range 1 .. 10;

   I1         : Int_1_10;
   I1_Overlay : Integer
     with Address => I1'Address,
                     Import,
                     Volatile;
begin
   I1_Overlay := 0;
   --  ^^^^^^^^^^^
   --  We use this overlay to write an invalid
   --  value to I1.

   Put_Line ("I1 = " & I1'Image);
   --                  ^^^^^^^^
   --  Bounded error: value in
   --  I1 is out of range.

   I1 := I1 + 1;
   --    ^^
   --  Bounded error: using value
   --  in operation that is out of
```

(continues on next page)

[213] http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html

(continued from previous page)

```
   --  range.

   Put_Line ("I1 = " & I1'Image);
end Show_Bounded_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Classification_Of_Errors.
↪Data_Validity_Bounded_Error
MD5: 770ebb7b6e0015e373e96c0dce250caa
```

Runtime output

```
I1 =  0
I1 =  1
```

In this example, we simulate a missing initialization by using an overlay (`I1_Overlay`). As a consequence, `I1` has an invalid value that is out of the allowed range of the `Int_1_10` subtype. This situation causes two bounded errors:

- a bounded error when `I1` is evaluated in the call to `Image`; and
- a bounded error when the value of the right-sided `I1` is evaluated — in the increment `I1 := I1 + 1`.

In the Ada Reference Manual

- 13.9.1 Data Validity[214]

12.1.4 Erroneous execution

Erroneous execution is similar to bounded errors in the sense that having the compiler detect the erroneous condition at compilation time or at runtime isn't possible. However, unlike bounded errors, the effects are usually nondeterministic: a bound on possible effects is not described by the language.

Again, as an example of erroneous execution, consider the description from section 13.9.1 Data Validity[215], paragraph 12/3, which discusses the implications of using the `Unchecked_Conversion` function. Let's see a code example:

Listing 2: show_erroneous_execution.adb

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

procedure Show_Erroneous_Execution is
   subtype Int_1_10 is
     Integer range 1 .. 10;

   function To_Int_1_10 is new
     Ada.Unchecked_Conversion
       (Source => Integer,
        Target => Int_1_10);

   I1 : Int_1_10 := To_Int_1_10 (0);
   --                ^^^^^^^^^^^^^^^
```

(continues on next page)

214 http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html
215 http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html

(continued from previous page)

```ada
   --  Bounded error
begin
   Put_Line ("I1 = " & I1'Image);

   I1 := I1 + 1;
   --    ^^^^^^
   --  Erroneous execution: using value
   --  in operation that is out of range.

   Put_Line ("I1 = " & I1'Image);
end Show_Erroneous_Execution;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Classification_Of_Errors.Data_Validity_Erroneous_Execution
MD5: 19218e9bb2e153366dea9114a5e59314
```

Build output

```
show_erroneous_execution.adb:8:04: warning: types for unchecked conversion have different sizes [-gnatwz]
```

Runtime output

```
I1 =  0
I1 =  1
```

It is considered to be a bounded error to use the To_Int_1_10 function (based on Unchecked_Conversion) with a value that is invalid for the target data type. However, if we use the invalid value of I1 in an operation such as the I1 := I1 + 1 assignment, this leads to erroneous execution, and the effects are unpredictable: they aren't described in the Ada Reference Manual, as they are nondeterministic.

In the Ada Reference Manual

- 13.9.1 Data Validity[216]

12.2 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:

[216] http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html

Listing 3: show_pragma_assert.adb

```
procedure Show_Pragma_Assert is
   I : constant Integer := 10;

   pragma Assert (I = 10);
begin
   null;
end Show_Pragma_Assert;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Pragma_Assert_1
MD5: 8d40817304515169d0d5670904ca1e01
```

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 4: show_pragma_assert.adb

```
procedure Show_Pragma_Assert is
   I : constant Integer := 11;

   pragma Assert (I = 10, "I is not 10");
begin
   null;
end Show_Pragma_Assert;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Pragma_Assert_2
MD5: b70fa67c92542ade39c388964ce12302
```

Build output

```
show_pragma_assert.adb:4:19: warning: assertion will fail at run time [-gnatw.a]
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : I is not 10
```

Similarly, we can use the procedural form of Assert. For example, the code above can implemented as follows:

Listing 5: show_procedure_assert.adb

```
with Ada.Assertions; use Ada.Assertions;

procedure Show_Procedure_Assert is
   I : constant Integer := 11;

begin
   Assert (I = 10, "I is not 10");
end Show_Procedure_Assert;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Procedure_Assert
MD5: cbab23645ff89d4adffcaaddaeb6f0e3
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : I is not 10
```

Note that a call to `Assert` is simply translated to a check — and the `Assertion_Error` exception from the `Ada.Assertions` package being raised in the case that the check fails. For example, the code above roughly corresponds to this:

Listing 6: 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

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Assertion_Error
MD5: 9c846acf998ca7adabd47c3b5a6ce39f
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : I is not 10
```

In the Ada Reference Manual

- 11.4.2 Pragmas Assert and Assertion_Policy[217]

12.3 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 7: show_pragma_assertion_policy.adb

```ada
procedure Show_Pragma_Assertion_Policy is
   I : constant Integer := 11;

   pragma Assertion_Policy (Assert => Ignore);
begin
   pragma Assert (I = 10);
end Show_Pragma_Assertion_Policy;
```

Code block metadata

[217] http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Pragma_
↪Assertion_Policy_1
MD5: 39b8aa4a34b6169c03b54074f4136519
```

Build output

```
show_pragma_assertion_policy.adb:6:19: warning: assertion would fail at run time [-
↪gnatw.a]
```

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:

Policy	Descripton
`Assert`	Check assertions
`Static_Predicate`	Check static predicates
`Dynamic_Predicate`	Check dynamic predicates
`Pre`	Check pre-conditions
`Pre'Class`	Check pre-condition of classes of tagged types
`Post`	Check post-conditions
`Post'Class`	Check post-condition of classes of tagged types
`Type_Invariant`	Check type invariants
`Type_Invariant'Class`	Check type invariant of classes of tagged types

In the GNAT toolchain

Compilers are free to include policies that go beyond the ones listed above. For example, GNAT includes the following policies — called *assertion kinds* in this context:

- `Assertions`
- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`
- `Debug`
- `Ghost`
- `Initial_Condition`
- `Invariant`
- `Invariant'Class`
- `Loop_Invariant`
- `Loop_Variant`
- `Postcondition`
- `Precondition`
- `Predicate`
- `Refined_Post`
- `Statement_Assertions`
- `Subprogram_Variant`

Also, in addtion 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[218].

You can specify multiple policies in a single call to Assertion_Policy. For example, you can activate all policies by writing:

Listing 8: show_multiple_assertion_policies.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Multiple_
↪Assertion_Policies
MD5: 3abbf97160b755b84cc4f7e652ca5551
```

In the GNAT toolchain

With GNAT, policies can be specified in multiple ways. In addition to calls to Assertion_Policy, you can use configuration pragmas files[219]. You can use these files to specify all pragmas that are relevant to your application, including Assertion_Policy. In addition, you can manage the granularity for those pragmas. For example, you can use a global configuration pragmas file for your complete application, or even different files for each source-code file you have.

Also, by default, all policies listed in the table above are deactivated, i.e. they're all set to Ignore. You can use the command-line switch -gnata to activate them.

Note that the Assert procedure raises an exception independently of the assertion policy (Assertion_Policy (Assert => Ignore)). For example:

Listing 9: show_assert_procedure_policy.adb

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

procedure Show_Assert_Procedure_Policy is
   pragma Assertion_Policy (Assert => Ignore);

   I : constant Integer := 1;
begin
```

(continues on next page)

[218] https://gcc.gnu.org/onlinedocs/gnat_rm/Pragma-Assertion_005fPolicy
[219] https://gcc.gnu.org/onlinedocs/gnat_ugn/The-Configuration-Pragmas-Files#The-Configuration-Pragmas-Files

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Assert_
↪Procedure_Policy
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 Assertion_Policy[220]

12.4 Checks and exceptions

This table shows all language-defined checks and the associated exceptions:

Check	Exception
Access_Check	Constraint_Error
Discriminant_Check	Constraint_Error
Division_Check	Constraint_Error
Index_Check	Constraint_Error
Length_Check	Constraint_Error
Overflow_Check	Constraint_Error
Range_Check	Constraint_Error
Tag_Check	Constraint_Error
Accessibility_Check	Program_Error
Allocation_Check	Program_Error
Elaboration_Check	Program_Error
Program_Error_Check	Program_Error
Storage_Check	Storage_Error
Tasking_Check	Tasking_Error

[220] http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html

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.

12.4.1 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 10: 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

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Access_
 ↪Check
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

```
raised CONSTRAINT_ERROR : show_access_check.adb:7 access check failed
```

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 664).) For example:

Listing 11: show_access_check.adb

```ada
procedure Show_Access_Check is

   type Integer_Access is
     access all Integer;

   type Safe_Integer_Access is
     not null access all Integer;

   AI  : Integer_Access;
   SAI : Safe_Integer_Access := new Integer;

begin
```

(continues on next page)

(continued from previous page)

```
   SAI := Safe_Integer_Access (AI);
end Show_Access_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Access_
↪Check_2
MD5: 47895a404e2a111476cd67f43c12d4b5
```

Build output

```
show_access_check.adb:9:04: warning: variable "AI" is read but never assigned [-
↪gnatwv]
show_access_check.adb:13:32: warning: null value not allowed here [enabled by␣
↪default]
show_access_check.adb:13:32: warning: Constraint_Error will be raised at run time␣
↪[enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_access_check.adb:13 access check failed
```

Here, the value of AI is null (by default), so we cannot assign it to SAI because its type excludes null.

Note that, if we remove the `:= new Integer` assignment from the declaration of SAI, the null exclusion fails in the declaration itself (because the default value of the access type is `null`).

12.4.2 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 12: 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

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Discriminant_Check
MD5: 665ab37962f8f9c129acac543b1eb15d
```

Build output

```
show_discriminant_check.adb:14:09: warning: incorrect value for discriminant "Valid
↪" [enabled by default]
show_discriminant_check.adb:14:09: warning: Constraint_Error will be raised at run␣
↪time [enabled by default]
```

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 13: 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);
   I : Integer;
begin
   I := R.Counter;
end Show_Discriminant_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Discriminant_Check_2
MD5: 440973b0be7c4261ddf3c2211a2c1325
```

Build output

```
show_discriminant_check.adb:15:10: warning: component not present in subtype of
↪"Rec" defined at line 12 [enabled by default]
show_discriminant_check.adb:15:10: warning: Constraint_Error will be raised at run␣
↪time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_discriminant_check.adb:15 discriminant check failed
```

Here, R's discriminant (Valid) is **False**, so we cannot access the Counter component, for it only exists when the Valid discriminant is **True**.

12.4.3 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:

Listing 14: ops.ads

```
package Ops is
   function Div_Op (A, B : Integer)
                    return Integer is
     (A / B);

   function Rem_Op (A, B : Integer)
                    return Integer is
     (A rem B);

   function Mod_Op (A, B : Integer)
                    return Integer is
     (A mod B);
end Ops;
```

Listing 15: show_division_check.adb

```
with Ops; use Ops;

procedure Show_Division_Check is
   I : Integer;
begin
   I := Div_Op (10, 0);
   I := Rem_Op (10, 0);
   I := Mod_Op (10, 0);
end Show_Division_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Division_Check
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.

12.4.4 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:

Listing 16: show_index_check.adb

```
procedure Show_Index_Check is

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

   function Value_Of (A : Integer_Array;
```

(continues on next page)

(continued from previous page)

```
                      I : Integer)
                      return Integer
   is
      type Half_Integer_Array is new
        Integer_Array (A'First ..
                       A'First + A'Length / 2);

      A_2 : Half_Integer_Array := (others => 0);
   begin
      return A_2 (I);
   end Value_Of;

   Arr_1 : Integer_Array (1 .. 10) :=
             (others => 1);

begin
   Arr_1 (10) := Value_Of (Arr_1, 10);

end Show_Index_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Index_
↪Check
MD5: fa791718701c4ac805badf368df9064e
```

Runtime output

```
raised CONSTRAINT_ERROR : show_index_check.adb:16 index check failed
```

The range of A_2 — which is passed as an argument to the Value_Of function — is 1 to 6. However, in that function call, we're trying to access position 10, which is outside A_2 's bounds.

12.4.5 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 17: show_length_check.adb

```
procedure Show_Length_Check is

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

   procedure Assign (To   : out Integer_Array;
                     From :     Integer_Array) is
   begin
      To := From;
   end Assign;

   Arr_1 : Integer_Array (1 .. 10);
   Arr_2 : Integer_Array (1 .. 9) :=
             (others => 1);

begin
   Assign (Arr_1, Arr_2);
end Show_Length_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Length_
↪Check
MD5: a521afd0a46a67d260e8b0bd5f046ce4
```

Runtime output

```
raised CONSTRAINT_ERROR : show_length_check.adb:9 length check failed
```

Here, the length of `Arr_1` is 10, while the length of `Arr_2` is 9, so we cannot assign `Arr_2` (`From` parameter) to `Arr_1` (`To` parameter) in the `Assign` procedure.

12.4.6 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 18: show_overflow_check.adb

```ada
procedure Show_Overflow_Check is
   A, B : Integer;
begin
   A := Integer'Last;
   B := 1;

   A := A + B;
end Show_Overflow_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Overflow_Check
MD5: baa46d9085cbd14863aaa7e24dc7b9cc
```

Build output

```
show_overflow_check.adb:7:11: warning: value not in range of type "Standard.Integer
↪" [enabled by default]
show_overflow_check.adb:7:11: warning: Constraint_Error will be raised at run time␣
↪[enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_overflow_check.adb:7 overflow check failed
```

In this example, A already has the last possible value of the `Integer'Base` range, so increasing it by one causes an overflow error.

12.4.7 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 19: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Range_
↪Check
MD5: 54b1d67d98d97a58d4265a854fcfa992
```

Build output

```
show_range_check.adb:8:09: warning: value not in range of type "Int_1_10" defined␣
↪at line 3 [enabled by default]
show_range_check.adb:8:09: warning: Constraint_Error will be raised at run time␣
↪[enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_range_check.adb:8 range check failed
```

In this example, we're trying to assign 11 to the variable I of the `Int_1_10` subtype, which has a range from 1 to 10. Since 11 is outside that range, the range check fails.

12.4.8 Tag Check

The tag check ensures that the tag of a tagged object matches the expected tag in a dispatching operation. For example:

Listing 20: 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 21: 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
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("A1'Tag: "
             & Ada.Tags.Expanded_Name (A1'Tag));
   Put_Line ("A2'Tag: "
             & Ada.Tags.Expanded_Name (A2'Tag));

   A2 := A1;

end Show_Tag_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Tag_
↪Check
MD5: 5a685be7804200a884649f54c175ee42
```

Runtime output

```
A1'Tag: P.T1
A2'Tag: P.T2

raised CONSTRAINT_ERROR : show_tag_check.adb:17 tag check failed
```

Here, A1 and A2 have different tags:

- A1'Tag = T1'Tag, while
- A2'Tag = T2'Tag.

Since the tags don't match, the tag check fails in the assignment of A1 to A2.

12.4.9 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 645).

Let's look at an example that mixes access types and anonymous access types. Here, we use an anonymous access type in the declaration of A1 and a named access type in the declaration of A2:

Listing 22: p.ads

```
package P is

   type T is tagged null record;
   type T_Class is access all T'Class;

end P;
```

Listing 23: show_accessibility_check.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Accessibility_Check
MD5: 7120d908b55ef576db93e9a15db257f2
```

Build output

```
show_accessibility_check.adb:9:19: warning: accessibility check fails [enabled by␣
↪default]
show_accessibility_check.adb:9:19: warning: Program_Error will be raised at run␣
↪time [enabled by default]
```

Runtime output

```
raised PROGRAM_ERROR : show_accessibility_check.adb:9 accessibility check failed
```

The anonymous type (`access T'Class`), which is used in the declaration of A1, doesn't have the same accessibility level as the T_Class type. Therefore, the accessibility check fails during the `T_Class (A1)` conversion.

We can see the accessibility check failing in this example as well:

Listing 24: show_accessibility_check.adb

```
with P; use P;

procedure Show_Accessibility_Check is

   A : access T'Class := new T;

   procedure P (A : T_Class) is null;

begin
   P (T_Class (A));

end Show_Accessibility_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Accessibility_Check
MD5: 97db82410dd3459249d0e7a97118b7ef
```

Build output

```
show_accessibility_check.adb:10:16: warning: accessibility check fails [enabled by␣
↪default]
show_accessibility_check.adb:10:16: warning: Program_Error will be raised at run␣
↪time [enabled by default]
```

Runtime output

```
raised PROGRAM_ERROR : show_accessibility_check.adb:10 accessibility check failed
```

Again, the check fails in the `T_Class (A)` conversion and raises a `Program_Error` exception.

12.4.10 Allocation Check

The allocation check ensures, when a task is about to be created, that its master has not been completed. Also, it ensures that the finalization has not started.

This is an example adapted from AI-00280[221]:

Listing 25: p.ads

```
with Ada.Finalization;
with Ada.Unchecked_Deallocation;

package P is
   type T1 is new
     Ada.Finalization.Controlled with null record;
   procedure Finalize (X : in out T1);

   type T2 is new
     Ada.Finalization.Controlled with null record;
   procedure Finalize (X : in out T2);

   X1 : T1;

   type T2_Ref is access T2;
   procedure Free is new
     Ada.Unchecked_Deallocation (T2, T2_Ref);
end P;
```

Listing 26: p.adb

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

package body P is

   procedure Finalize (X : in out T1) is
      X2 : T2_Ref := new T2;
   begin
      Put_Line ("Finalizing T1...");
      Free (X2);
   end Finalize;

   procedure Finalize (X : in out T2) is
   begin
      Put_Line ("Finalizing T2...");
   end Finalize;

end P;
```

221 http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00280.txt?rev=1.12&raw=N

Listing 27: show_allocation_check.adb

```
with P; use P;

procedure Show_Allocation_Check is
   X2 : T2_Ref := new T2;
begin
   Free (X2);
end Show_Allocation_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪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 while the finalization of the master has already started. (Note that X1 was declared in the P package.) This is forbidden, so the allocation check raises a `Program_Error` exception.

12.4.11 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[222]:

Listing 28: p.ads

```
function P return Integer;
```

Listing 29: p.adb

```
function P return Integer is
begin
   return 1;
end P;
```

Listing 30: 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;
```

(continues on next page)

222 http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00064.txt?rev=1.12&raw=N

(continued from previous page)

```
    Z : constant Pointer_To_Func := X;

    --  Renaming-as-body
    function F return Integer renames Z.all;
begin
    null;
end Show_Elaboration_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
  ↪Elaboration_Check
MD5: 80a39df912aae8788296f81ee9d4a79e
```

Build output

```
show_elaboration_check.adb:12:28: warning: cannot call "F" before body seen␣
  ↪[enabled by default]
show_elaboration_check.adb:12:28: warning: Program_Error will be raised at run␣
  ↪time [enabled by default]
```

Runtime output

```
raised PROGRAM_ERROR : show_elaboration_check.adb:12 access before elaboration
```

This is a curious example: first, we declare a function F and assign the value returned by this function to constant Y in its declaration. Then, we declare F as a renamed function, thereby providing a body to F — this is called renaming-as-body. Consequently, the compiler doesn't complain that a body is missing for function F. (If you comment out the function renaming, you'll see that the compiler can then detect the missing body.) Therefore, at runtime, the elaboration check fails because the body of the first declaration of the F function is actually missing.

12.4.12 `Program_Error_Check`

Note

This concept was introduced in Ada 2022.

As we've seen before, there are three checks that may raise a `Program_Error` exception: the `Accessibility_Check`, the `Allocation_Check` and the `Elaboration_Check`. In addition to that, we have the `Program_Error_Check`, which is actually a collection of various different checks that may raise a `Program_Error`, but don't have a category for themselves.

For completeness, these are the error conditions checked by the `Program_Error_Check` (listed in the Action Item (AI) 12-0309 document[223]), according to their definition in the Ada Reference Manual:

[223] http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0309-1.txt?rev=1.5&raw=N

Ada Reference Manual	Paragraph	Description
3.2.4 Subtype Predicates[224]	(29.1	It checks that subtypes with predicates are not used to index an array in generic units.
5.5 Loop Statements[225]	(8.1/!	It checks that the maximum number of chunks for statement-level parallelism is greater than zero.
6.4.1 Parameter Associations[226]	(13.4	It checks that the default value of an out parameter is convertible: an error occurs when we have an out parameter with `Default_Value`, and the actual is a view conversion of an unrelated type that does not have `Default_Value`.
12.5.1 Formal Private and Derived Types[227]	(23.3	It checks that there is no misuse of functions in a generic with a class-wide actual type.
13.3 Operational and Representation Attributes[228]	(75.1	It checks that there are no colliding `External_Tag` values.
B.3.3 Unchecked Union Types[229]	(22/2	It checks that there is no misuse of operations of `Unchecked_Unions` without inferable discriminants.

In the Ada Reference Manual

- 11.5 Suppressing Checks[230]
- 3.2.4 Subtype Predicates[231]
- 5.5 Loop Statements[232]
- 6.4.1 Parameter Associations[233]
- 12.5.1 Formal Private and Derived Types[234]
- 13.3 Operational and Representation Attributes[235]
- B.3.3 Unchecked Union Types[236]

224 http://www.ada-auth.org/standards/22rm/html/RM-3-2-4.html
225 http://www.ada-auth.org/standards/22rm/html/RM-5-5.html
226 http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
227 http://www.ada-auth.org/standards/22rm/html/RM-12-5-1.html
228 http://www.ada-auth.org/standards/22rm/html/RM-13-3.html
229 http://www.ada-auth.org/standards/22rm/html/RM-B-3-3.html
230 http://www.ada-auth.org/standards/22rm/html/RM-11-5.html
231 http://www.ada-auth.org/standards/22rm/html/RM-3-2-4.html
232 http://www.ada-auth.org/standards/22rm/html/RM-5-5.html
233 http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
234 http://www.ada-auth.org/standards/22rm/html/RM-12-5-1.html
235 http://www.ada-auth.org/standards/22rm/html/RM-13-3.html
236 http://www.ada-auth.org/standards/22rm/html/RM-B-3-3.html

Example of a Program_Error_Check

Just as an example, let's look at the check for subtype predicates in generic units:

Listing 31: some_generic_package.ads

```
generic
   type R is (<>);
package Some_Generic_Package is
   procedure Process;
end Some_Generic_Package;
```

Listing 32: some_generic_package.adb

```
package body Some_Generic_Package is

   procedure Process is
      type Arr is
        array (R) of Integer;

      Dummy : Arr := (others => 0);
   begin
      null;
   end Process;

end Some_Generic_Package;
```

Listing 33: show_subtype_predicate_programm_error.adb

```
with Some_Generic_Package;

procedure Show_Subtype_Predicate_Programm_Error is

   type Custom_Range is range 1 .. 5
     with Dynamic_Predicate =>
            4 not in Custom_Range;

   package P is new
     Some_Generic_Package (Custom_Range);
   use P;
begin
   Process;
end Show_Subtype_Predicate_Programm_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Subtype_Predicate_Programm_Error
MD5: b1a5cc579393162dedecb6b65b75eef4
```

Build output

```
show_subtype_predicate_programm_error.adb:9:04: warning: in instantiation at some_
↪generic_package.adb:5 [enabled by default]
show_subtype_predicate_programm_error.adb:9:04: warning: subtype "R" has predicate,
↪ not allowed as index subtype [enabled by default]
show_subtype_predicate_programm_error.adb:9:04: warning: Program_Error will be␣
↪raised at run time [enabled by default]
```

Runtime output

```
raised PROGRAM_ERROR : some_generic_package.adb:5 improper use of generic subtype␣
↪with predicate
```

Here, we're using the `Custom_Range` type for the formal type `R` in the instantiation of the generic package `Some_Generic_Package`. Since we use `R` as an index for the array type `Arr` (in the procedure `Process`), we cannot map a type to `R` that uses a predicate. Therefore, because `Custom_Range` type has a dynamic predicate, the `Program_Error` exception is raised.

12.4.13 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 91):

Listing 34: custom_types.ads

```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;
```

Listing 35: show_storage_check.adb

```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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Storage_Check
MD5: 4e4bd284adb1c1d97f8f7563068c18de
```

Runtime output

```
Allocating RAV1...
Allocating RAV2...

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

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.

12.4.14 Tasking_Check

The Tasking_Check ensures that all tasks have been activated successfully and that no terminated task is called. If the check fails, a Tasking_Error exception is raised.

Note

This concept was introduced in Ada 2022. It was created to group all checks that might raise the Tasking_Error exception.

Let's look at a simple example:

Listing 36: workers.ads

```ada
package Workers is

    task type Worker  is
        entry Start;
    end Worker;

end Workers;
```

Listing 37: workers.adb

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

package body Workers is

    task body Worker  is
    begin
       Put_Line ("Task has started.");
       delay 1.0;
       Put_Line ("Task has finished.");
    end Worker;

end Workers;
```

Listing 38: show_tasking_check_error.adb

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

procedure Show_Tasking_Check_Error is
    W : Worker;
begin
    Put_Line ("W.Start...");
    W.Start;
    Put_Line ("Finished");
end Show_Tasking_Check_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↪Tasking_Check_Error
MD5: 38f9093082d3fe545847ea3d22376e39
```

Build output

```
workers.adb:5:05: warning: no accept for entry "Start" [enabled by default]
```

Runtime output

```
W.Start...
Task has started.
Task has finished.

raised TASKING_ERROR
```

In this example, the body of `Worker` doesn't have an **accept**. Therefore, no rendezvous can happen for the `W.Start` call. Since the task eventually terminates (as you can see in the user messages), the call to `Start` constitutes a call to a terminated task. This condition is checked by the `Tasking_Check`, which fails in this case, thereby raising a `Tasking_Error`.

12.5 Ada.Exceptions package

> **Note**
>
> Parts of this section were originally published as Gem #142 : Exception-ally[237]

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:

```
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.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[238]

[237] https://www.adacore.com/gems/gem-142-exceptions
[238] http://www.ada-auth.org/standards/22rm/html/RM-11-4-1.html

12.5.1 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:

```
exception
   when E : others =>
     Put_Line
       (Ada.Exceptions.Exception_Information (E));
```

In the GNAT toolchain

In the case of GNAT, the information provided by an exception might include the source location where the exception was raised and a nonsymbolic traceback.

You can also retrieve this information individually. Here, you can use:

- the `Exception_Name` functions — and its derivatives `Wide_Exception_Name` and `Wide_Wide_Exception_Name` — to retrieve the name of an exception.
- the `Exception_Message` function to retrieve the message associated with an exception.

Let's see a complete example:

Listing 39: show_exception_info.adb

```
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;
```

12.5.2 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:

Listing 40: exception_tests.ads

```
with Ada.Exceptions; use Ada.Exceptions;

package Exception_Tests is

   Custom_Exception : exception;

   type All_Exception_Occur is
     array (Positive range <>) of
       Exception_Occurrence;

   procedure Test_Exceptions
     (All_Occur  : in out All_Exception_Occur;
      Last_Occur :    out Integer);

end Exception_Tests;
```

Listing 41: exception_tests.adb

```
package body Exception_Tests is

   procedure Save_To_List
     (E          :        Exception_Occurrence;
      All_Occur  : in out All_Exception_Occur;
      Last_Occur : in out Integer)
   is
      L : Integer renames Last_Occur;
      O : All_Exception_Occur renames All_Occur;
   begin
      L := L + 1;
      if L > O'Last then
         raise Constraint_Error
           with "Cannot save occurrence";
      end if;

      Save_Occurrence (Target => O (L),
                       Source => E);
   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,
```

(continues on next page)

(continued from previous page)

```
                              Last_Occur);
      end Nested_1;

      procedure Nested_2 is
      begin
         raise Constraint_Error
           with "Constraint is not correct";
      exception
         when E : others =>
            Save_To_List (E,
                          All_Occur,
                          Last_Occur);
      end Nested_2;

   begin
      Last_Occur := 0;

      Nested_1;
      Nested_2;
   end Test_Exceptions;

end Exception_Tests;
```

Listing 42: 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

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Save_
↪Occurrence
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 43: 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;
```

(continues on next page)

(continued from previous page)

```
      Close (F);
   end Read_Exceptions;

end Exception_Occurrence_Stream;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Exception_
↪Occurrence_Stream
MD5: 3d9f2bd9480aa6dcc250b249b9ef4870
```

Runtime output

```
raised EXCEPTION_OCCURRENCE_STREAM.CUSTOM_EXCEPTION : We got a problem

raised CONSTRAINT_ERROR : Constraint is not correct
```

In this example, we store the exceptions raised in the application in the *exceptions_file.bin* file. In the exception part of procedures Nested_1 and Nested_2, we call Exception_Occurrence'Write to store an exception occurence in the file. In the Read_Exceptions block, we read the exceptions from the the file by calling Exception_Occurrence'Read.

12.5.3 Debugging exceptions in the GNAT toolchain

Here is a typical exception handler that catches all unexpected exceptions in the application:

Listing 44: main.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Exception_
↪Information
MD5: f95068ca90d79b92a7c2031322349153
```

Runtime output

```
raised CONSTRAINT_ERROR : some message
```

The output we get when running the application is not very informative. To get more information, we need to rerun the program in the debugger. To make the session more interesting though, we should add debug information in the executable, which means using the -g switch in the **gnatmake** command.

The session would look like the following (omitting some of the output from the debugger):

```
> rm *.o      # Cleanup previous compilation
> gnatmake -g main.adb
> gdb ./main
(gdb)  catch exception
(gdb)  run
Catchpoint 1, CONSTRAINT_ERROR at 0x0000000000402860 in main.nested () at main.
↪adb:8
8               raise Constraint_Error with "some message";

(gdb) bt
#0  <__gnat_debug_raise_exception> (e=0x62ec40 <constraint_error>) at s-excdeb.
↪adb:43
#1  0x000000000040426f in ada.exceptions.complete_occurrence (x=x@entry=0x637050)
at a-except.adb:934
#2  0x000000000040427b in ada.exceptions.complete_and_propagate_occurrence (
x=x@entry=0x637050) at a-except.adb:943
#3  0x00000000004042d0 in <__gnat_raise_exception> (e=0x62ec40 <constraint_error>,
message=...) at a-except.adb:982
#4  0x0000000000402860 in main.nested ()
#5  0x000000000040287c in main ()
```

And we now know exactly where the exception was raised. But in fact, we could have this information directly when running the application. For this, we need to bind the application with the switch -E, which tells the binder to store exception tracebacks in exception occurrences. Let's recompile and rerun the application.

```
> rm *.o   # Cleanup previous compilation
> gnatmake -g main.adb -bargs -E
> ./main

Exception name: CONSTRAINT_ERROR
Message: some message
Call stack traceback locations:
0x10b7e24d1 0x10b7e24ee 0x10b7e2472
```

The traceback, as is, is not very useful. We now need to use another tool that is bundled with GNAT, called **addr2line**. Here is an example of its use:

```
> 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 0x1000014d1 0x1000014ee 0x100001472
main__nested.2550 (in main) (main.adb:8)
_ada_main (in main) (main.adb:12)
main (in main) + 90
```

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 45: main.adb

```
1  with Ada.Exceptions;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  procedure Main is
5
6     procedure Nested (Index : Integer) is
7        type T_Array is array (1 .. 2) of Integer;
8        T : constant T_Array := (10, 20);
9     begin
10       Put_Line (T (Index)'Img);
11    end Nested;
12
13 begin
14    Nested (3);
15
16 exception
17    when E : others =>
18       Put_Line
19         (Ada.Exceptions.Exception_Information (E));
20 end Main;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Exception_
↪Information
MD5: 3590f2bf48f6ed1cf7745d576924cad4
```

Runtime output

```
raised CONSTRAINT_ERROR : main.adb:10:17 index check failed
index 3 not in 1..2
```

When running the application, we see that the exception information (traceback) is the same as before, but this time the exception message is set automatically by the compiler. So we know we got a Constraint_Error because an incorrect index was used at the named source location (main.adb, line 10). But the significant addition is the second line of the message, which indicates exactly the cause of the error. Here, we wanted to get the element at index 3, in an array whose range of valid indexes is from 1 to 2. (No need for a debugger in this case.)

The column information on the first line of the exception message is also very useful when dealing with null pointers. For instance, a line such as:

```
A := Rec1.Rec2.Rec3.Rec4.all;
```

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.

12.6 Exception renaming

We can rename exceptions by using the an exception renaming declaration in this form `Renamed_Exception : exception renames Existing_Exception;`. For example:

Listing 46: show_exception_renaming.adb

```ada
procedure Show_Exception_Renaming is
   CE : exception renames Constraint_Error;
begin
   raise CE;
end Show_Exception_Renaming;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exception_Renaming.Exception_
↪Renaming
MD5: ff20825162ee9eef6ac8ed329da2a80f
```

Runtime output

```
raised CONSTRAINT_ERROR : show_exception_renaming.adb:4
```

Exception renaming creates a new view of the original exception. If we rename an exception from package A in package B, that exception will become visible in package B. For example:

Listing 47: internal_exceptions.ads

```ada
package Internal_Exceptions is

   Int_E : exception;

end Internal_Exceptions;
```

Listing 48: test_constraints.ads

```ada
with Internal_Exceptions;

package Test_Constraints is

   Ext_E : exception renames
             Internal_Exceptions.Int_E;

end Test_Constraints;
```

Listing 49: 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;

```

(continues on next page)

(continued from previous page)

```
procedure Show_Exception_Renaming_View is
begin
   raise Ext_E;
exception
   when E : others =>
      Put_Line
        (Ada.Exceptions.Exception_Information (E));
end Show_Exception_Renaming_View;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exception_Renaming.Exception_
↪Renaming_View
MD5: a44e2698170c6fab79241d0f33ef8c2e
```

Runtime output

```
raised INTERNAL_EXCEPTIONS.INT_E : show_exception_renaming_view.adb:8
```

Here, we're renaming the Int_E exception in the Test_Constraints package. The Int_E exception isn't directly visible in the Show_Exception_Renaming procedure because we're not **with**ing the Internal_Exceptions package. However, it is indirectly visible in that procedure via the renaming (Ext_E) in the Test_Constraints package.

In the Ada Reference Manual

- 8.5.2 Exception Renaming Declarations[239]

12.7 Out and Uninitialized

Note

This section was originally written by Robert Dewar and published as Gem #150: Out and Uninitialized[240]

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 50: show_out_uninitialized.adb

```
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;
```

(continues on next page)

239 http://www.ada-auth.org/standards/22rm/html/RM-8-5-2.html
240 https://www.adacore.com/gems/gem-150out-and-uninitialized

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
↪Uninitialized_1
MD5: cebcf14e9fd088e38b98a5132d9fd998
```

Runtime output

```
Value for B is 0
```

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:

Listing 51: show_out_uninitialized_warnings.adb

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

procedure Show_Out_Uninitialized_Warnings is

    procedure Local (A : in out Integer) is
    begin
       A := 1;
       raise Program_Error;
    end Local;

   B : Integer := 0;

begin
   Local (B);
exception
   when others =>
      Put_Line ("Value for B is"
                & Integer'Image (B));
end Show_Out_Uninitialized_Warnings;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
 ↪Uninitialized_2
MD5: 5b6960974c729ea37a70fb313d6e5084
```

Build output

```
show_out_uninitialized_warnings.adb:7:10: warning: assignment to pass-by-copy␣
 ↪formal may have no effect [enabled by default]
show_out_uninitialized_warnings.adb:7:10: warning: "raise" statement may result␣
 ↪in abnormal return (RM 6.4.1(17)) [enabled by default]
```

Runtime output

```
Value for B is 0
```

We now get a compilation warning that the pass-by-copy formal may have no effect.

Of course, GNAT is not able to point out all such errors (see first example above), which in general would require full flow analysis.

The behavior is different when using parameter types that the standard mandates be passed by reference, such as tagged types for instance. So the following code will work as expected, updating the actual parameter despite the exception:

Listing 52: show_out_initialized_rec.adb

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
 ↪Uninitialized_3
MD5: 370031a404657ea18ffabf3c1d507cd4
```

Runtime output

```
Value of Field is 1
```

In the GNAT toolchain

It's worth mentioning that GNAT provides a pragma called `Export_Procedure` that forces reference semantics on **out** parameters. Use of this pragma would ensure updates of the actual parameter prior to abnormal completion of the procedure. However, this pragma only applies to library-level procedures, so the examples above have to be rewritten to avoid the use of a nested procedure, and really this pragma is intended mainly for use in interfacing with foreign code. The code below shows an example that ensures that B is set to 1 after the call to Local:

Listing 53: exported_procedures.ads

```
package Exported_Procedures is

   procedure Local (A     : in out Integer;
                    Error : Boolean);
   pragma Export_Procedure
     (Local,
      Mechanism => (A => Reference));

end Exported_Procedures;
```

Listing 54: exported_procedures.adb

```
package body Exported_Procedures is

   procedure Local (A     : in out Integer;
                    Error : Boolean) is
   begin A := 1;
      if Error then
         raise Program_Error;
      end if;
   end Local;

end Exported_Procedures;
```

Listing 55: show_out_reference.adb

```
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

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
↪Uninitialized_4
MD5: aed2788be2b3ceeec19b28421c53fc66
```

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[241]). For instance, the following code makes an incorrect assumption:

```
X := 0;     -- about to try addition
Y := Y + 1; -- see if addition raises exception
X := 1      -- addition succeeded
```

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[242]

12.8 Suppressing checks

12.8.1 pragma Suppress

Note

This section was originally written by Gary Dismukes and published as Gem #63: The Effect of Pragma Suppress[243].

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:

241 http://www.ada-auth.org/standards/22rm/html/RM-11-6.html
242 http://www.ada-auth.org/standards/22rm/html/RM-11-6.html
243 https://www.adacore.com/gems/gem-63

```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;
```

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:

```
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.

12.8.2 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:

Listing 56: show_index_check.adb

```
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);
```

(continues on next page)

(continued from previous page)

```
   --  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);
   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

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Pragma_Unsuppress.Pragma_
↪Unsuppress
MD5: 0585b78fd57913d3172c7ab1ea6f4864
```

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[244]

[244] http://www.ada-auth.org/standards/22rm/html/RM-11-5.html

Part III

Modular programming

CHAPTER

THIRTEEN

PACKAGES

13.1 Package renaming

We've seen in the Introduction to Ada course that we can rename packages[245].

In the Ada Reference Manual

- 10.1.1 Compilation Units - Library Units[246]

13.1.1 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;
```

245 https://learn.adacore.com/courses/intro-to-ada/chapters/modular_programming.html#intro-ada-package-renaming
246 http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html

Listing 5: drivers-m2.ads

```
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: 8d6a6bec32f7ec4397de1faf9f0b44d9
```

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

```
package Driver_M1 is

   Counter_2 : Integer := Drivers.Counter;

end Driver_M1;
```

Listing 7: drivers.ads

```
package Drivers is

   Counter : Integer := 0;

end Drivers;
```

Listing 8: drivers-m1.ads

```
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: d174746d8151d9a2cd048ad44e853850
```

Build output

```
driver_m1.ads:3:27: error: "Drivers" is undefined
gprbuild: *** compilation phase failed
```

As expected, compilation fails here because `Drivers.Counter` isn't visible in `Driver_M1`, even though the renaming (`Drivers.M1`) creates a virtual hierarchy.

13.1.2 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:

```
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:

Listing 9: driver_m1-ext.ads

```
package Driver_M1.Ext is

end Driver_M1.Ext;
```

Listing 10: dummy.adb

```
--  A package called Drivers.M1.Ext is
--  automatically available!

with Drivers.M1.Ext;

procedure Dummy is
begin
   null;
end Dummy;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Package_Renaming.Package_
↪Renaming_1
MD5: e338d668dbd98b1a3917a8d3d948a439
```

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.)

13.1.3 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:

Listing 11: drivers.ads

```
package Drivers
  with Pure is

end Drivers;
```

Listing 12: drivers-m1.ads

```
--  We've converted Driver_M1 to
--  Drivers.M1:

package Drivers.M1 is

end Drivers.M1;
```

Listing 13: drivers-m2.ads

```
--  We've converted Driver_M2 to
--  Drivers.M2:

package Drivers.M2 is

end Drivers.M2;
```

Listing 14: driver_m1.ads

```
--  Original Driver_M1 package still
--  available via package renaming:

with Drivers.M1;

package Driver_M1 renames Drivers.M1;
```

Listing 15: driver_m2.ads

```
--  Original Driver_M2 package still
--  available via package renaming:

with Drivers.M2;

package Driver_M2 renames Drivers.M2;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Package_Renaming.Package_
↪Renaming_2
MD5: 27f8066b5f5954514fea51b6e9b9de81
```

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.

13.2 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[247], we've seen that encapsulation plays an important role in modular programming. By using the private part of a package specification, we can disclose some information, but, at the same time, prevent that this information gets accessed where it shouldn't be used directly. Similarly, we've seen that we can use the private part of a package to distinguish between the *partial and full view* (page 43) of a data type.

[247] https://learn.adacore.com/courses/intro-to-ada/chapters/privacy.html#intro-ada-course-privacy

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.

13.2.1 Declaration and usage

We declare private packages by using the **private** keyword. For example, let's say we have a package named `Data_Processing`:

Listing 16: data_processing.ads

```
package Data_Processing is

--  ...

end Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↪Package_Decl
MD5: 502811212890785d90c6f891d7f8e557
```

We simply write **private package** to declare a private child package named `Calculations`:

Listing 17: data_processing-calculations.ads

```
private package Data_Processing.Calculations is

--  ...

end Data_Processing.Calculations;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↪Package_Decl
MD5: 20df8b2ac4c9aa93f03a12afd9b7ef30
```

Let's see a complete example:

Listing 18: data_processing.ads

```
package Data_Processing is

   type Data is private;

   procedure Process (D : in out Data);

private

   type Data is null record;

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

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↪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.

13.2.2 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

```
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

```
private package Data_Processing.Calculations is

   procedure Calculate (D : in out Data);

end Data_Processing.Calculations;
```

Listing 25: data_processing-advanced_calculations.ads

```
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

```
with Data_Processing.Advanced_Calculations;
use  Data_Processing.Advanced_Calculations;

package body Data_Processing is

   procedure Process (D : in out Data) is
   begin
      Advanced_Calculate (D);
   end Process;

end Data_Processing;
```

Listing 27: data_processing-calculations.adb

```
package body Data_Processing.Calculations is

   procedure Calculate (D : in out Data) is
   begin
      --  Dummy implementation...
      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

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↪Package_2
MD5: 32fc76ae13f1eecdd854a029793034d8
```

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

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↪Package_2
MD5: 27fd3bdb063a11ed7797cc44fa1e8349
```

Build output

```
data_processing-advanced_calculations.ads:1:06: error: current unit must also be␣
↪private descendant of "Data_Processing"
gprbuild: *** compilation phase failed
```

Now, the compilation doesn't work anymore. However, we could still refer to `Calculations` packages in the body of the `Advanced_Calculations` package:

Listing 30: data_processing-advanced_calculations.ads

```
package Data_Processing.Advanced_Calculations is

   procedure Advanced_Calculate (D : in out Data);

end Data_Processing.Advanced_Calculations;
```

Listing 31: data_processing-advanced_calculations.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↪Package_2
MD5: 3f37c129a6994c6b71a25ad17dcb440e
```

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.

13.2.3 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↪Package
MD5: c844327995b28d60c9a79b138a0f21d2
```

Build output

```
test_data_processing.adb:3:06: error: unit in with clause is private child unit
test_data_processing.adb:3:06: error: current unit must also have parent "Data_
↪Processing"
gprbuild: *** compilation phase failed
```

As expected, we get a compilation error because Calculations is only accessible within the Data_Processing, but not in the Test_Data_Processing procedure.

The same restrictions apply to child packages of private packages. For example, if we implement a child package of the Calculations package — let's name it Calculations. Child —, we cannot refer to it in the Test_Data_Processing procedure:

Listing 33: data_processing-calculations-child.ads

```
package Data_Processing.Calculations.Child is

   procedure Process (D : in out Data);

end Data_Processing.Calculations.Child;
```

Listing 34: data_processing-calculations-child.adb

```
package body Data_Processing.Calculations.Child is

   procedure Process (D : in out Data) is
   begin
      Calculate (D);
   end Process;

end Data_Processing.Calculations.Child;
```

Listing 35: test_data_processing.adb

```
with Data_Processing; use Data_Processing;

with Data_Processing.Calculations.Child;
use  Data_Processing.Calculations.Child;

procedure Test_Data_Processing is
   D : Data;
begin
   Calculate (D);
end Test_Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
 ↪Package
MD5: 2eaf23ddbab72578246ac07424008d9d
```

Build output

```
test_data_processing.adb:3:06: error: unit in with clause is private child unit
test_data_processing.adb:3:06: error: current unit must also have parent "Data_
 ↪Processing"
test_data_processing.adb:9:04: error: "Calculate" is not visible
test_data_processing.adb:9:04: error: non-visible declaration at data_processing-
 ↪calculations.ads:3
gprbuild: *** compilation phase failed
```

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 568).

Note that subprograms can also be declared private. We'll see this *in another section* (page 585).

Important

We've discussed package renaming *in a previous section* (page 549). We can rename a package as a private package, too. For example:

Listing 36: driver_m1.ads

```
package Driver_M1 is

end Driver_M1;
```

Listing 37: drivers.ads

```
package Drivers
  with Pure is

end Drivers;
```

Listing 38: drivers-m1.ads

```
with Driver_M1;

private package Drivers.M1 renames Driver_M1;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_Package_Renaming
MD5: c03584dc26abb108c9c04074234b9637

Obviously, Drivers.M1 has the same restrictions as any private package:

Listing 39: test_driver.adb

```
with Driver_M1;
with Drivers.M1;

procedure Test_Driver is
begin
   null;
end Test_Driver;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_Package_Renaming
MD5: 55415978604ccea4eeaeb02df13cd2f4

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

- 10.1.1 Compilation Units - Library Units[248]

[248] http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html

13.3 Private with clauses

13.3.1 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:

Listing 40: p.ads

```
package P is

   type T is null record;

end P;
```

Listing 41: data.ads

```
private with P;

package Data is

   type T2 is private;

private

   --  Information from P is
   --  visible here
   type T2 is new P.T;

end Data;
```

Listing 42: main.adb

```
with Data; use Data;

procedure Main is
   A : T2;
begin
   null;
end Main;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Simple_
 ↪Private_With_Clause
MD5: d0705add0dd7861c83822b0d35dacba4
```

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:

Listing 43: data.ads

```
private with P;

package Data is

   --  ERROR: information from P
   --  isn't visible here

```

(continues on next page)

(continued from previous page)

```
   type T2 is new P.T;

end Data;
```

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:

Listing 44: p.ads

```
package P is

   type T is null record;

   procedure Process (A : T) is null;

end P;
```

Listing 45: data.ads

```
private with P;

package Data is

   type T2 is private;

   procedure Process (A : T2);

private

   --  Information from P is
   --  visible here
   type T2 is new P.T;

end Data;
```

Listing 46: data.adb

```
package body Data is

   procedure Process (A : T2) is
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Simple_
↪Private_With_Clause
MD5: cecc09f95bd43dd7fd34a9e289bd2674
```

In the body of the Data, we can access information from the P package — as we do in the P.Process (P.T (A)) statement of the Process procedure.

13.3.2 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;
```

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;
```

Listing 51: test_parent_child.adb

```
with P.Public_Child; use P.Public_Child;

procedure Test_Parent_Child is
   A : T2;
```

(continues on next page)

(continued from previous page)

```
begin
   null;
end Test_Parent_Child;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Private_
↪With_Clause
MD5: a6028416a957184be55a54f96a319e61
```

In this example, we're referring to the `P.Private_Child` package in the `P.Public_Child` package. As expected, this works fine. However, using a *normal* with clause doesn't work in this case:

Listing 52: p-public_child.ads

```
with P.Private_Child;

package P.Public_Child is

   type T2 is private;

private

   type T2 is new P.Private_Child.T;

end P.Public_Child;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Private_
↪With_Clause
MD5: 2f32f29ecb4ae13bb4487c94d3bf18d9
```

Build output

```
p-public_child.ads:1:06: error: current unit must also be private descendant of "P"
gprbuild: *** compilation phase failed
```

This gives an error because the information from the `P.Private_Child`, being a private child package, cannot be accessed in the public part of another child package. In summary, unless both packages are private packages, it's only possible to access the information from a private package in the private part of a non-private child package.

In the Ada Reference Manual

- 10.1.2 Context Clauses - With Clauses[249]

13.4 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:

[249] http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html

Listing 53: a.ads

```
with B; use B;

package A is

   type T1 is record
      Value : T2;
   end record;

end A;
```

Listing 54: b.ads

```
with A; use A;

package B is

   type T2 is record
      Value : T1;
   end record;

end B;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Circular_
↪Dependency
MD5: ae64f33706f1c58603aff2c33b02c910
```

Build output

```
b.ads:1:06: error: circular unit dependency
b.ads:1:06: error: "B (spec)" depends on "A (spec)"
b.ads:1:06: error: "A (spec)" depends on "B (spec)"
b.ads:1:06: error: "B (spec)" depends on "B (spec)"
gprbuild: *** compilation phase failed
```

Here, we have two *mutually dependent types* (page 181) 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 41). For the specific case of the previous source-code example, this would be the limited visibility to package B from package A's perspective:

```
package B is

   --  Incomplete type
   type T2;

end B;
```

As we've seen previously,

- we cannot declare objects of incomplete types, but we can declare access types and anonymous access objects of incomplete types. Also,

- we can use anonymous access types to declare *mutually dependent types* (page 181).

Keeping this information in mind, we can now correct the previous code by using limited with clauses for package A and declaring the component of the T1 record using an anonymous access type:

Listing 55: a.ads

```
limited with B;

package A is

   type T1 is record
      Ref : access B.T2;
   end record;

end A;
```

Listing 56: b.ads

```
with A; use A;

package B is

   type T2 is record
      Value : T1;
   end record;

end B;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↪Visibility
MD5: 48591850665085a6fbb184f51b658a1b
```

As expected, we can now compile the code without issues.

Note that we can also use limited with clauses for both packages. If we do that, we must declare all components using anonymous access types:

Listing 57: a.ads

```
limited with B;

package A is

   type T1 is record
      Ref : access B.T2;
   end record;

end A;
```

Listing 58: b.ads

```
limited with A;

package B is

   type T2 is record
      Ref : access A.T1;
   end record;
```

(continues on next page)

(continued from previous page)

```

end B;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↪Visibility_2
MD5: 3884086e89400245346acfbbf0691906
```

Now, both packages A and B have limited visibility to each other.

In the Ada Reference Manual

- 10.1.2 Context Clauses - With Clauses[250]

13.4.1 Limited visibility and private with clauses

We can limit the visibility and use *private with clauses* (page 560) at the same time. For a package P, we do this by simply writing **limited private with** P.

Let's reuse the previous source-code example and convert types T1 and T2 to private types:

Listing 59: a.ads

```
limited private with B;

package A is

   type T1 is private;

private

   --  Here, we have limited visibility
   --  of package B

   type T1 is record
      Ref : access B.T2;
   end record;

end A;
```

Listing 60: b.ads

```
private with A;

package B is

   type T2 is private;

private

   use A;

   --  Here, we have full visibility
   --  of package A

   type T2 is record
```

(continues on next page)

[250] http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html

(continued from previous page)

```
      Value : T1;
   end record;

end B;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↪Private_Visibility
MD5: b3ac546e2f55fb91229e834ca7a9783d
```

In this updated version of the source-code example, we have not only limited visibility to package B, but also, each package is just visible in the private part of the other package.

13.4.2 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

```ada
package Info is

   function Zero_Func return Integer is (0);

   Zero_Const : constant := 0;

end Info;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↪Private_Visibility_Other_Elements
MD5: e9b01b4d59db5982532634f9162518ce
```

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

```ada
limited private with Info;

package A is

   type T1 is private;

private

   type T1 is record
      V : Integer := Info.Zero_Const;
      W : Integer := Info.Zero_Func;
   end record;

end A;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↪Private_Visibility_Other_Elements
MD5: 61ecb5dc2617eecac62a05d7d2c6c0df
```

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
```

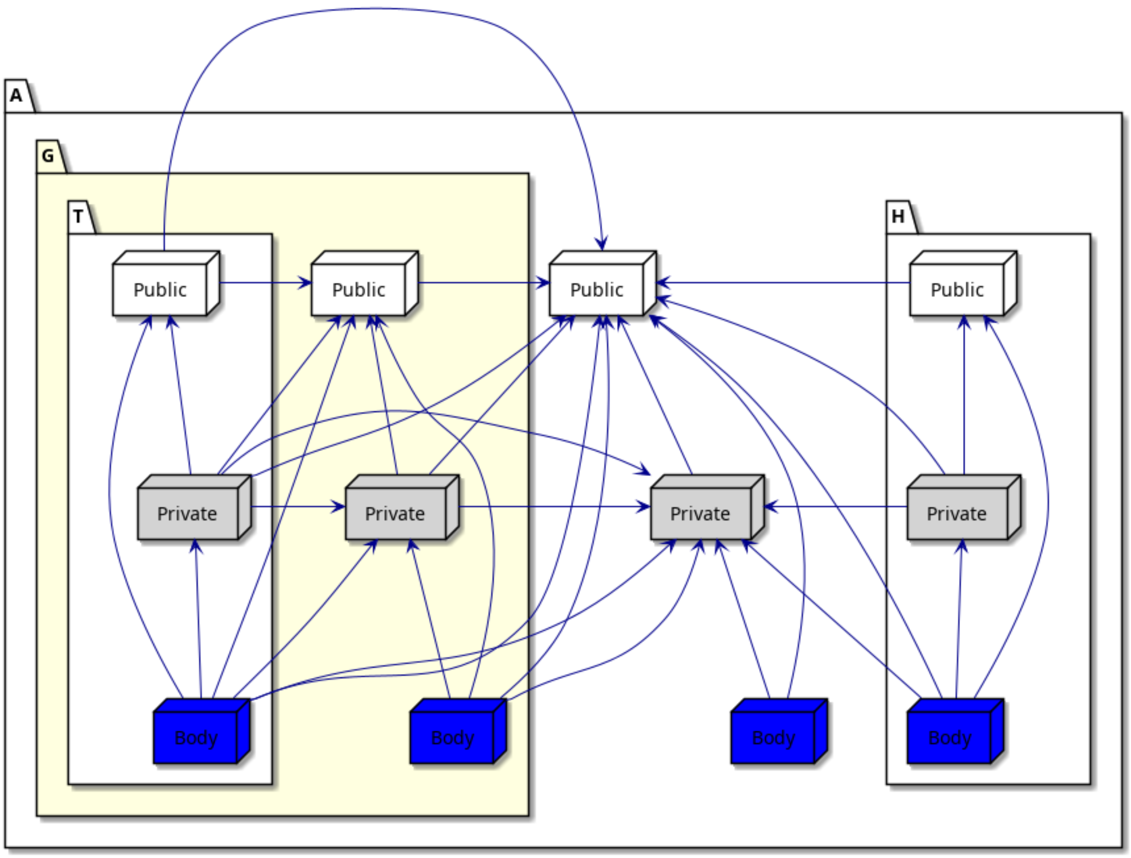
As expected, compilation fails because of the limited visibility — as `Zero_Const` and `Zero_Func` from the `Info` package are not visible in the private part of `A`. (Of course, if we revert to full visibility by simply removing the **limited** keyword from the example, the code compiles just fine.)

13.5 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.

13.5.1 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:

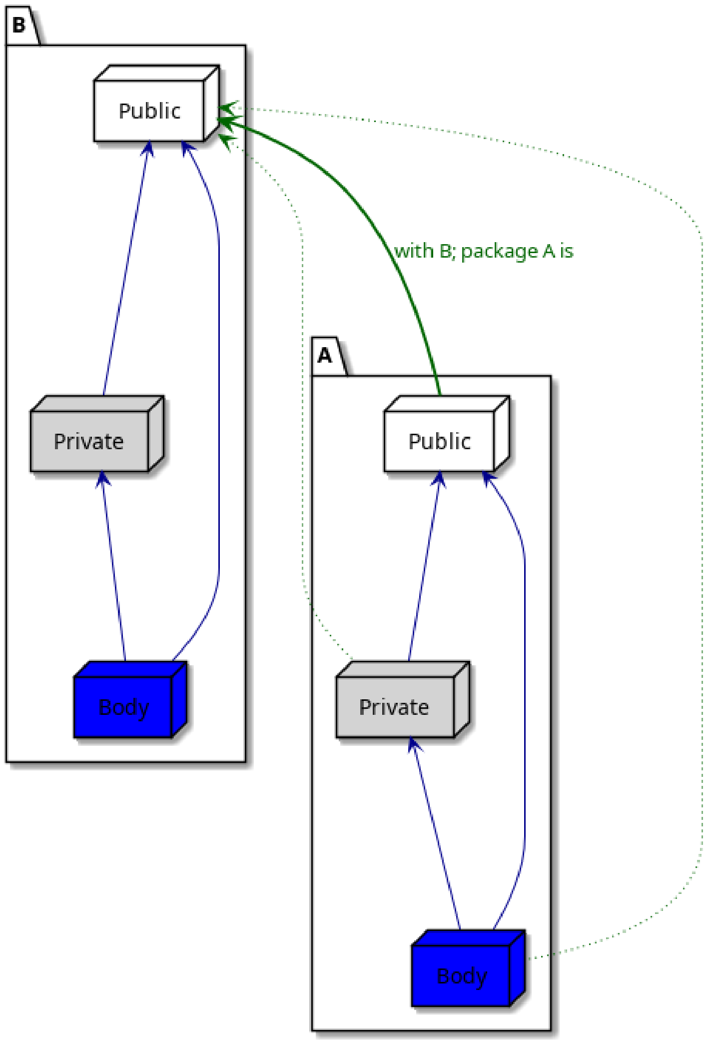


Because of this automatic visibility, many with clauses would be redundant in child packages. For example, we don't have to write `with A; package A.G is`, since the specification of package A is already visible in its child packages.

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.

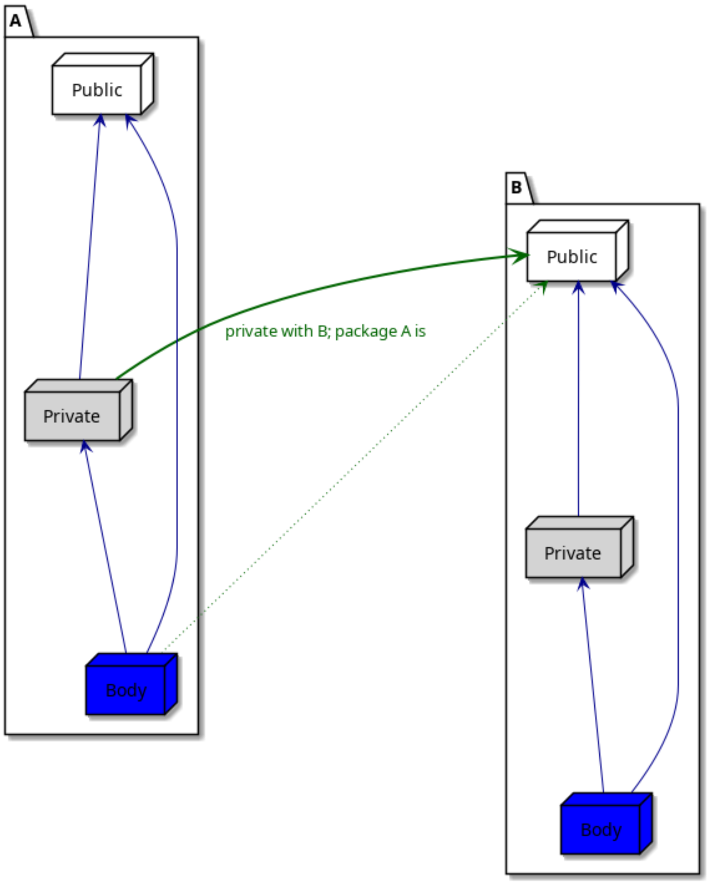
13.5.2 With clauses and visibility

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`):

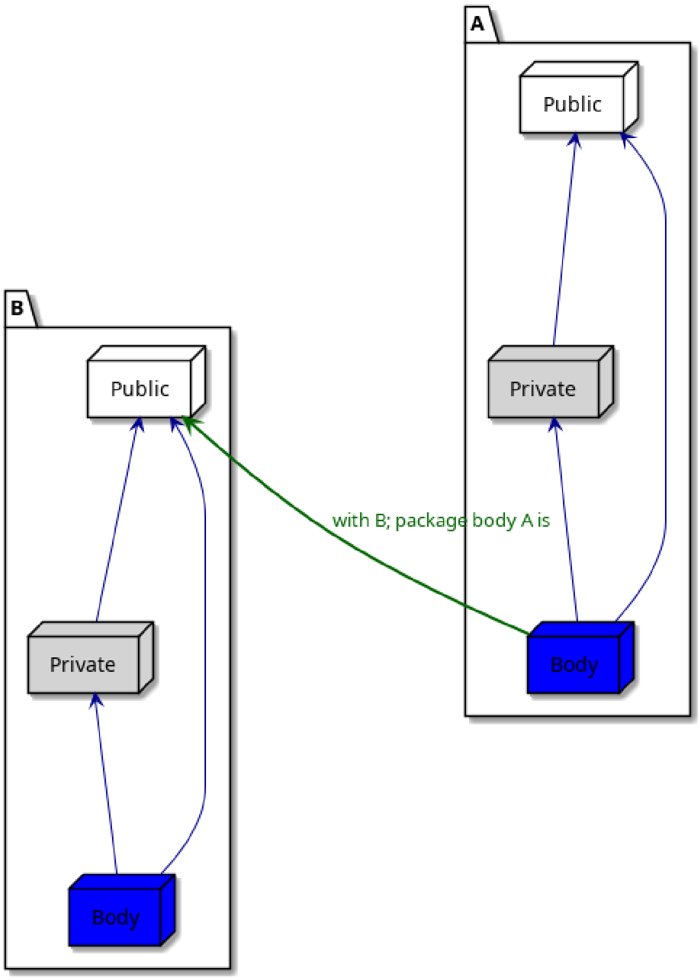


As we already know, the information from the public part of package B is visible in the public part of package A. In addition to that, it's also visible in the private part and in the body of package A. This is indicated by the dotted green arrows in the figure above.

Now, let's see the case where we refer to package B in the private part of package A (using `private with B`):



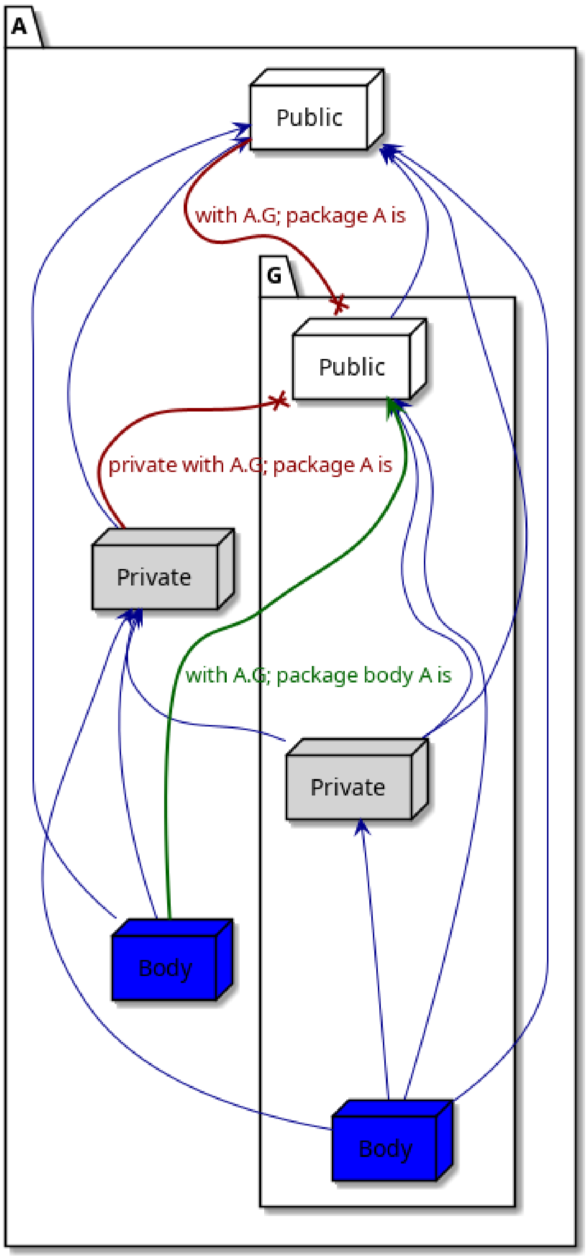
Here, the information is visible in the private part of package A, as well as in its body. Finally, let's see the case where we refer to package B in the body of package A:



Here, the information is only visible in the body of package A.

13.5.3 Circular dependency

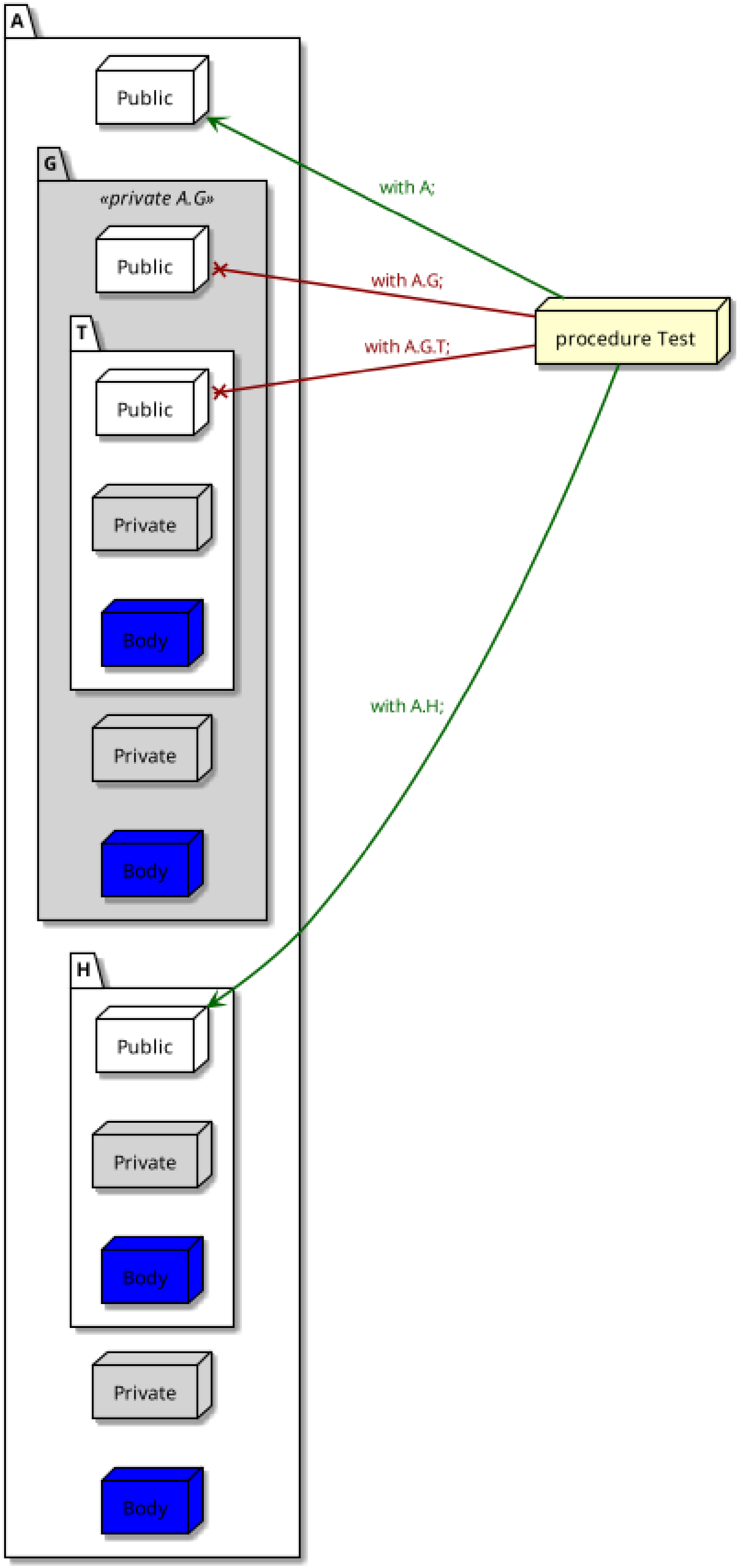
Let's return to package A and its descendants. As we've seen in previous sections, we cannot refer to a child package in the specification of its parent package because that would constitute circular dependency. (For example, we cannot write `with A.G; package A is`.) This situation — which causes a compilation error — is indicated by the red arrows in the figure below:



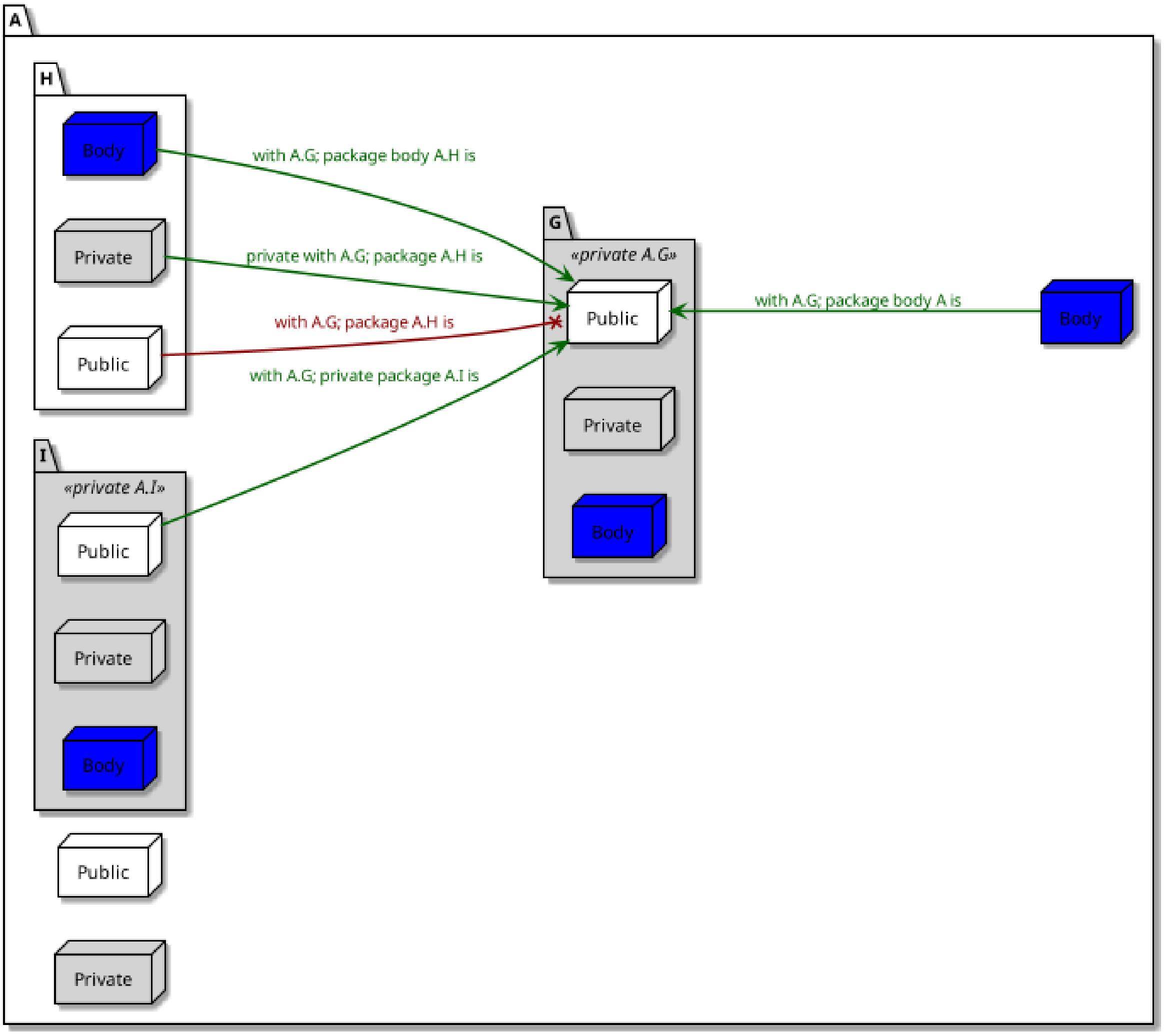
Note that referring to the child package `A.G` in the body of its parent is perfectly fine.

13.5.4 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.

13.6 Use type clause

Back in the Introduction to Ada course[251], we saw that use clauses provide direct visibility — in the scope where they're used — to the content of a package's visible part.

For example, consider this simple procedure:

Listing 63: display_message.adb

```
with Ada.Text_IO;

procedure Display_Message is
begin
   Ada.Text_IO.Put_Line ("Hello World!");
end Display_Message;
```

[251] https://learn.adacore.com/courses/intro-to-ada/chapters/modular_programming.html#intro-ada-use-clause

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.No_Use_Clause
MD5: 4c6ff19809c13ebd2fdfda482914e5f8
```

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:

Listing 64: display_message.adb

```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[252]

13.6.1 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.

Listing 65: points.ads

```ada
package Points is

   type Point is private;

   function Init return Point;

   function "+" (P : Point;
                 I : Integer) return Point;

private

   type Point is record
      X, Y : Integer;
```

(continues on next page)

[252] http://www.ada-auth.org/standards/22rm/html/RM-8-4.html

(continued from previous page)

```
   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: f5d44dd1fee8cf4d1a7e730f9a7c64cc
```

Here, we have a use clause, so we have direct visibility to the content of `Points`'s visible part.

13.6.2 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

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: ca896b456a90c19b29ec4f262144c131
```

Although this code is correct, it might be difficult to read, as we have to specify the package whenever we're referring to a type or a subprogram from that package. Even worse: we now have to write operators in the prefixed form — such as `Points."+" (P, 1)`.

13.6.3 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:

Listing 68: show_point.adb

```
with Points;

procedure Show_Point is
   use type Points.Point;

   P : Points.Point;
begin
   P := Points.Init;
   P := P + 1;
end Show_Point;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: a9527276c27a67be8b5a59efcf6e5cfd
```

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.

13.6.4 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:

Listing 69: show_point.adb

```
with Points;

procedure Show_Point is
   use all type Points.Point;

   P : Points.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: 4a8f6edd4e1811c4e8acb24393690282
```

Now, we've removed the prefix from all operations on the P variable.

13.7 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[253]

13.7.1 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:

Listing 70: show_any_complex.ads

```
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);
```

Listing 71: show_any_complex.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪Use_Type_Clause_Complex_Types
MD5: 2527291906d3a600eecd6d36e4359c1a
```

Then, we implement a test procedure where we declare the Complex_Float_Types package as an instance of the **Generic**_Complex_Types package:

Listing 72: show_use.adb

```
with Ada.Numerics; use Ada.Numerics;

with Ada.Numerics.Generic_Complex_Types;

with Show_Any_Complex;

procedure Show_Use is
```

(continues on next page)

[253] http://www.ada-auth.org/standards/22rm/html/RM-8-4.html

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪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.

13.7.2 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:

Listing 73: 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;

```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪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
gprbuild: *** compilation phase failed
```

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.

13.7.3 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;

```

(continues on next page)

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪Use_Type_Clause_Complex_Types
MD5: 0b3285364ea0188a678db2fc406741b8
```

Runtime output

```
C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)
```

Another possibility is to write a use clause in the form **use** all **type**:

Listing 75: show_use.adb

```
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);

   C, D, X : Complex_Float_Types.Complex;
begin
   C := Compose_From_Polar (3.0, Pi / 2.0);
   D := Compose_From_Polar (5.0, Pi / 2.0);
   X := C + D;

   Show_Complex_Float ("C:", C);
   Show_Complex_Float ("D:", D);
   Show_Complex_Float ("X:", X);
end Show_Use;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪Use_Type_Clause_Complex_Types
MD5: 90333ff41e25afb1399f7f94f7e2b566
```

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);

   C, D, X : Complex_Float_Types.Complex;
   E, F, Y : Complex_Long_Float_Types.Complex;
begin
   C := Compose_From_Polar (3.0, Pi / 2.0);
   D := Compose_From_Polar (5.0, Pi / 2.0);
   X := C + D;

   Show_Complex_Float ("C:", C);
   Show_Complex_Float ("D:", D);
   Show_Complex_Float ("X:", X);

   E := Compose_From_Polar (3.0, Pi / 2.0);
   F := Compose_From_Polar (5.0, Pi / 2.0);
   Y := E + F;

   Show_Complex_Long_Float ("E:", E);
   Show_Complex_Long_Float ("F:", F);
   Show_Complex_Long_Float ("Y:", Y);
end Show_Use;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪Use_Type_Clause_Complex_Types
```

(continues on next page)

(continued from previous page)

```
MD5: 48f31250116f107d3143703debb3107d
```

Runtime output

```
C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)
E: ( 1.83697019872103E-16, 3.00000000000000E+00)
F: ( 3.06161699786838E-16, 5.00000000000000E+00)
Y: ( 4.89858719658941E-16, 8.00000000000000E+00)
```

As expected, the code compiles correctly.

CHAPTER

FOURTEEN

SUBPROGRAMS AND MODULARITY

14.1 Private subprograms

We've seen *previously* (page 552) 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

```
Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↪Subprograms.Private_Test_Procedure
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

```
Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↪Subprograms.Private_Test_Procedure
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[254]
- 10.1.2 Context Clauses - With Clauses[255]

14.1.1 Private subprograms of a package

A more useful example is to declare private subprograms of a package. For example:

Listing 4: data_processing.ads

```
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

```
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

```
private
procedure Data_Processing.Calculate
  (D : in out Data);
```

Listing 7: data_processing-calculate.adb

```
procedure Data_Processing.Calculate
  (D : in out Data)
is
begin
   --  Dummy implementation...
   D.F := 0.0;
end Data_Processing.Calculate;
```

254 http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html
255 http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html

Listing 8: 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

```
Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↪Subprograms.Private_Package_Procedure
MD5: 0f6af1b02f37e011abac5b2a6dfc482d
```

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.

14.1.2 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

```
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

```
package body Private_Data_Processing is

   procedure Process (D : in out Data) is
   begin
      D.F := 0.0;
   end Process;

end Private_Data_Processing;
```

Listing 11: test_private_data_processing.ads

```
private procedure Test_Private_Data_Processing;
```

Listing 12: test_private_data_processing.adb

```
with Private_Data_Processing;
use  Private_Data_Processing;

procedure Test_Private_Data_Processing is
   D : Data;
begin
   Process (D);
end Test_Private_Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↪Subprograms.Private_Subprogram_Private_Package
MD5: 3527e54f99eb2cb52317c987b499caaf
```

In this code example, we have the private Private_Data_Processing package. In order to test it, we implement the private procedure Test_Private_Data_Processing. The fact that this procedure is private allows us to use the Private_Data_Processing package as if it was a non-private package. We then use the private Test_Private_Data_Processing procedure as our main application, so we can run it to test application the private package.

Child subprograms of private packages

We could also implement the Test subprogram that we use to test a private package P as a child subprogram of that package. In other words, we could write a procedure P.Test and use it as our main application. The advantage here is that this allows us to access the private part of the parent package P in the test procedure.

Let's rewrite the Test_Private_Data_Processing procedure from the previous example as the child procedure Private_Data_Processing.Test:

Listing 13: private_data_processing.ads

```
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 14: private_data_processing.adb

```
package body Private_Data_Processing is

   procedure Process (D : in out Data) is
   begin
      null;
```

(continues on next page)

(continued from previous page)

```
   end Process;

end Private_Data_Processing;
```

Listing 15: private_data_processing-test.ads

```
procedure Private_Data_Processing.Test;
```

Listing 16: private_data_processing-test.adb

```
procedure Private_Data_Processing.Test is
   D : Data := (F => 0.0);
begin
   Process (D);
end Private_Data_Processing.Test;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↪Subprograms.Private_Package_Child_Subprogram
MD5: 0726f5890a5b3847244d1ae08989e158
```

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

CHAPTER

FIFTEEN

ACCESS TYPES

We discussed access types back in the Introduction to Ada course[256]. 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.

15.1 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.

15.1.1 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:

Listing 1: show_access_type_declaration.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Terminology.Access_
↪Type_Declaration
MD5: 64e4e0847a73a9ed23e29e09798934de
```

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 677)).

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,

256 https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-types-overview

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 706) later on.

15.1.2 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:

Listing 2: show_simple_allocation.adb

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Terminology.Simple_
↪Allocation
MD5: 32ca8cf523e19b25dabb55da6df1f18d
```

In this example, `I1` is an access object and the object allocated via `new Integer` is its designated object.

15.1.3 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:

Listing 3: show_values.adb

```
procedure Show_Values is

   --  Access-to-object type
   type Integer_Access is access Integer;

   I1, I2, I3 : Integer_Access;

begin
   I1 := new Integer;
   I3 := new Integer;

   --  Copying the access value of I1 to I2
   I2 := I1;

   --  Copying the designated value of I1
   I3.all := I1.all;

end Show_Values;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Terminology.Values
MD5: a152ee813b8ed9fad985cf4e2c25d847
```

In this example, the assignment `I2 := I1` copies the access value of `I1` to `I2`. The assignment `I3.all := I1.all` copies `I1`'s designated value to `I3`'s designated object. (As we already know, `.all` is used to dereference an access object. We discuss this topic again *later in this chapter* (page 623).)

In the Ada Reference Manual

- 3.10 Access Types[257]

15.2 Access types: Allocation

Ada makes the distinction between pool-specific and general access types, as we'll discuss in this section. Before doing so, however, let's talk about memory allocation.

In general terms, memory can be allocated dynamically on the heap or statically on the stack. (Strictly speaking, both are dynamic allocations, in that they occur at run-time with amounts not previously specified.) For example:

Listing 4: show_simple_allocation.adb

```
procedure Show_Simple_Allocation is

   --  Declaring access type:
   type Integer_Access is access Integer;

   --  Declaring access object:
   A1 : Integer_Access;

begin
   --  Allocating an Integer object on the heap
```

(continues on next page)

[257] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

(continued from previous page)

```
   A1 := new Integer;

   declare
      --  Allocating an Integer object on the
      --  stack
      I : Integer;
   begin
      null;
   end;

end Show_Simple_Allocation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↪Allocation.Simple_Allocation
MD5: 4144feb99e6e0b1a0749fce0b20370a1
```

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 597) 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:

Listing 5: show_general_access_type.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↪Allocation.General_Access_Types
MD5: f166291ad1975396131775d0aff6ad9d
```

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 636) later on.)

For further reading...

Note that it is possible to use general access types to allocate objects on the heap:

Listing 6: show_simple_allocation.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↪Allocation.General_Access_Types_Heap
MD5: 3fa5efeac2f66794f066ab29f26bf7ca
```

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[258]

15.2.1 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`.

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

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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↪Allocation.Pool_Specific_Access_Types
MD5: 80d0e9764917fa8352b6616e3a8425de
```

Runtime output

```
A1  : (access 2f38b2a0)
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL'{SYSTEM.STORAGE_POOLS.TROOT_
↪STORAGE_POOLC object}
A2:   (access 2f38b360)
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL'{SYSTEM.STORAGE_POOLS.TROOT_
↪STORAGE_POOLC object}
A2B:  (access 2f38b360)
```

(continues on next page)

(continued from previous page)

```
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL'{SYSTEM.STORAGE_POOLS.TROOT_
↪STORAGE_POOLC object}
```

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[259]:
  - an access type can be unconstrained, but the actual object allocation must be constrained;
  - we can use a *qualified expression* (page 68) 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 89).
- When running out of memory while allocating via **new**, we get a `Storage_Error` exception because of the *storage check* (page 528).

For example:

Listing 8: show_array_allocation.adb

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

procedure Show_Array_Allocation is

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

   --  Access type with unconstrained
   --  designated subtype and limited storage
   --  size.
   type Integer_Array_Access is
     access Integer_Array
       with Storage_Size => 128;

   --  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;
```

15.2.2 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

```
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);

```

(continues on next page)

[259] https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-type-allocation-constraints

(continued from previous page)

```
   Put_Line ("Arr (designated values): ");
   for E of Arr loop
      Put (E.all'Image);
   end loop;

end Show_Access_Array_Allocation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↪Allocation.Integer_Access_Array
MD5: 4afc9358c8aa9426a97ca8932c75d932
```

Runtime output

```
Arr:
[(access 18bc13e0), (access 18bc1400), (access 18bc1420), (access 18bc1440),
 (access 18bc1460), (access 18bc1480), (access 18bc14a0), (access 18bc14c0),
 (access 18bc14e0), (access 18bc1500)]
Arr (designated values):
 0 0 0 0 0 0 0 0 0 0
```

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 266) 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

```ada
package Workers is

   task type Worker is
      entry Start (Id : Positive);
      entry Stop;
   end Worker;

   type Worker_Access is access Worker;

   type Worker_Array is
     array (Positive range <>) of Worker_Access;

end Workers;
```

Listing 11: workers.adb

```ada
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;
```

(continues on next page)

(continued from previous page)

```
      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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↪Allocation.Workers
MD5: d29e3d56585f8d9a63b805c680e5dc54
```

Runtime output

```
Started Worker # 1
Started Worker # 2
Started Worker # 3
Started Worker # 4
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
```

(continues on next page)

(continued from previous page)

```
Started Worker # 16
Started Worker # 17
Started Worker # 18
Started Worker # 19
Started Worker # 20
Some processing...
Stopped Worker # 1
Stopped Worker # 2
Stopped Worker # 3
Stopped Worker # 4
Stopped Worker # 18
Stopped Worker # 6
Stopped Worker # 7
Stopped Worker # 9
Stopped Worker # 10
Stopped Worker # 12
Stopped Worker # 5
Stopped Worker # 13
Stopped Worker # 19
Stopped Worker # 15
Stopped Worker # 16
Stopped Worker # 17
Stopped Worker # 20
Stopped Worker # 14
Stopped Worker # 8
Stopped Worker # 11
```

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)`.

15.3 Discriminants as Access Values

We can use access types when declaring discriminants. Let's see an example:

Listing 13: custom_recs.ads

```
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;
      --             ^^^^^^^^^
      --  Setting I's default to use the
      --  designated value of IA:
   end record;

   procedure Show (R : Rec);

end Custom_Recs;
```

Listing 14: 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 15: show_discriminants_as_access_values.adb

```
with Custom_Recs; use Custom_Recs;

procedure Show_Discriminants_As_Access_Values is

   IA : constant Integer_Access :=
          new Integer'(10);
   R  : Rec (IA);

begin
   Show (R);

   IA.all := 20;
   R.I    := 30;
   Show (R);

   --  As expected, we cannot change the
   --  discriminant. The following line is
   --  triggers a compilation error:
   --
   --  R.IA := new Integer;

end Show_Discriminants_As_Access_Values;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
↪Access_Values.Discriminant_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:

Listing 16: custom_recs.ads

```
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;
```

Listing 17: show_discriminants_as_access_values.adb

```
with Custom_Recs; use Custom_Recs;

procedure Show_Discriminants_As_Access_Values is

   R1 : Rec;
   --   ^^^
   --   no discriminant: use default

   R2 : Rec (new Integer'(20));
   --        ^^^^^^^^^^^^^^^^
   --        allocating an unnamed integer object

begin
   Show (R1);
   Show (R2);
end Show_Discriminants_As_Access_Values;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
↪Access_Values.Discriminant_Access_Values
MD5: 968cb88ed7e9e6958ab66fb6f5a7ce2d
```

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[260]
- 3.7.1 Discriminant Constraints[261]

15.3.1 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;
```

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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
↪Access_Values.Persons
MD5: 9b1109d076b6f06632c8685a41616210
```

Runtime output

[260] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html
[261] http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html

```
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 200), we can only use discrete types (or access types) as discriminants. Therefore, you wouldn't be able to use a string, for example, directly as a discriminant without using access types:

Listing 21: persons.ads

```ada
package Persons is

   --  ERROR: Declaring a discriminant with an
   --         unconstrained type:
   type Person (Name : String) is record
      Age : Integer;
   end record;

end Persons;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
↪Access_Values.Persons_Error
MD5: 4144852aaf95da62bc4781b1e8dc2717
```

Build output

```
persons.ads:5:24: error: discriminants must have a discrete or access type
gprbuild: *** compilation phase failed
```

As expected, compilation fails for this code because the discriminant of the Person type is indefinite.

However, the advantage of discriminants as access values isn't restricted to being able to 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

```
with Gen_Custom_Recs;

package Custom_Recs is

    type Incomp;
    --  Incomplete type declaration!

   type Incomp_Access is access Incomp;

   --  Instantiating package using
   --  incomplete type Incomp:
   package Inst is new
     Gen_Custom_Recs
       (T        => Incomp,
        T_Access => Incomp_Access);
   subtype Rec is Inst.T_Rec;

   --  At this point, Rec (Inst.T_Rec) uses
   --  an incomplete type as the designated
   --  subtype of its discriminant type

   procedure Show (R : Rec) is null;

   --  Now, we complete the Incomp type:
   type Incomp (B : Boolean := True) is private;

private
   --  Finally, we have the full view of the
   --  Incomp type:
   type Incomp (B : Boolean := True) is
     null record;

end Custom_Recs;
```

Listing 24: show_rec.adb

```
with Custom_Recs; use Custom_Recs;

procedure Show_Rec is
   R : Rec (new Incomp);
begin
   Show (R);
end Show_Rec;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
↪Access_Values.Generic_Access
MD5: c65510e8c6a7625cbd08aa9e68f05115
```

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 725), which provide some advantages in terms of *accessibility rules* (page 645).

15.3.2 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:

Listing 25: show_person.adb

```ada
with Persons; use Persons;

procedure Show_Person is
   S1 : String_Access := new String'("John");
   S2 : String_Access := new String'("Mark");
   P : Person := (Name => S1,
                  Age  => 30);
begin
   P := (Name => S1, Age => 31);
   --            ^^ OK: we didn't change the
   --                   discriminant.
   Show (P);

   --  We can just repeat the discriminant:
   P := (Name => P.Name, Age => 32);
   --            ^^^^^^ OK: we didn't change the
   --                       discriminant.
   Show (P);

   --  Of course, we can change the string itself:
   S1.all := "Mark";
   Show (P);

   P := (Name => S2, Age => 40);
   --            ^^ ERROR: we changed the
   --                      discriminant!
   Show (P);
end Show_Person;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
↪Access_Values.Persons
MD5: 96f4742365eb6a07c377a5dec28b5767
```

Runtime output

```
Name = John
Age  =  31
Name = John
Age  =  32
Name = Mark
Age  =  32

raised CONSTRAINT_ERROR : show_person.adb:24 discriminant check failed
```

The first and the second assignments to P are OK because we didn't change the discriminant. However, the last assignment raises the `Constraint_Error` exception at runtime because we're changing the discriminant.

15.4 Parameters as Access Values

In addition to *using discriminants as access values* (page 603), 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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Names
MD5: 82ce94987dce9026aed54a0deb3cc548
```

This is the complete code example:

Listing 27: names.ads

```
package Names is

   type Name is access String;

   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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Names
```

(continues on next page)

(continued from previous page)

```
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

```ada
package Names is

   type Name is access String;

   procedure Show (N : String);

end Names;
```

Listing 31: 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 32: show_names.adb

```ada
with Names; use Names;

procedure Show_Names is
   N : Name := new String'("John");
begin
   Show (N.all);
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Names_String
MD5: 097ec1ff781fda9deed1de23cae39ae5
```

Runtime output

```
Name: John
```

It's important to highlight the difference between passing an access value to a subprogram and passing an object by reference. In both versions of this code example, the compiler will make use of a reference for the actual parameter of the N parameter of the Show procedure. However, the difference between these two cases is that:

- N : Name is a reference to an object (because it's an access value) that is passed by value, and

- `N : String` is an object passed by reference.

15.4.1 Changing the referenced object

Since the Name type gives us access to an object in the Show procedure, we could actually change this object inside the procedure. To illustrate this, let's change the Show procedure to lower each character of the string before displaying it (and rename the procedure to Lower_And_Show):

Listing 33: names.ads

```ada
package Names is

   type Name is access String;

   procedure Lower_And_Show (N : Name);

end Names;
```

Listing 34: names.adb

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

with Ada.Characters.Handling;
use  Ada.Characters.Handling;

package body Names is

   procedure Lower_And_Show (N : Name) is
   begin
      for I in N'Range loop
         N (I) := To_Lower (N (I));
      end loop;
      Put_Line ("Name: " & N.all);
   end Lower_And_Show;

end Names;
```

Listing 35: show_changed_names.adb

```ada
with Names; use Names;

procedure Show_Changed_Names is
   N : Name := new String'("John");
begin
   Lower_And_Show (N);
end Show_Changed_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Changed_Names
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;

end Names;
```

Listing 38: show_changed_names.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Changed_Names_String
MD5: 783ea8c45ed8ad3e0007524c11b6bcc4
```

Runtime output

```
Name: john
```

15.4.2 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

```
package Names is

```

(continues on next page)

(continued from previous page)

```
   type Name is access String;

   procedure Lower_And_Show (N : in out Name);

end Names;
```

Listing 40: 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 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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Replaced_Names
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 656), as explained later on.

15.4.3 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:

Subprogram	Parameter type	Parameter mode
Show	Name	**in**
Show	**String**	**in**
Lower_And_Show	Name	**in**
Lower_And_Show	**String**	**in out**

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 637), 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:

Listing 42: names.ads

```
package Names is

   type Name is access String;

   type Constant_Name is access constant String;

   procedure Show (N : Constant_Name);

end Names;
```

Listing 43: 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 44: show_names.adb

```
with Names; use Names;

procedure Show_Names is
   N : Name := new String'("John");
```

(continues on next page)

(continued from previous page)

```
begin
   Show (Constant_Name (N));
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Names_Constant
MD5: 77526e0a159bf1bcbef08a21be250f3c
```

Runtime output

```
Name: John
```

In this case, the `Constant_Name` type ensures that the `N` parameter won't be changed in the Show procedure. Note that we need to convert from Name to `Constant_Name` to be able to call the Show procedure (in the Show_Names procedure). Although using **in String** is still a simpler solution, this approach works fine.

(Feel free to uncomment the call to `To_Lower` in the Show procedure and the corresponding with- and use-clauses to see that the compilation fails when trying to change the constant object.)

We could also mitigate the problem by using contracts. For example:

Listing 45: names.ads

```
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

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

--  with Ada.Characters.Handling;
--  use  Ada.Characters.Handling;

package body Names is

   procedure Show (N : Name) is
   begin
      --  for I in N'Range loop
      --     N (I) := To_Lower (N (I));
      --  end loop;
      Put_Line ("Name: " & N.all);
   end Show;

end Names;
```

Listing 47: show_names.adb

```
with Names; use Names;

procedure Show_Names is
   N : Name := new String'("John");
begin
   Show (N);
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Names_Postcondition
MD5: 2a70993232baca9d58d36e537a6fd32b
```

Runtime output

```
Name: John
```

Although a bit more verbose than a simple `in String`, the information in the specification of Show at least gives us an indication that the object won't be affected by the call to this subprogram. Note that this code actually compiles if we try to modify `N.all` in the Show procedure, but the post-condition fails at runtime when we do that.

(By uncommentating and building the code again, you'll see an exception being raised at runtime when trying to change the object.)

In the postcondition above, we're using `'Old` to refer to the original object before the subprogram call. Unfortunately, we cannot use this attribute when dealing with *limited private types* (page 787) — 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

```
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

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

--  with Ada.Characters.Handling;
--  use  Ada.Characters.Handling;

package body Names is

   procedure Show (N : Name) is
   begin
      --  for I in N'Range loop
      --     N (I) := To_Lower (N (I));
      --  end loop;
      Put_Line ("Name: " & N.all);
   end Show;

end Names;
```

Listing 50: show_names.adb

```
with Names; use Names;

procedure Show_Names is
   N : Name := Init ("John");
begin
   Show (N);
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Names_Limited_Private
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

```
package Names is

   type Name is limited private;

   type Constant_Name is limited private;
```

```

   function Init (S : String) return Name;

   function To_Constant_Name
     (N : Name)
      return Constant_Name;

   procedure Show (N : Constant_Name);

private

   type Name is
     access String;

   type Constant_Name is
     access constant String;

   function Init (S : String) return Name is
     (new String'(S));

   function To_Constant_Name
     (N : Name)
      return Constant_Name is
        (Constant_Name (N));

end Names;
```

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↪Access_Values.Names_Constant_Limited_Private
MD5: 30da588b57e6b4dfbf9934f77d348473
```

Runtime output

```
Name: John
```

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 is of course justifiable. We'll see such a case in the *next section* (page 620).

15.5 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

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Self_Reference.
↪Linked_List_Example
MD5: b2d3a048473d498bbe691bc6e38ca1e9
```

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

```
generic
   type T is private;
package Linked_Lists is

   type List is limited private;

   procedure Append_Front
      (L : in out List;
```

(continues on next page)

(continued from previous page)

```
       E :        T);

   procedure Append_Rear
      (L : in out List;
       E :        T);

   procedure Show (L : List);

private

   --  Incomplete type declaration:
   type Component;

   --  Using incomplete type:
   type List is access Component;

   type Component is record
      Value : T;
      Next  : List;
      --      ^^^^
      --   Self-reference via access type
   end record;

end Linked_Lists;
```

Listing 56: linked_lists.adb

```
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;
```

(continues on next page)

(continued from previous page)

```
      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;

end Linked_Lists;
```

Listing 57: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Self_Reference.
↪Linked_List_Example
MD5: 6ab1f79c8c3e641eba8057874efc48d7
```

Runtime output

```
[ 1  2  3  4  5  6  7 ]
```

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 610) for the `Append_Front`, `Append_Rear` and `Show` procedures.

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations[262]

15.6 Mutually dependent types using access types

In the section on *mutually dependent types* (page 181), 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

```
package Mutually_Dependent is

   type T2;
   type T2_Access is access T2;

   type T1 is record
      B : T2_Access;
   end record;

   type T1_Access is access T1;

   type T2 is record
      A : T1_Access;
   end record;

end Mutually_Dependent;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Mutually_Dependent_
↪Access_Types.Example
MD5: b21ffc4cdfe3db939dfc841cf8434344
```

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.

15.7 Dereferencing

In the Introduction to Ada course[263], we discussed the **.all** syntax to dereference access values:

Listing 59: show_dereferencing.adb

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

procedure Show_Dereferencing is

   --  Declaring access type:
   type Integer_Access is access Integer;

   --  Declaring access object:
```

(continues on next page)

262 http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html
263 https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-dereferencing

(continued from previous page)

```
   A1 : Integer_Access;

begin
   A1 := new Integer;

   --  Dereferencing access value:
   A1.all := 22;

   Put_Line ("A1: " & Integer'Image (A1.all));
end Show_Dereferencing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.
↪Simple_Dereferencing
MD5: 65655768c17a02991ffeda9a853b6ffb
```

Runtime output

```
A1:  22
```

In this example, we declare A1 as an access object, which allows us to access objects of **Integer** type. We dereference A1 by writing A1.**all**.

Here's another example, this time with an array:

Listing 60: show_dereferencing.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_
↪Dereferencing
MD5: 0e533dfd8ec1a74af17c99633c292e95
```

Runtime output

```
Arr (:  1):  1
Arr (:  2):  2
Arr (:  3):  3
```

(continues on next page)

(continued from previous page)

```
Arr (:  4):  5
Arr (:  5):  8
Arr (:  6):  13
```

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.

In the Ada Reference Manual

- 4.1 Names[264]

15.7.1 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

```
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.
```

(continues on next page)

264 http://www.ada-auth.org/standards/22rm/html/RM-4-1.html

(continued from previous page)

```
   end loop;
end Show_Dereferencing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_
↪Implicit_Dereferencing
MD5: ade602a9e6976018e0c00f930a2399f1
```

Runtime output

```
Arr (:  1):  1
Arr (:  2):  2
Arr (:  3):  3
Arr (:  4):  5
Arr (:  5):  8
Arr (:  6):  13
```

Both forms — `Arr.all (I)` and `Arr (I)` — are equivalent. Note, however, that there's no implicit dereferencing when we want to access the whole array. (Therefore, we cannot write `Arr := (1, 2, 3, 5, 8, 13);`.) However, as slices are implicitly dereferenced, we can write `Arr (1 .. 6) := (1, 2, 3, 5, 8, 13);` instead of `Arr.all (1 .. 6) := (1, 2, 3, 5, 8, 13);`. Alternatively, we can assign to the array components individually and use implicit dereferencing for each component:

```
Arr (1) := 1;
Arr (2) := 2;
Arr (3) := 3;
Arr (4) := 5;
Arr (5) := 8;
Arr (6) := 13;
```

Implicit dereferencing isn't available for the whole array because we have to distinguish between assigning to access objects and assigning to actual arrays. For example:

Listing 62: show_array_assignments.adb

```
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);
```

(continues on next page)

(continued from previous page)

```
   Arr_2 :          Integer_Array_Access :=
                      new Integer_Array (1 .. 6);
begin
   Arr_1.all := (1,   2,  3,  5,   8,  13);
   Arr_2.all := (21, 34, 55, 89, 144, 233);

   --  Array assignment
   Arr_2.all := Arr_1.all;

   Show_Array ("Arr_2", Arr_2);

   --  Access value assignment
   Arr_2 := Arr_1;

   Arr_1.all := (377, 610, 987, 1597, 2584, 4181);

   Show_Array ("Arr_2", Arr_2);
end Show_Array_Assignments;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_
↪Assignments
MD5: 9b1f99af081000c28a6bf9b033127ea3
```

Runtime output

```
Arr_2 1 2 3 5 8 13
Arr_2 377 610 987 1597 2584 4181
```

Here, `Arr_2.all := Arr_1.all` is an array assignment, while `Arr_2 := Arr_1` is an access value assignment. By forcing the usage of the `.all` suffix, the distinction is clear. Implicit dereferencing, however, could be confusing here. (For example, the `.all` suffix in `Arr_2 := Arr_1.all` is an oversight by the programmer when the intention actually was to use access values on both sides.) Therefore, implicit dereferencing is only supported in those 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));
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("R.F: "
             & Float'Image (R.F));
end Show_Dereferencing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.
↪Record_Implicit_Dereferencing
MD5: 9af72502d04f128785f77dcc829d5d48
```

Runtime output

```
R.I:  1
R.F:  5.00000E+00
```

Again, we can replace R.all.I by R.I, as record components are implicitly dereferenced. Also, we could use implicit dereference when assigning to record components individually:

```
R.I := 1;
R.F := 5.0;
```

However, we have to write R.all when assigning to the whole record R.

Attributes

Finally, let's see an example of implicit dereference when using attributes:

Listing 64: show_dereferencing.adb

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

procedure Show_Dereferencing is

   type Integer_Array is
     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
```

(continues on next page)

(continued from previous page)

```
      ("Arr.all'Size: "
       & Integer'Image (Arr.all'Size));
end Show_Dereferencing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_
↪Implicit_Dereferencing
MD5: 5730e18c8d2ed5e26a4d7d325a46a7e9
```

Runtime output

```
Arr'First:  1
Arr'Last:  6
Arr'Component_Size:  32
Arr.all'Component_Size:  32
Arr'Size:  128
Arr.all'Size:  192
```

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:

Entities	Standard Usage	Implicit Dereference
Array components	Arr.all (I)	Arr (I)
Array slices	Arr.all (F .. L)	Arr (F .. L)
Record components	Rec.all.C	Rec.C
Array attributes	Arr.all’First	Arr’First
	Arr.all’First (N)	Arr’First (N)
	Arr.all’Last	Arr’Last
	Arr.all’Last (N)	Arr’Last (N)
	Arr.all’Range	Arr’Range
	Arr.all’Range (N)	Arr’Range (N)
	Arr.all’Length	Arr’Length
	Arr.all’Length (N)	Arr’Length (N)
	Arr.all’Component_Size	Arr’Component_Size
Task attributes	T.all'Identity	T'Identity
	T.all'Storage_Size	T'Storage_Size
	T.all'Terminated	T'Terminated
	T.all'Callable	T'Callable
Tagged type attributes	X.all’Tag	X’Tag
Other attributes	X.all'Valid	X'Valid
	X.all'Old	X'Old
	A.all’Constrained	A’Constrained

In the Ada Reference Manual

- 4.1 Names[265]
- 4.1.1 Indexed Components[266]
- 4.1.2 Slices[267]
- 4.1.3 Selected Components[268]
- 4.1.4 Attributes[269]

15.8 Ragged arrays

Ragged arrays — also known as jagged arrays — are non-uniform, multidimensional arrays. They can be useful to implement tables with varying number of coefficients, as we discuss as an example in this section.

15.8.1 Uniform multidimensional arrays

Consider an algorithm that processes data based on coefficients that depends on a selected quality level:

Quality level	Number of coefficients	#1	#2	#3	#4	#5
Simplified	1	0.15				
Better	3	0.02	0.16	0.27		
Best	5	0.01	0.08	0.12	0.20	0.34

(Note that this is just a bogus table with no real purpose, as we're not trying to implement any actual algorithm.)

We can implement this table as a two-dimensional array (`Calc_Table`), where each quality level has an associated array:

Listing 65: data_processing.ads

```
package Data_Processing is

   type Quality_Level is
     (Simplified, Better, Best);

private

   Calc_Table : constant array
     (Quality_Level, 1 .. 5) of Float :=
       (Simplified =>
            (0.15, 0.00, 0.00, 0.00, 0.00),
        Better     =>
            (0.02, 0.16, 0.27, 0.00, 0.00),
        Best       =>
            (0.01, 0.08, 0.12, 0.20, 0.34));

```

(continues on next page)

265 http://www.ada-auth.org/standards/22rm/html/RM-4-1.html
266 http://www.ada-auth.org/standards/22rm/html/RM-4-1-1.html
267 http://www.ada-auth.org/standards/22rm/html/RM-4-1-2.html
268 http://www.ada-auth.org/standards/22rm/html/RM-4-1-3.html
269 http://www.ada-auth.org/standards/22rm/html/RM-4-1-4.html

(continued from previous page)

```
   Last : constant array
     (Quality_Level) of Positive :=
       (Simplified => 1,
        Better     => 3,
        Best       => 5);

end Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
↪Uniform_Table
MD5: befa8d2b684ee20495f2dd6907dc44d4
```

Note that, in this implementation, we have a separate table `Last` that indicates the actual number of coefficients of each quality level.

Alternatively, we could use a record (`Table_Coefficient`) that stores the number of coefficients and the actual coefficients:

Listing 66: data_processing.ads

```
package Data_Processing is

   type Quality_Level is
     (Simplified, Better, Best);

   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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
↪Uniform_Table
MD5: 4c8602f6ecede0ac1231838c0a0a54b7
```

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

```
package Data_Processing.Operations is

  procedure Process (D : in out Data;
                     Q :        Quality_Level);

end Data_Processing.Operations;
```

Listing 68: data_processing-operations.adb

```
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 1 .. Calc_Table (Q).Last loop
           --  ... * Calc_Table (Q).Coef (J)
           null;
         end loop;
         --  D (I) := ...
         null;
      end loop;
   end Process;

end Data_Processing.Operations;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
↪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 coefficients (or 40 percent of the table) aren't being used. Naturally, this is wasted memory space. We can improve this by using ragged arrays.

15.8.2 Non-uniform multidimensional array

Ragged arrays are declared by using an access type to an array. By doing that, each array can be declared with a different size, thereby creating a non-uniform multidimensional array.

For example, we can declare a constant array `Table` as a ragged array:

Listing 69: data_processing.ads

```
package Data_Processing is

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

private

```

(continues on next page)

(continued from previous page)

```
   type Integer_Array_Access is
     access constant Integer_Array;

   Table : constant array (1 .. 3) of
             Integer_Array_Access :=
     (1 => new Integer_Array'(1 => 15),
      2 => new Integer_Array'(1 => 12,
                              2 => 15,
                              3 => 20),
      3 => new Integer_Array'(1 => 12,
                              2 => 15,
                              3 => 20,
                              4 => 20,
                              5 => 25,
                              6 => 30));

end Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
↪Simple_Ragged_Array
MD5: 28e044a43bf45585a0268c60d63c629e
```

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:

Listing 70: data_processing.ads

```
package Data_Processing is

   type Quality_Level is
     (Simplified, Better, Best);

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

private

   type Coefficients is access constant Data;

   Calc_Table : constant array (Quality_Level) of
                  Coefficients :=
     (Simplified =>
          new Data'(1 => 0.15),
      Better     =>
          new Data'(0.02, 0.16, 0.27),
      Best       =>
          new Data'(0.01, 0.08, 0.12,
                    0.20, 0.34));

end Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
↪Ragged_Table
MD5: 0781b27cba27dbd1e74da54e425a1f4b
```

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 637) in more details later on.

This is the adapted Process procedure:

Listing 71: data_processing-operations.ads

```
package Data_Processing.Operations is

  procedure Process (D : in out Data;
                     Q :        Quality_Level);

end Data_Processing.Operations;
```

Listing 72: data_processing-operations.adb

```
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
           --  ... * Calc_Table (Q).Coef (J)
           null;
         end loop;
         --  D (I) := ...
         null;
      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.

15.9 Aliasing

The term aliasing[270] 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:

Listing 73: show_aliasing.adb

```
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;

```

(continues on next page)

[270] https://en.wikipedia.org/wiki/Aliasing_(computing)

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliasing_
↪Via_Access
MD5: 2fde6073cec9823a1a9d93aec82384e1
```

Runtime output

```
A1:  22
A2:  22
A1:  24
A2:  24
```

In this example, we access the object allocated via **new** by using either A1 or A2, as both refer to the same *aliased* object. In other words, A1 or A2 allow us to access the same object in memory.

> **Important**
>
> 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
>
> ```
> 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;
> ```
>
> **Code block metadata**
>
> ```
> Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Renaming
> MD5: 99a47d02000b91f7464dffe994fd8ee6
> ```
>
> **Runtime output**
>
> ```
> A1:  22
> A2:  22
> A1:  24
> A2:  24
> ```
>
> 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[271]

15.9.1 Aliased objects

As we discussed *previously* (page 595), 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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Access_
↪Aliased_Obj
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));
```

(continues on next page)

[271] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Access_
↪Aliased_Obj
MD5: ac331285456462f05abe7e1fd5e3ca2b
```

Runtime output

```
A1:  22
A2:  22
A1:  24
A2:  24
```

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[272]
- 3.10 Access Types[273]

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:

Type	Declaration
Access-to-variable	**type T_Acc is access all** T
Access-to-constant	**type T_Acc is access constant** T

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
```

(continues on next page)

[272] http://www.ada-auth.org/standards/22rm/html/RM-3-3-1.html
[273] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

(continued from previous page)

```
     access all Integer;

   type Integer_Access_Const is
     access constant Integer;

end Integer_Access_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Show_
↪Access_Modifiers
MD5: 98ccaa703194ae88222ccc5a4400e967
```

As we've seen previously, we can use a type such as `Integer_Access` to allocate objects dynamically. However, we cannot use this type to refer to declared objects, for example. In this case, we have to use an access-to-variable type such as `Integer_Access_All`. Also, if we want to access constants — or access objects that we want to treat as constants —, we use a type such as `Integer_Access_Const`.

Access attribute

To get access to a variable or a constant, we make use of the **Access** attribute. For example, `I_Var'Access` gives us access to the `I_Var` object.

Let's look at an example of how to use the integer access types from the previous code snippet:

Listing 78: integer_access_types.ads

```
package Integer_Access_Types is

   type Integer_Access is
     access Integer;

   type Integer_Access_All is
     access all Integer;

   type Integer_Access_Const is
     access constant Integer;

   procedure Show;

end Integer_Access_Types;
```

Listing 79: integer_access_types.adb

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

package body Integer_Access_Types is

   I_Var : aliased          Integer :=  0;
   Fact  : aliased constant Integer := 42;

   Dyn_Ptr     : constant Integer_Access
                   := new Integer'(30);
   I_Var_Ptr   : constant Integer_Access_All
                   := I_Var'Access;
   I_Var_C_Ptr : constant Integer_Access_Const
                   := I_Var'Access;
   Fact_Ptr    : constant Integer_Access_Const
```

(continues on next page)

(continued from previous page)

```
                    := Fact'Access;

   procedure Show is
   begin
      Put_Line ("Dyn_Ptr:      "
                & Integer'Image (Dyn_Ptr.all));
      Put_Line ("I_Var_Ptr:    "
                & Integer'Image (I_Var_Ptr.all));
      Put_Line ("I_Var_C_Ptr: "
                & Integer'Image
                    (I_Var_C_Ptr.all));
      Put_Line ("Fact_Ptr:     "
                & Integer'Image (Fact_Ptr.all));
   end Show;

end Integer_Access_Types;
```

Listing 80: show_access_modifiers.adb

```
with Integer_Access_Types;

procedure Show_Access_Modifiers is
begin
   Integer_Access_Types.Show;
end Show_Access_Modifiers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Show_
↪Access_Modifiers
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[274]

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:

[274] http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html

Listing 81: show_access_error.adb

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

procedure Show_Access_Error is
   type Integer_Access is access all Integer;
   I_Var : Integer;
   A1    : Integer_Access;
begin
   A1 := I_Var'Access;

   A1.all := 22;
   Put_Line ("A1: " & Integer'Image (A1.all));
end Show_Access_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Access_Non_
↪Aliased_Obj
MD5: 2a9904062eea96ae6dc209493d6f20d4
```

Build output

```
show_access_error.adb:8:10: error: prefix of "Access" attribute must be aliased
gprbuild: *** compilation phase failed
```

In this example, the compiler complains that we cannot get a reference to `I_Var` because `I_Var` is not aliased.

Ragged arrays using aliased objects

We can use aliased objects to declare *ragged arrays* (page 630). For example, we can rewrite a previous program using aliased constant objects:

Listing 82: data_processing.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Ragged_
↪Array_Aliased_Objs
MD5: 7e284560c447c02628e34bac982d4ad5
```

Here, instead of allocating the constant arrays dynamically via **new**, we declare three aliased arrays (Tab_1, Tab_2 and Tab_3) and get a reference to them in the declaration of Table.

Aliased access objects

It's interesting to mention that access objects can be aliased themselves. Consider this example where we declare the Integer_Access_Access type to refer to an access object:

Listing 83: show_aliased_access_obj.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliased_
↪Access
MD5: 77e9be5e29cfb99aef9409728202ba9d
```

Runtime output

```
A:  22
B:  22
```

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.

15.9.2 Aliased components

Components of an array or a record can be aliased. This allows us to get access to those components:

Listing 84: show_aliased_components.adb

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

procedure Show_Aliased_Components is
```

(continues on next page)

(continued from previous page)

```ada

   type Integer_Access is access all Integer;

   type Rec is record
      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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliased_
Components
MD5: 5dfaa248caf8e37a4a3a1e1a24973777
```

Runtime output

```
A:  24
A:  42
```

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 gives 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[275]

[275] http://www.ada-auth.org/standards/22rm/html/RM-3-6.html

15.9.3 Aliased parameters

In addition to aliased objects and components, we can declare *aliased parameters* (page 472), 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:

Parameter mode	Type
aliased in	Access-to-constant
aliased out	Access-to-variable
aliased in out	Access-to-variable

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

```
package Data_Processing is

   procedure Proc (I : aliased in out Integer);

end Data_Processing;
```

Listing 86: data_processing.adb

```
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)

(continued from previous page)

```
      type Integer_Access is access all Integer;

      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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliased_
↪Rec_Component
MD5: 076238603036aa51cafcc013f38bc8f3
```

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[276]
- 6.2 Formal Parameter Modes[277]
- 6.4.1 Parameter Associations[278]

[276] http://www.ada-auth.org/standards/22rm/html/RM-6-1.html
[277] http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
[278] http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html

15.10 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[279]

15.10.1 Lifetime of objects

First, let's talk a bit about lifetime of objects[280]. We assume you understand the concept, so this section is very short.

In very simple terms, the lifetime of an object indicates when an object still has relevant information. For example, if a variable V gets out of scope, we say that its lifetime has ended. From this moment on, V no longer exists.

For example:

Listing 88: show_lifetime.adb

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

procedure Show_Lifetime is
   I_Var_1 : Integer := 22;
begin

   Inner_Block : declare
      I_Var_2 : Integer := 42;
   begin
      Put_Line ("I_Var_1: "
                & Integer'Image (I_Var_1));
      Put_Line ("I_Var_2: "
                & Integer'Image (I_Var_2));

      --  I_Var_2 will get out of scope
      --  when the block finishes.
   end Inner_Block;

   --  I_Var_2 is now out of scope...

   Put_Line ("I_Var_1: "
             & Integer'Image (I_Var_1));
   Put_Line ("I_Var_2: "
             & Integer'Image (I_Var_2));
   --                          ^^^^^^^
   --  ERROR: lifetime of I_Var_2 has ended!
end Show_Lifetime;
```

Code block metadata

[279] http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html
[280] https://en.wikipedia.org/wiki/Variable_(computer_science)#Scope_and_extent

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
↪Levels_Rules_Introduction.Lifetime
MD5: ebe36f12c832ecfe71399b89801808d4
```

Build output

```
show_lifetime.adb:24:31: error: "I_Var_2" is undefined
gprbuild: *** compilation phase failed
```

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.)

15.10.2 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:

```
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.

15.10.3 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 652), which we discuss later. Also, they are based on the *accessibility levels* (page 646) 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

```
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

```
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;
      --       ^^^^^^
      --       ILLEGAL: L2 object to
      --                L1 access object

      L2_IA := L2_Var'Access;
      --       ^^^^^^
```

(continues on next page)

(continued from previous page)

```
      --       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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Levels_Rules_Introduction.Accessibility_Library_Level
MD5: b3bed7eb2a8dfc78a2e7a7d2ce99f736
```

Build output

```
show_library_level.adb:20:16: error: non-local pointer cannot point to local object
show_library_level.adb:42:13: error: non-local pointer cannot point to local object
show_library_level.adb:47:15: error: non-local pointer cannot point to local object
show_library_level.adb:62:13: error: expected type "L0_Integer_Access" defined at library_level.ads:3
```

(continues on next page)

(continued from previous page)

```
show_library_level.adb:62:13: error: found type "L1_Integer_Access" defined at␣
↪line 4
show_library_level.adb:66:32: error: cannot convert local pointer to non-local␣
↪access type
gprbuild: *** compilation phase failed
```

In this example, we declare

- in the Library_Level package: the L0_Integer_Access type, the L0_IA access object, and the L0_Var aliased variable;
- in the Show_Library_Level procedure: the L1_Integer_Access type, the L0_IA_2 and L1_IA access objects, and the L1_Var aliased variable;
- in the nested Test procedure: the L2_Integer_Access type, the L2_IA, and the L2_Var aliased variable.

As mentioned earlier, the Ln prefix indicates the level of each type or object. Here, the n value is zero at library level. We then increment the n value each time we refer to a deeper level.

For instance:

- when we declare the L1_Integer_Access type in the Show_Library_Level procedure, that declaration is one level deeper than the level of the Library_Level package — so it has the L1 prefix.
- when we declare the L2_Integer_Access type in the Test procedure, that declaration is one level deeper than the level of the Show_Library_Level procedure — so it has the L2 prefix.

Types and Accessibility Levels

It's very important to highlight the fact that:

- types themselves also have an associated level, and
- objects have the same accessibility level as their types.

When we declare the L0_IA_2 object in the code example, its accessibility level is at library level because its type (the L0_Integer_Access type) is at library level. Even though this declaration is in the Show_Library_Level procedure — whose declarative part is one level deeper than the library level —, the object itself has the same accessibility level as its type.

Now that we've discussed the accessibility levels of this code example, let's see how the accessibility rules use those levels.

Operations on Access Types

In very simple terms, the accessibility rules say that:

- operations on access types at the same accessibility level are legal;
- assigning or converting to a deeper level is legal;

Otherwise, operations targeting objects at a *less-deep* level are illegal.

For example, `L0_IA := new Integer'(22)` and `L1_IA := L1_Var'Access` are legal because we're operating at the same accessibility level. Also, `L1_IA := L0_Var'Access` is legal because L1_IA is at a deeper level than `L0_Var'Access`.

However, many operations in the code example are illegal. For instance, `L0_IA := L1_Var'Access` and `L0_IA_2 := L1_Var'Access` are illegal because the target objects in the assignment are *less* deep.

Note that the L0_IA := L1_IA assignment is mainly illegal because the access types don't match. (Of course, in addition to that, assigning L1_Var'Access to L0_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 L0_Integer_Access (L1_IA) conversion is illegal because the resulting object is less deep. That being said, conversions on the same level are fine:

Listing 91: show_same_level_conversion.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
↪Levels_Rules_Introduction.Same_Level_Conversion
MD5: 7276a06e9f5b634d4f5a10a892071d87
```

Here, we're converting from the L1_Integer_Access type to the L1_B_Integer_Access, which are both at the same level.

15.10.4 Accessibility rules on parameters

Note that the accessibility rules also apply to access values as subprogram parameters. For example, compilation fails for this example:

Listing 92: names.ads

```
package Names is

   type Name is access all String;

   type Constant_Name is
     access constant String;

   procedure Show (N : Constant_Name);

end Names;
```

Listing 93: 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 94: show_names.adb

```
with Names; use Names;

procedure Show_Names is
   S : aliased String := "John";
begin
   Show (S'Access);
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
↪Levels_Rules_Introduction.Accessibility_Checks_Parameters
MD5: 6b8bf2799caa32f55d216ac0b58fcd39
```

Build output

```
show_names.adb:6:10: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed
```

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

```
with Names; use Names;

procedure Show_Names is
   S : Name := new String'("John");
begin
   Show (Constant_Name (S));
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
↪Levels_Rules_Introduction.Accessibility_Checks_Parameters
MD5: 30237c83426db758804b802e1953d5d9
```

Runtime output

```
Name: John
```

This version of the code works because both object and access object have the same level.

15.10.5 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 660).

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 521). 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

```
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
      --
      --  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: "
```

(continues on next page)

(continued from previous page)

```
                 & 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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
↪Levels_Rules_Introduction.Dangling_Reference_Rules
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 A2 := I_Var_2'Access assignment, however, both A2 and I_Var_2 have the same lifetime. In that sense, the assignment may actually look pretty much OK.
   - However, as mentioned in the previous section, Ada also cares about the lifetime of access types. In fact, since the Integer_Access type is declared outside of the Inner_Block, it has a longer lifetime than A2 and I_Var_2.
   - To be more precise, the accessibility rules detect that A2 is an access object of a type that has a longer lifetime than I_Var_2.

At first glance, this last accessibility rule may seem too strict, as both A2 and I_Var_2 have the same lifetime — so nothing bad could occur when dereferencing A2. However, consider the following change to the code:

```
A2 := I_Var_2'Access;

A1 := A2;
```

(continues on next page)

(continued from previous page)

```
--    PROBLEM: A1 will still be referring
--             to I_Var_2 after the
--             Inner_Block, i.e. when the
--             lifetime of I_Var_2 has
--             ended!
```

Here, we're introducing the A1 := A2 assignment. The problem with this is that I_Var_2's lifetime ends when the Inner_Block finishes, but A1 would continue to refer to an I_Var_2 object that doesn't exist anymore — thereby creating a dangling reference.

Even though we're actually not assigning A2 to A1 in the original code, we could have done it. The accessibility rules ensure that such an error is never introduced into the program.

For further reading...

In the original code, we can consider the A2 := I_Var_2'Access assignment to be safe, as we're not using the A1 := A2 assignment there. Since we're confident that no error could ever occur in the Inner_Block due to the assignment to A2, we could replace it with A2 := I_Var_2'Unchecked_Access, so that the compiler accepts it. We discuss more about the unchecked access attribute *later in this chapter* (page 654).

Alternatively, we could have solved the compilation issue that we see in the A2 := I_Var_2'Access assignment by declaring another access type locally in the Inner_Block:

```
Inner_Block : declare
   type Integer_Local_Access is
     access all Integer;

   I_Var_2 : aliased Integer := 42;

   A2      : Integer_Local_Access;
begin
   A2 := I_Var_2'Access;
   --   This assignment is fine because
   --   the Integer_Local_Access type has
   --   the same lifetime as I_Var_2.
end Inner_Block;
```

With this change, A2 becomes an access object of a type that has the same lifetime as I_Var_2, so that the assignment doesn't violate the rules anymore.

(Note that in the Inner_Block, we could have simply named the local access type Integer_Access instead of Integer_Local_Access, thereby masking the Integer_Access type of the outer block.)

We discuss the effects of dereferencing dangling references *later in this chapter* (page 662).

15.11 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 645) that the accessibility levels verify the lifetime of access types. Let's see a simplified version of a code example from that section:

Listing 97: integers.ads

```
package Integers is

   type Integer_Access is access all Integer;

end Integers;
```

Listing 98: show_access_issue.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Access.
↪Dangling_Reference_Rules
MD5: 646acabf3f388b52809349463d20d314
```

Build output

```
show_access_issue.adb:10:09: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed
```

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 the use Unchecked_Access attribute. For instance, we can use this attribute to circumvent the compilation error in the previous code example, since we know that the assignment is actually safe:

Listing 99: integers.ads

```
package Integers is

   type Integer_Access is access all Integer;

end Integers;
```

Listing 100: show_access_issue.adb

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

with Integers; use Integers;
```

(continues on next page)

(continued from previous page)

```

procedure Show_Access_Issue is
   I_Var : aliased Integer := 42;

   A     : Integer_Access;
begin
   A := I_Var'Unchecked_Access;
   --   OK: assignment is now accepted.

   Put_Line ("A.all: " & Integer'Image (A.all));
end Show_Access_Issue;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Access.
↪Dangling_Reference_Rules
MD5: a71b9076d9e2983ffb9811183afdf6c1
```

Runtime output

```
A.all:  42
```

When we use the Unchecked_Access attribute, most rules still apply. The only difference to the standard **Access** attribute is that unchecked access applies the rules as if the object we're getting access to was being declared at library level. (For the code example we've just seen, the check would be performed as if I_Var was declared in the Integers package instead of being declared in the procedure.)

It is strongly recommended to avoid unchecked access in general. You should only use it when you can safely assume that the access object will be discarded before the object we had access to gets out of scope. Therefore, if this situation isn't clear enough, it's best to avoid unchecked access. (Later in this chapter, we'll see some of the nasty issues that arrive from creating dangling references.) Instead, you should work on improving the software design of your application by considering alternatives such as using containers or encapsulating access types in well-designed abstract data types.

In the Ada Reference Manual

- Unchecked Access Value Creation[281]

15.12 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
```

(continues on next page)

[281] http://www.ada-auth.org/standards/22rm/html/RM-13-10.html

(continued from previous page)

```

   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↪Deallocation.Simple_Unchecked_Deallocation
MD5: 328b244cf406853e87494c381c9c4c9e
```

Here, we declare the Free procedure, which we can then use to deallocate objects that were allocated for the Integer_Access type.

Ada.Unchecked_Deallocation is a generic procedure that we can instantiate for access types. When declaring an instance of Ada.Unchecked_Deallocation, we have to specify arguments for:

- the formal Object parameter, which indicates the type of actual objects that we want to deallocate; and
- the formal Name parameter, which indicates the access type.

In a type declaration such as `type Integer_Access is access Integer`, `Integer` denotes the Object, while Integer_Access denotes the Name.

Because each instance of Ada.Unchecked_Deallocation is bound to a specific access type, we cannot use it for another access type, even if the type we use for the Object parameter is the same:

Listing 102: integer_types.ads

```
with Ada.Unchecked_Deallocation;

package Integer_Types is

   type Integer_Access is access Integer;

   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => Integer,
        Name   => Integer_Access);

   type Another_Integer_Access is access Integer;

   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => Integer,
        Name   => Another_Integer_Access);
end Integer_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↪Deallocation.Simple_Unchecked_Deallocation
MD5: b9bc58ff60632287237e2e322fcbc63e
```

Here, we're declaring two Free procedures: one for the Integer_Access type, another for the Another_Integer_Access. 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;
```

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);

```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↪Deallocation.Unchecked_Deallocation
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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↪Deallocation.Unchecked_Deallocation
MD5: ce7f4f912f12d723ca673ca36a478765
```

The multiple calls to Free for the same access object don't cause any issues. Because the access value is null after the first call to Free (I), we're actually just passing **null** as an argument in the second and third calls to Free. However, any attempt to deallocate an access value of null is ignored in the Free procedure, so the second and third calls to Free don't have any effect.

In the Ada Reference Manual

- 4.8 Allocators[282]
- 13.11.2 Unchecked Storage Deallocation[283]

15.12.1 Unchecked Deallocation and Dangling References

We've discussed *dangling references* (page 652) 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: show_unchecked_deallocation.adb

```
with Integer_Types; use Integer_Types;

procedure Show_Unchecked_Deallocation is

   I, I_2 : Integer_Access;

begin
   I := new Integer;

   I_2 := I;

   --  NOTE: I_2 points to the same
   --        object as I.

   --
   --  Use I and I_2...
   --
   --  ... then deallocate memory...
   --

   Free (I);

   --  NOTE: at this point, I_2 is a
   --        dangling reference!

   --  Further calls to Free (I)
   --  are OK!

   Free (I);
   Free (I);

   --  A call to Free (I_2) is
   --  NOT OK:

   Free (I_2);
end Show_Unchecked_Deallocation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↪Deallocation.Unchecked_Deallocation
MD5: ee5c20209a113a6c1bc7895b8ebdb174
```

[282] http://www.ada-auth.org/standards/22rm/html/RM-4-8.html
[283] http://www.ada-auth.org/standards/22rm/html/RM-13-11-2.html

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

```ada
with Integer_Types; use Integer_Types;

procedure Show_Unchecked_Deallocation is

   I, I_2 : Integer_Access;

begin
   I := new Integer;

   I_2 := I;

   --  NOTE: I_2 points to the same
   --        object as I.

   --
   --  Use I and I_2...
   --
   --  ... then deallocate memory...
   --

   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↪Deallocation.Unchecked_Deallocation
MD5: 3381ba594cbbc0f1547e3f819bae0f97
```

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 669) later on.

15.12.2 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:

Listing 109: show_unchecked_deallocation.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
 ↪Deallocation.Unchecked_Deallocation
MD5: 8536190aa5bbafa715ad8153aaeb4889
```

Runtime output

```
I_1.all =  42
I_2.all =  42
Freeing I_1
I_2.all =  6446
```

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[284]
- 13.11.2 Unchecked Storage Deallocation[285]

15.12.3 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:

Listing 110: show_unchecked_deallocation_errors.adb

```
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;
```

(continues on next page)

284 http://www.ada-auth.org/standards/22rm/html/RM-13-9-1.html
285 http://www.ada-auth.org/standards/22rm/html/RM-13-11-2.html

(continued from previous page)

```ada

   --  ERROR: Cannot use access-to-constant type
   --         for Name
   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => Integer,
        Name   => Constant_Integer_Access);

   I : Integer_Access_Zero;

begin
   --  ERROR: Cannot allocate objects from
   --         empty storage pool
   I := new Integer;

   --  ERROR: Cannot deallocate objects from
   --         empty storage pool
   Free (I);
end Show_Unchecked_Deallocation_Errors;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↪Deallocation.Unchecked_Deallocation_Error
MD5: 5032d13b2eb6b7ca1979282ddd6df98a
```

Build output

```
show_unchecked_deallocation_errors.adb:21:19: error: actual type must be access-to-
↪variable type
show_unchecked_deallocation_errors.adb:21:19: error: instantiation abandoned
show_unchecked_deallocation_errors.adb:28:09: error: allocation from empty storage␣
↪pool
show_unchecked_deallocation_errors.adb:32:04: error: deallocation from empty␣
↪storage pool
gprbuild: *** compilation phase failed
```

Here, we see that trying to instantiate `Ada.Unchecked_Deallocation` for the `Constant_Integer_Access` type is rejected by the compiler. Similarly, we cannot allocate or deallocate an object for the `Integer_Access_Zero` type because its storage pool is empty.

15.13 Null & Not Null Access

> **Note**
>
> This section was originally written by Robert A. Duff and published as Gem #23: Null Considered Harmful[286] and Gem #24[287].

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:

[286] https://www.adacore.com/gems/ada-gem-23
[287] https://www.adacore.com/gems/ada-gem-24

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↪Access.Null_Return
MD5: 6c4eed750d42685198ec9495805e3e23
```

An alternative design for Lookup would be to raise an exception:

Listing 112: show_not_found_exception.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↪Access.Not_Found_Exception
MD5: 6ef47b32d4923838ffc28f43e5db323c
```

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:

Listing 113: show_not_null_return.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↪Access.Not_Null_Return
MD5: 4c0bb95da3b5a7c555a763c4951f7e21
```

This is a complete package for the code snippets above:

Listing 114: example.ads

```
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

```
package body Example is

   An_Element : aliased Element;

   function Lookup (T : Table)
                    return Ref_Element is
      pragma Unreferenced (T);
   begin
      --  ...
      return Not_Found;
   end Lookup;

   function Lookup_2 (T : Table)
                      return not null Ref_Element
   is
   begin
      --  ...
      raise Not_Found_2;

      return An_Element'Access;
      --  suppress error: 'missing "return"
      --  statement in function body'
   end Lookup_2;

   procedure P (X : not null Ref_Element) is
   begin
      X.all.Component := X.all.Component + 1;
```

(continues on next page)

(continued from previous page)

```
   end P;

   procedure Q (X : not null Ref_Element) is
   begin
      for I in 1 .. 1000 loop
         P (X);
      end loop;
   end Q;

   procedure R is
   begin
      Q (An_Element'Access);
   end R;

  pragma Unreferenced (R);

end Example;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↪Access.Complete_Null_Return
MD5: 01895c7d5f843fd215dcc21d807d4187
```

In general, it's better to use the language proper for documentation, when possible, rather than comments, because compile-time and/or run-time checks can help ensure that the "documentation" is actually true. With comments, there's a greater danger that the comment will become false during maintenance, and false documentation is obviously a menace.

In many, perhaps most cases, **null** is just a tripping hazard. It's a good idea to put in **not null** when possible. In fact, a good argument can be made that **not null** should be the default, with extra syntax required when **null** is wanted. This is the way Standard ML[288] 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:

Listing 116: example-processing.ads

```
package Example.Processing is

   procedure P (X : not null Ref_Element);

   procedure Q (X : not null Ref_Element);

end Example.Processing;
```

Listing 117: example-processing.adb

```
package body Example.Processing is

```

(continues on next page)

[288] https://en.wikipedia.org/wiki/Standard_ML

(continued from previous page)

```ada
   procedure P (X : not null Ref_Element) is
   begin
      X.all.Component := X.all.Component + 1;
   end P;

   procedure Q (X : not null Ref_Element) is
   begin
      for I in 1 .. 1000 loop
         P (X);
      end loop;
   end Q;

end Example.Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↪Access.Complete_Null_Return
MD5: dc34b1a27737d57c041be6260dd577fd
```

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:

Listing 118: show_process_array.ads

```ada
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↪Access.Process_Array
MD5: 32424432f5b2e3013292680f92a04320
```

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.

15.14 Design strategies for access types

Previously, we learned about *dangling references* (page 652) and discussed the effects of *dereferencing them* (page 662). Also, we've seen the relationship between *unchecked deallocation and dangling references* (page 660). 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.

15.14.1 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 787).)

Let's see an example:

Listing 120: access_type_abstraction.ads

```ada
package Access_Type_Abstraction is

   type Info is limited private;

   function To_Info (S : String) return Info;

   function To_String (Obj : Info)
                       return String;

   function Copy (Obj : Info) return Info;

   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 (To_String (Obj)));

   procedure Copy (To   : in out Info;
                   From :        Info) is
   begin
      Destroy (To);
      To := Copy (From);
   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);
   begin
      Free (Obj);
   end Destroy;

end Access_Type_Abstraction;
```

Listing 122: main.adb

```
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 : "
```

(continues on next page)

(continued from previous page)

```
               & 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 ("----------");

   Put_Line ("APPEND");
   Append (Obj_1, " there");
   Put_Line ("Obj_1 : "
               & To_String (Obj_1));

   Destroy (Obj_1);
   Destroy (Obj_2);
end Main;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Design_Strategies.
 ↪Access_Type_Abstraction
MD5: a335caeba4f1fb952a2e0d8d6bc52f75
```

Runtime output

```
TO_INFO / COPY
Obj_1 : hello
```

(continues on next page)

(continued from previous page)

```
Obj_2 : hello
----------
RESET / APPEND
Obj_1 :
Obj_2 : hello world
----------
COPY
Obj_1 : hello world
Obj_2 : hello world
----------
DESTROY
Obj_1 :
Obj_2 :
----------
APPEND
Obj_1 : hey
----------
APPEND
Obj_1 : hey there
```

In this example, we hide an access type in the `Info` type — a limited private type. We allocate an object of this type in the `To_Info` function and deallocate it in the `Destroy` procedure. Also, we make sure that the reference isn't copied in the `Copy` function — we only copy the designated value in this function. This strategy eliminates the possibility of dangling references, as each reference is encapsulated in an object of `Info` type.

15.14.2 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* (page 835) in another chapter.)

Let's adapt the previous example and declare `Info` as a limited controlled type:

Listing 123: access_type_abstraction.ads

```
with Ada.Finalization;

package Access_Type_Abstraction is

   type Info is limited private;

   function To_Info (S : String) return Info;

   function To_String (Obj : Info)
                       return String;

   function Copy (Obj : Info) return Info;

   procedure Copy (To   : in out Info;
                   From :        Info);

   procedure Append (Obj : in out Info;
                     S   :        String);

   procedure Reset (Obj : in out Info);

private

```

(continues on next page)

(continued from previous page)

```
   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;
```

Listing 124: access_type_abstraction.adb

```
with Ada.Unchecked_Deallocation;

package body Access_Type_Abstraction is

   --
   --  STRING_ACCESS SUBPROGRAMS
   --

   function To_String_Access (S : String)
                              return String_Access
   is
     (new String'(S));

   function To_String (S : String_Access)
                       return String is
     (if S /= null then S.all else "");

   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => String,
        Name   => String_Access);

   --
   --  PRIVATE SUBPROGRAMS
   --

   procedure Initialize (Obj : in out Info) is
   begin
      --  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;

```

(continues on next page)

(continued from previous page)

```
   --
   --  PUBLIC SUBPROGRAMS
   --

   function To_Info (S : String) return Info is
     (Ada.Finalization.Limited_Controlled
      with Str_A => To_String_Access (S));

   function To_String (Obj : Info)
                       return String is
     (To_String (Obj.Str_A));

   function Copy (Obj : Info) return Info is
     (To_Info (To_String (Obj.Str_A)));

   procedure Copy (To   : in out Info;
                   From :        Info) is
   begin
      Free (To.Str_A);
      To.Str_A := To_String_Access
                    (To_String (From.Str_A));
   end Copy;

   procedure Append (Obj : in out Info;
                     S   :        String) is
      New_Str_A : constant String_Access :=
                    To_String_Access
                      (To_String (Obj.Str_A) & S);
   begin
      Free (Obj.Str_A);
      Obj.Str_A := New_Str_A;
   end Append;

   procedure Reset (Obj : in out Info) is
   begin
      Free (Obj.Str_A);
   end Reset;

end Access_Type_Abstraction;
```

Listing 125: main.adb

```
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 ("----------");
```

(continues on next page)

(continued from previous page)

```

   --
   --  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));
   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"
   --
```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Design_Strategies.
↪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
Obj_1 :
Obj_2 :
----------
COPY
Obj_1 :
Obj_2 :
----------
APPEND
Obj_1 : hey
----------
APPEND
Obj_1 : hey there
```

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.)

15.15 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.

15.15.1 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

```
procedure P (I : in out Integer);
```

Listing 127: p.adb

```
procedure P (I : in out Integer) is
begin
   null;
end P;
```

Listing 128: proc.adb

```
with P;

procedure Proc is
   I : Integer := 0;
begin
   P (I);
end Proc;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Subprogram_Call
MD5: 0e9547e53d0d02d39920f4d1d6787af6
```

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`.)

15.15.2 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:

Listing 129: access_to_subprogram_types.ads

```
package Access_To_Subprogram_Types is

   type Access_To_Procedure is
     access procedure (I : in out Integer);

   type Access_To_Function is
     access function (I : Integer) return Integer;

end Access_To_Subprogram_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: 5f834c1b2044ba5ea7d4835c3ebdedb1
```

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:

Listing 130: access_to_subprogram_params.ads

```
with Access_To_Subprogram_Types;
use  Access_To_Subprogram_Types;

package Access_To_Subprogram_Params is

   procedure Proc (P : Access_To_Procedure);

end Access_To_Subprogram_Params;
```

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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
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
   --              to Proc
end Show_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: 599e9d1306da48e3c532692b34c02a1d
```

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[289]

15.15.3 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:

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

Listing 135: add_twenty.ads

```
procedure Add_Twenty (I : in out Integer);
```

Listing 136: add_twenty.adb

```
procedure Add_Twenty (I : in out Integer) is
begin
   I := I + 20;
end Add_Twenty;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: 697959b806f6f2bfba248ec15c47883b
```

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

```
with Access_To_Subprogram_Types;
use  Access_To_Subprogram_Types;

with Access_To_Subprogram_Params;
use  Access_To_Subprogram_Params;

with Add_Ten;
with Add_Twenty;

procedure Show_Access_To_Subprograms is
   P        : Access_To_Procedure;
   Some_Int : Integer := 0;
begin
   P := Add_Ten'Access;
   --          ^ Getting access to Add_Ten
   --            procedure and assigning it
   --            to P

   Proc (P);
   --    ^ Passing access-to-subprogram as an
   --      actual parameter

   P (Some_Int);
   --  ^ Using access-to-subprogram object in a
   --    subprogram call

   P := Add_Twenty'Access;
   --             ^ Getting access to Add_Twenty
   --               procedure and assigning it
   --               to P

   Proc (P);
   P (Some_Int);
end Show_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
```

(continues on next page)

(continued from previous page)

```
MD5: 7b4ea19187806e88ba65847876cafb4f
```

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.

15.15.4 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:

Listing 138: 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;

   type Access_To_Procedure_Array is
     array (Positive range <>) of
       Access_To_Procedure;

   type Access_To_Function_Array is
     array (Positive range <>) of
       Access_To_Function;

   type Rec_Access_To_Procedure is record
      AP : Access_To_Procedure;
   end record;

   type Rec_Access_To_Function is record
      AF : Access_To_Function;
   end record;

end Access_To_Subprogram_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: 32203838b97af66ef6ca3f6b1ce646a5
```

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:

Listing 139: show_access_to_subprograms.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: f1d10056b4b3424bd30d954f34caa255
```

Runtime output

```
Some_Int:  0
Some_Int:  10
Some_Int:  30
```

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:

Listing 140: show_access_to_subprograms.adb

```
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);
```

(continues on next page)

(continued from previous page)

```
   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: 4b23b5f6a8c252a1a014a2b54fa32c1a
```

Runtime output

```
Some_Int:  0
Some_Int:  10
Some_Int:  30
```

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.

15.15.5 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 603) 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

```
package Custom_Processing is

   --  Declaring an access type:
   type Integer_Processing is
     access procedure (I : in out Integer);

   --  Declaring a discriminant with this
   --  access type:
   type Rec (IP : Integer_Processing) is
     private;

   procedure Init (R     : in out Rec;
                   Value :        Integer);

   procedure Process (R : in out Rec);

   procedure Show (R : Rec);

private

   type Rec (IP : Integer_Processing) is
   record
      I : Integer := 0;
   end record;

end Custom_Processing;
```

Listing 142: custom_processing.adb

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

package body Custom_Processing is

   procedure Init (R     : in out Rec;
                   Value :        Integer) is
   begin
      R.I := Value;
   end Init;

   procedure Process (R : in out Rec) is
   begin
      R.IP (R.I);
      --  ^^^^^^
      --  Calling procedure that we specified as
      --  the record's discriminant
   end Process;

   procedure Show (R : Rec) is
   begin
      Put_Line ("R.I = "
                & Integer'Image (R.I));
   end Show;

end Custom_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Access_To_Subprogram_Types
MD5: 02fc0c51722c321c4ec6115de68d1c06
```

In this example, we declare the access-to-subprogram type `Integer_Processing`, which we use as the `IP` discriminant of the `Rec` type. In the `Process` procedure, we call the `IP` procedure that we specified as the record's discriminant (`R.IP (R.I)`).

Before we look at a test application for this package, let's implement another small procedure:

Listing 143: mult_two.ads

```
procedure Mult_Two (I : in out Integer);
```

Listing 144: mult_two.adb

```
procedure Mult_Two (I : in out Integer) is
begin
   I := I * 2;
end Mult_Two;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Access_To_Subprogram_Types
MD5: cd43fa39dac9a1c9182f69d32eab1d26
```

Now, let's look at the test application:

Listing 145: show_access_to_subprogram_discriminants.adb

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

with Custom_Processing; use Custom_Processing;

with Add_Ten;
with Mult_Two;

procedure Show_Access_To_Subprogram_Discriminants
is

   R_Add_Ten  : Rec (IP => Add_Ten'Access);
   --                 ^^^^^^^^^^^^^^^^^^^^
   --       Using access-to-subprogram as a
   --       discriminant

   R_Mult_Two : Rec (IP => Mult_Two'Access);
   --                 ^^^^^^^^^^^^^^^^^^^^^
   --       Using access-to-subprogram as a
   --       discriminant

begin
   Init (R_Add_Ten,  1);
   Init (R_Mult_Two, 2);

   Put_Line ("---- R_Add_Ten ----");
   Show (R_Add_Ten);

   Put_Line ("Calling Process procedure...");
   Process (R_Add_Ten);
   Show (R_Add_Ten);

   Put_Line ("---- R_Mult_Two ----");
   Show (R_Mult_Two);

   Put_Line ("Calling Process procedure...");
   Process (R_Mult_Two);
   Show (R_Mult_Two);
end Show_Access_To_Subprogram_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
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.

15.15.6 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
   type T is private;

   --
   --  Declaring formal access-to-subprogram
   --  type:
   --
   type T_Processing is
     access procedure (Element : in out T);

   --
   --  Declaring formal access-to-subprogram
   --  parameter:
   --
   Proc : T_Processing;

   with function Image_T (Element : T)
                          return String;
package Gen_Custom_Processing is

   type Rec is private;

   procedure Init (R     : in out Rec;
                   Value :        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;

```

(continues on next page)

(continued from previous page)

```
   procedure Show (R : Rec) is
   begin
      Put_Line ("R.Comp = "
                & Image_T (R.Comp));
   end Show;

end Gen_Custom_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: 6f06e066bafa5f02abb3ee1b33ea0831
```

In this version of the procedure, instead of declaring Proc as a discriminant of the Rec record, we're declaring it as a formal parameter of the Gen_Custom_Processing package. Also, we're declaring an access-to-subprogram type (T_Processing) as a formal parameter. (Note that, in contrast to these two parameters that we've just mentioned, Image_T is not a formal access-to-subprogram parameter: it's actually just a formal subprogram.)

We then instantiate the Gen_Custom_Processing package in our test application:

Listing 148: show_access_to_subprogram_as_formal_parameter.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_To_Subprogram_Types
MD5: 6ae27ebd59e5307551e9a38f3b94c70c
```

Runtime output

```
---- R_Add_Ten ----
R.Comp =  1
Calling Process procedure...
R.Comp =  11
```

Here, we instantiate the `Gen_Custom_Processing` package as `Custom_Processing` and specify the access-to-subprogram type and the access-to-subprogram.

15.15.7 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Log_Procedure
MD5: 59399e0809deb476f608faab7e4398bd
```

In the implementation of Process, we check whether Log_Proc is null or not. (If it's not null, we call the procedure. Otherwise, we just skip the call.)

Now, let's implement two logging procedures that match the expected form of the Log_Procedure type:

Listing 151: log_element_per_line.adb

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

procedure Log_Element_Per_Line
  (D : Data_Container) is
begin
   Put_Line ("Elements: ");
   for V of D loop
      Put_Line (V'Image);
   end loop;
   Put_Line ("------");
end Log_Element_Per_Line;
```

Listing 152: log_csv.adb

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

procedure Log_Csv (D : Data_Container) is
begin
   for I in D'First .. D'Last - 1 loop
      Put (D (I)'Image & ", ");
   end loop;
   Put (D (D'Last)'Image);
   New_Line;
end Log_Csv;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Log_Procedure
MD5: 468789f7331ffcd16f754f7116b076d7
```

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

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

(continues on next page)

(continued from previous page)

```
   Process (D);
end Show_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Log_Procedure
MD5: 134aa682cea1999efa0ea97052f315c8
```

Runtime output

```
==== Log_Element_Per_Line ====
Elements:
 1.00000E+00
 1.00000E+00
 1.00000E+00
 1.00000E+00
 1.00000E+00
------
==== Log_Csv ====
 1.00000E+00,  1.00000E+00,  1.00000E+00,  1.00000E+00,  1.00000E+00
==== None ====
```

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.

15.15.8 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

```
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

```
package body Data_Processing is

   procedure Process
     (D         : in out Data_Container;
      Init_Func :        Init_Function) is
   begin
      for I in D'Range loop
```

(continues on next page)

(continued from previous page)

```
         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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_Init_Function
MD5: 444110d50ddb430fd5be31cf1b417fc8
```

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;
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("------");
end Log_Element_Per_Line;
```

Listing 161: show_access_to_subprograms.adb

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

with Init_Zero;
with Init_One;

with Log_Element_Per_Line;

procedure Show_Access_To_Subprograms is
   D : Data_Container (1 .. 5) := (others => 1.0);
begin
   Put_Line ("==== Init_Zero ====");
   Process (D, Init_Zero'Access);
   Log_Element_Per_Line (D);

   Put_Line ("==== Init_One ====");
   Process (D, Init_One'Access);
   Log_Element_Per_Line (D);

   --  Put_Line ("==== None ====");
   --  Process (D, null);
   --  Log_Element_Per_Line (D);
end Show_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Access_Init_Function
MD5: ae0e3fd58e9bb83061248967c709190a
```

Runtime output

```
==== Init_Zero ====
Elements:
 0.00000E+00
 0.00000E+00
 0.00000E+00
 0.00000E+00
 0.00000E+00
------
==== Init_One ====
Elements:
 1.00000E+00
 1.00000E+00
 1.00000E+00
 1.00000E+00
 1.00000E+00
------
```

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[290].

Here's another example, first with **null**:

Listing 162: 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

   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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪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:

Listing 164: example.ads

```
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;
```

Listing 165: example.adb

```
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;

end Example;
```

Listing 166: show_example.adb

```
with Example; use Example;

procedure Show_Example is
   T : Table;
begin
   Iterate_2 (T);
end Show_Example;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Complete_Not_Null_Procedure
MD5: ab0a41e0d39a8a16b0b69f8c6b2a43fd
```

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.

15.15.9 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`:

[290] https://www.adacore.com/gems/ada-gem-24

Listing 167: simple_protected_access.ads

```
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

```
package body Simple_Protected_Access is

   protected body Obj is

      procedure Do_Something is
      begin
         --  Not doing anything
         --  for the moment...
         null;
      end Do_Something;

   end Obj;

end Simple_Protected_Access;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Simple_Protected_Access
MD5: d82f7c90355e9810bd1e35f65e278626
```

Here, we declare the Access_Proc type as an access type to protected procedures. Then, we declare the variable Acc and assign to it the access to the Do_Something procedure (of the protected object Obj).

Now, let's discuss a more useful example: a simple system that allows us to register protected procedures and execute them. This is implemented in Work_Registry package:

Listing 169: work_registry.ads

```
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;
```

(continues on next page)

(continued from previous page)

```

   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

```
package body Work_Registry is

   protected body Work_Handler_Registry is

      procedure Register (T : Valid_Work_Handler)
      is
      begin
         if Curr < Last then
            Curr := Curr + 1;
            D (Curr) := T;
         end if;
      end Register;

      procedure Reset is
      begin
         Curr := 0;
      end Reset;

      procedure Process_All is
         Dummy_ID : Work_Id;
      begin
         for I in D'First .. Curr loop
            D (I).all (Dummy_ID);
         end loop;
      end Process_All;

   end Work_Handler_Registry;

end Work_Registry;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Protected_Access_Init_Function
MD5: 5dfa8ab098900ab4f6b7575e1cde5e53
```

Here, we declare the protected Work_Handler_Registry type with the following subprograms:

- Register, which we can use to register a protected procedure;
- Reset, which we can use to reset the system; and
- Process_All, which we can use to call all procedures that were registered in the system.

Work_Handler is our access to protected subprogram type. Also, we declare the Valid_Work_Handler subtype, which excludes **null**. By doing so, we can ensure that only valid procedures are passed to the Register procedure. In the protected Work_Handler_Registry type, we store the procedures in an array (of Work_Handlers type).

Important

Note that, in the type declaration Work_Handler, we say that the protected procedure must have a parameter of Work_Id type. In this example, this parameter is just used to *bind* the procedure to the Work_Handler_Registry type. The Work_Id type itself is actually declared as a null record (in the private part of the package), and it isn't really useful on its own.

If we had declared **type Work_Handler is access protected procedure**; instead, we would be able to register *any* protected procedure into the system, even the ones that might not be suitable for the system. By using a parameter of Work_Id type, however, we make use of strong typing to ensure that only procedures that were designed for the system can be registered.

In the next part of the code, we declare the Integer_Storage type, which is a simple protected type that we use to store an integer value:

Listing 171: integer_storage_system.ads

```
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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Protected_Access_Init_Function
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):

Listing 173: show_access_to_protected_subprograms.adb

```
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
```

(continues on next page)

(continued from previous page)

```
      Int_Stor (I) := new Integer_Storage;
      Int_Stor (I).Set (I);
   end loop;

   --  Register handlers

   for I in Int_Stor'Range loop
      WHR.Register (Int_Stor (I).all.Show'Access);
   end loop;

   --  Now, use Process_All to call the handlers
   --  (in this case, the Show procedure for
   --  each protected object from Int_Stor).

   WHR.Process_All;

end Show_Access_To_Protected_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Protected_Access_Init_Function
MD5: 44c24ef07333e1d31844cc2ea6d91ab6
```

Runtime output

```
Value:  1
Value:  2
Value:  3
```

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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↪Subprograms.Protected_Access_Init_Function
MD5: 359241c84cd30313fe2d7701b55f303e
```

Build output

```
show_access_to_protected_subprograms.adb:28:21: error: non-local pointer cannot␣
↪point to local object
gprbuild: *** compilation phase failed
```

As we've just discussed, this error is due to the fact that Int_Stor is now a "local" protected object, and the accessibility rules don't allow mixing it with non-local accesses in order to prevent the possibility of dangling references.

When we call `WHR.Process_All`, the registry system calls each procedure that has been registered with the system. When looking at the values displayed by the test application, we may notice that each call to Show is referring to a different protected object. In fact, even though we're passing just the access to a protected *procedure* in the call to `Register`, that access is also associated to a specific protected object. (This is different from access to non-protected subprograms we've discussed previously: in that case, there's no object associated.) If we replace the argument to `Register` by `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.

15.16 Accessibility Rules and Access-To-Subprograms

In general, the accessibility rules that we discussed *previously for access-to-objects* (page 645) 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 646) and the *accessibility rules* (page 647) that are based on those levels. The same accessibility rules apply to access-to-subprograms. *As we said previously* (page 649), operations targeting objects at a *less-deep* level are illegal, as it's the case for subprograms as well:

Listing 175: access_to_subprogram_types.ads

```
package Access_To_Subprogram_Types is

   type Access_To_Procedure is
     access procedure (I : in out Integer);

   type Access_To_Function is
     access function (I : Integer) return Integer;

end Access_To_Subprogram_Types;
```

Listing 176: 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;

   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);
   Put_Line ("Value: " & Value'Image);
end Show_Access_To_Subprogram_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Rules_
↪Access_To_Subprograms.Access_To_Subprogram_Accessibility_Error_Less_Deep
```

(continues on next page)

(continued from previous page)

```
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;
```

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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Rules_
↪Access_To_Subprograms.Access_To_Subprogram_Accessibility_Error_Less_Deep_Fix
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.

15.16.1 Unchecked Access

Previously, we discussed about the *Unchecked_Access attribute* (page 654), which we can use to circumvent accessibility issues in specific cases for access-to-objects. We also said in that section that this attribute only exists for objects, not for subprograms. We can use the previous example to illustrate this limitation:

Listing 181: access_to_subprogram_types.ads

```ada
package Access_To_Subprogram_Types is

   type Access_To_Procedure is
     access procedure (I : in out Integer);

   type Access_To_Function is
     access function (I : Integer) return Integer;

end Access_To_Subprogram_Types;
```

Listing 182: show_access_to_subprogram_error.adb

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

with Access_To_Subprogram_Types;
use  Access_To_Subprogram_Types;

procedure Show_Access_To_Subprogram_Error is
   Func : Access_To_Function;

   function Add_One (I : Integer)
            return Integer is
     (I + 1);

   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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Rules_
 ↪Access_To_Subprograms.Access_To_Subprogram_Accessibility_Error_Same_Lifetime
MD5: c1ee1946f0c979eb30fbf2c72c426f50
```

Build output

```
show_access_to_subprogram_error.adb:15:12: error: subprogram must not be deeper␣
 ↪than access type
```

(continues on next page)

(continued from previous page)

```
gprbuild: *** compilation phase failed
```

When we analyze the Show_Access_To_Subprogram_Error procedure, we see that the Func object and the Add_One function have the same lifetime. Therefore, in this very specific case, we could safely assign Add_One'Access to Func and call Func for Value. Due to the accessibility rules, however, this assignment is illegal. (Obviously, the accessibility issue here is that the Access_To_Function type has a potentially longer lifetime.)

In the case of access-to-objects, we could use Unchecked_Access to enforce assignments that we consider safe after careful analysis. However, because this attribute isn't available for access-to-subprograms, the best solution is to move the subprogram to a level that allows the assignment to be legal, as we said before.

In the GNAT toolchain

GNAT offers an equivalent for Unchecked_Access that can be used for subprograms: the Unrestricted_Access attribute. Note, however, that this attribute is not portable.

Listing 183: access_to_subprogram_types.ads

```ada
package Access_To_Subprogram_Types is

   type Access_To_Procedure is
     access procedure (I : in out Integer);

   type Access_To_Function is
     access function (I : Integer) return Integer;

end Access_To_Subprogram_Types;
```

Listing 184: show_access_to_subprogram_error.adb

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

with Access_To_Subprogram_Types;
use  Access_To_Subprogram_Types;

procedure Show_Access_To_Subprogram_Error is
   Func : Access_To_Function;

   function Add_One (I : Integer)
            return Integer is
     (I + 1);

   Value : Integer := 0;
begin
   Func := Add_One'Unrestricted_Access;
   --                ^^^^^^^^^^^^^^^^^^^
   --        Allowing access to local function

   Put_Line ("Value: " & Value'Image);
   Value := Func (Value);
   Put_Line ("Value: " & Value'Image);
end Show_Access_To_Subprogram_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
↪Rules_Access_To_Subprograms.Unrestricted_Access
MD5: 90e2c57c01463cbe6efee6e093d01e5b
```

Runtime output

```
Value:  0
Value:  1
```

As we can see, the `Unrestricted_Access` attribute can be safely used in this specific case to circumvent the accessibility rule limitation.

15.17 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 127). 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.

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes[291]
- 13.7 The Package System[292]

15.17.1 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

```
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 : "
```

(continues on next page)

[291] http://www.ada-auth.org/standards/22rm/html/RM-13-3.html
[292] http://www.ada-auth.org/standards/22rm/html/RM-13-7.html

(continued from previous page)

```
             & 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Address.
↪Address_Conversion
MD5: 717532026247044a667b60f6c1e1c7da
```

Runtime output

```
I'Address : 00007FFD4132F434
AI.all'Address : 00007FFD4132F434
To_Address (AI) : 00007FFD4132F434
```

In this example, we instantiate the generic `System.Address_To_Access_Conversions` package using **Integer** as our target object type. This new package (`Integer_AAC`) has an `Object_Pointer` type, which is equivalent to a declaration such as **type Integer_Access is access all Integer**. (In this example, we declare `Integer_Access` as a subtype of `Integer_AAC.Object_Pointer` to illustrate that.)

The `Integer_AAC` package also includes the `To_Address` function, which converts an access object to an address. If the actual parameter is not null, `To_Address` returns the same information as if we were using the **Address** attribute for the designated object. In other words, `To_Address (AI) = AI.all'Address` when `AI /= null`.

If the access value is null, `To_Address` returns `Null_Address`, while `.all'Address` makes the *access check* (page 514) fail because we have to dereference the access object (via `.all`) before retrieving its address (via the **Address** attribute).

In addition to the `To_Address` function, the `To_Pointer` function is available to convert from an address to an object of access type. For example:

Listing 186: show_address_conversion.adb

```
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;
```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Address.
↪Address_Conversion
MD5: 5c6fc19ca1aa227feba97ea610dd9218
```

Runtime output

```
AI_1.all'Address : 00007FFEECEED46C
AI_2.all'Address : 00007FFEECEED46C
AI_1 = AI_2
```

Here, we convert the A address back to an access value by calling `To_Pointer (A)`. (When running this object, we see that AI_1 and AI_2 have the same access value.)

Conversion of unbounded designated types

Note that the conversions might not work in all cases. For instance, when the designated type — indicated by the formal `Object` parameter of the generic `Address_To_Access_Conversions` package — is unbounded, the result of a call to `To_Pointer` may not have bounds.

Let's adapt the previous code example and replace the **`Integer`** type by the (unbounded) **`String`** type:

Listing 187: show_address_conversion.adb

```
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;

```

(continues on next page)

(continued from previous page)

```ada
   subtype Integer_Access is
     String_AAC.Object_Pointer;

   S          : aliased String := "Hello";
   AI_1, AI_2 : Integer_Access;
   A          : Address;
begin
   AI_1 := S'Access;
   A    := To_Address (AI_1);

   AI_2 := To_Pointer (A);
   --      ^^^^^^^^^^^^^^
   --   WARNING: Result might not have bounds

   Put_Line ("AI_1.all'Address : "
             & System.Address_Image
                 (AI_1.all'Address));
   Put_Line ("AI_2.all'Address : "
             & System.Address_Image
                 (AI_2.all'Address));

   if AI_1 = AI_2 then
      Put_Line ("AI_1 = AI_2");
   else
      Put_Line ("AI_1 /= AI_2");
   end if;

   Put_Line ("AI_1: " & AI_1.all);
   Put_Line ("AI_2: " & AI_2.all);
   --                 ^^^^^^^^^^
   --   WARNING: As AI_2 might not have bounds
   --            due to the call to To_Pointer
   --            the behavior of this call to
   --            the "&" operator is
   --            unpredictable.
end Show_Address_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Address.Address_Conversion
MD5: b1adcaa1f2cb4dfbd157aebf7893bd72
```

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 : 00007FFE11929348
AI_2.all'Address : 00007FFE11929348
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[293]

[293] http://www.ada-auth.org/standards/22rm/html/RM-13-7-2.html

CHAPTER

SIXTEEN

ANONYMOUS ACCESS TYPES

16.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```
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:

```
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 735) and it's specifying an *anonymous access-to-object type* (page 715). Another flavor of anonymous access types are *anonymous access-to-subprograms* (page 758). We discuss all these topics in more details later.

Let's see a complete example:

Listing 1: show_anonymous_access_types.adb

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

procedure Show_Anonymous_Access_Types is
   I_Var : aliased Integer;

   A     : access Integer;
   --       ^ Anonymous access type
begin
   A := I_Var'Access;
   --   ^ Assignment to object of
   --     anonymous access type.

   A.all := 22;

   Put_Line ("A.all: " & Integer'Image (A.all));
end Show_Anonymous_Access_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Types.Simple_Anonymous_Access_Types
MD5: f0c92c76d970089c1d503c599d6869dd
```

Runtime output

```
A.all:  22
```

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[294]

16.1.1 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 718).)

16.1.2 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

```
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;
```

[294] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

Listing 3: show_anonymous_access_types.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Types.Anonymous_Access_Object_Assignment
MD5: 2822ca0bd6ac251dccc1ced60747fbe1
```

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.

(Note that the use of an access parameter in the Show procedure is for demonstration purpose only: a simply **Integer** as the type of this input parameter would have been more than sufficient to implement the procedure. Actually, in this case, avoiding the access parameter would be the recommended approach in terms of clean Ada software design.)

In contrast, if we had used named type declarations, the code would be more complicated and more limited:

Listing 4: aux.ads

```
package Aux is

   type Integer_Access is access all Integer;

   procedure Show (Name : String;
                   V    : Integer_Access);

end Aux;
```

Listing 5: aux.adb

```
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)

(continued from previous page)

```
      Put_Line (Name & ".all: "
                & Integer'Image (V.all));
   end Show;

end Aux;
```

Listing 6: show_anonymous_access_types.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Types.Anonymous_Access_Object_Assignment
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 indicate in the comments.

This restriction regarding non-local pointer assignments is an example of the stricter accessibility rules that apply to named access types. As mentioned earlier, the `S'Access` attribute always works when we use anonymous access types — this is not always the case for named access types.

Important

As mentioned earlier, if we want to use two access objects in an operation, the rule says that the subtype S of the anonymous type used in their corresponding declaration must match. In the following example, we can see how this rule works:

Listing 7: show_anonymous_access_subtype_error.adb

```
procedure Show_Anonymous_Access_Subtype_Error is
   subtype Integer_1_10 is Integer range 1 .. 10;

   I_Var : aliased Integer;
   A     : access Integer := I_Var'Access;
   B     : access Integer_1_10;
begin
   A := I_Var'Access;

   B := A;
   --  ^ ERROR: subtype doesn't match!

   B := I_Var'Access;
   --  ^ ERROR: subtype doesn't match!
end Show_Anonymous_Access_Subtype_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_
↪Types.Anonymous_Access_Types.Anonymous_Access_Subtype_Error
MD5: cecfe703ea8b42bad61c45f33cbcb67b
```

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 715) and *anonymous access-to-subprograms* (page 758) cover more specific details on anonymous access types.

16.2 Anonymous Access-To-Object Types

In the *previous chapter* (page 593), we introduced named access-to-object types and used those types throughout the chapter. Also, in the *previous section* (page 711), we've seen some simple examples of anonymous access-to-object types:

```
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 725) and *anonymous access parameters* (page 735) later on.) Let's see a code example that includes all these cases:

Listing 8: all_anonymous_access_to_object_types.ads

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Object_Types.All_Anonymous_Access_To_Object_Types
MD5: 6533b22a4e4526702320cb327bf6f69a
```

In this example, we see multiple examples of anonymous access-to-object types:

- as the A parameter of the Add_One procedure;
- in the declaration of the AI access object;
- as the AI discriminant of the Rec type;
- as the component type of the Access_Array type;
- as the component type of the Arr array;
- in the AI_Renaming renaming;
- as the return type of the Init_Access_Integer;
- as the Internal_AI of component of the Rec type.

In the Ada Reference Manual

- 3.10 Access Types[295]

16.2.1 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

```

(continues on next page)

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

(continued from previous page)

```
   end record;

end All_Anonymous_Access_To_Object_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Object_Types.All_Not_Null_Anonymous_Access_To_Object_Types
MD5: 027430aa9d5e19979206110f5e260d13
```

As you might have noticed, we took the previous code example and used **not null** for each usage instance of the anonymous access type. In this sense, this version of the code example is very similar to the previous one. Note, however, that we now have to explicitly initialize some elements to avoid the `Constraint_Error` exception being raised at runtime. This is the case for example for the `AI` access object:

```
AI : not null access Integer := I'Access;
```

If we hadn't initialized `AI` explicitly with `I'Access`, it would have been set to **null**, which would fail the **not null** constraint of the anonymous access type. Similarly, we also have to initialize the `Arr` array and return a valid access object for the `Init_Access_Integer` function.

16.2.2 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:

Listing 10: missing_features.ads

```
with Ada.Unchecked_Deallocation;

package Missing_Features is

   --  We cannot specify which pool will be used
```

(continues on next page)

(continued from previous page)

```
   --  in the anonymous access-to-object
   --  allocation; the pool is selected by the
   --  compiler:
   IA : access Integer := new Integer;

   --
   --  All the features below aren't available
   --  for an anonymous access-to-object:
   --

   --  Having a specific storage pool associated
   --  with the access type:
   type String_Access is
     access String;
   --  Automatically creates
   --  String_Access'Storage_Pool

   type Integer_Access is
     access Integer
       with Storage_Pool =>
              String_Access'Storage_Pool;
   --         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   --           Using the pool from another
   --           access type.

   --  Specifying a deallocation function for the
   --  access type:
   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => Integer,
        Name   => Integer_Access);

   --  Specifying a limited storage size for
   --  the access type:
   type Integer_Access_Store_128 is
      access Integer
        with Storage_Size => 128;

   --  Limiting the storage size for the
   --  access type to zero:
   type Integer_Access_Store_0 is
      access Integer
        with Storage_Size => 0;

end Missing_Features;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Object_Types.Missing_Anonymous_Access_To_Object_Features
MD5: 87a5c1413a720da84fab414cf63236ec
```

In the Missing_Features package, we see some of the features that we cannot use for the anonymous **access Integer** type, but that are available for equivalent named access types:

- There's no specific memory pool associated with the access object IA. In contrast, named types — such as String_Access and Integer_Access — have an associated pool, and we can use the Storage_Pool aspect and the Storage_Pool attribute to customize them.
- We cannot instantiate the Ada.Unchecked_Deallocation procedure for the **access**

Integer type. However, we can instantiate it for named access types such as the Integer_Access type.

- We cannot use the Storage_Size attribute for the access Integer type, but we're allowed to use it with named access types, which we do in the declaration of the Integer_Access_Store_128 and Integer_Access_Store_0 types.

Dangerous memory deallocation

We might think that we could make up for the absence of the Ada.Unchecked_Deallocation procedure for anonymous access-to-object types by converting those access objects (of anonymous access types) to a named type that has the same designated subtype. For example, if we have an access object IA of an anonymous access Integer type, we can convert it to the named Integer_Access type, provided this named access type is compatible with the anonymous access type, e.g.:

```
type Integer_Access is access all Integer
```

Let's see a complete code example:

Listing 11: show_dangerous_deallocation.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Object_Types.Deallocation_Anonymous_Access_To_Object_Erroneous
MD5: 91e024a4338e2e4f8d5b308d95499c1c
```

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

```
pragma Restrictions (No_Anonymous_Allocators);

procedure Show_Dangerous_Allocation is
   IA : access Integer;
begin
   IA := new Integer;
   IA.all := 30;
end Show_Dangerous_Allocation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Anonymous_Access_To_Object_Types.No_Anonymous_Allocators
MD5: 0976821ce632f9635e33fd4f79c81ecd
```

Build output

```
show_dangerous_allocation.adb:6:10: error: violation of restriction "No_
↪Anonymous_Allocators" at line 1
gprbuild: *** compilation phase failed
```

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

```
with Ada.Unchecked_Deallocation;

procedure Show_Successful_Deallocation is

   type Integer_Access is
      access Integer;

   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => Integer,
        Name   => Integer_Access);

   IA       : access Integer;
   Typed_IA : Integer_Access;

begin
   Typed_IA := new Integer;
   IA := Typed_IA;
   IA.all := 30;

   --  Deallocation of the access object that has
   --  an associated type:
   Free (Typed_IA);
```

(continues on next page)

(continued from previous page)

```ada

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: eff8b54adfcc8cce10920dc3620ff1b9
```

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 creates 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 652) 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:
      --  object is allocated from T_Access's pool
   end New_T;

   procedure Free (Obj : access T) is
```

(continues on next page)

(continued from previous page)

```
      Tmp : T_Access := T_Access (Obj);
   begin
      T_Access_Free (Tmp);
      --  Using deallocation procedure of the
      --  T_Access type
   end Free;

end Hidden_Anonymous_Allocation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Object_Types.Hidden_Alloc_Dealloc_Anonymous_Access_To_Object
MD5: bd3831829f34f06a1d3c25a975c850a3
```

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:

Listing 16: info.ads

```
with Hidden_Anonymous_Allocation;

package Info is

   type Rec is private;

   function New_Rec return not null access Rec;

   procedure Free (Obj : access Rec);

private

   type Rec is record
      I : Integer;
   end record;

   package Rec_Allocation is new
     Hidden_Anonymous_Allocation (T => Rec);

   function New_Rec return not null access Rec
     renames Rec_Allocation.New_T;

   procedure Free (Obj : access Rec)
     renames Rec_Allocation.Free;

end Info;
```

Listing 17: show_info_allocation_deallocation.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Object_Types.Hidden_Alloc_Dealloc_Anonymous_Access_To_Object
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

```
procedure Show_Automatic_Deallocation is
   I  : aliased Integer;
   --    ^ Allocating object on the stack

   IA : access Integer;
begin
   IA := I'Access;
   --  Indirect allocation:
   --  object creation on the stack.

   IA.all := 30;

   --  Automatic deallocation at the end of the
   --  procedure because the integer variable is
   --  on the stack.
end Show_Automatic_Deallocation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Object_Types.Deallocation_Anonymous_Access_To_Object_2
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 712). 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 738). 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.

16.3 Access discriminants

Previously, we've discussed *discriminants as access values* (page 603). 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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Simple_Example
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

```
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

```
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));
```

(continues on next page)

(continued from previous page)

```
   end Show;

end Persons;
```

Listing 24: show_person.adb

```
with Persons; use Persons;

procedure Show_Person is
   S : aliased String := "John";
   P : Person (S'Access);
begin
   P.Age := 30;
   Show (P);
end Show_Person;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Persons
MD5: f0149d572e0ec192476836bfdf00dd9e
```

Runtime output

```
Name = John
Age  =  30
```

In this example, for the discriminant of the `Person` type, we use an anonymous access type whose designated subtype is unconstrained. In the Show_Person procedure, we declare the P object and provide access to the S string.

In the Ada Reference Manual

- 3.7 Discriminants[296]
- 3.10.2 Operations of Access Types[297]

16.3.1 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:

Listing 25: custom_recs.ads

```
package Custom_Recs is

   --  Declaring a discriminant with an anonymous
   --  access type and a default value:
   type Rec (IA : access Integer :=
                  new Integer'(0)) is
   record
      I : Integer := IA.all;
   end record;

end Custom_Recs;
```

Code block metadata

296 http://www.ada-auth.org/standards/22rm/html/RM-3-7.html
297 http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Default_Expression_Non_Limited_Type
MD5: c3ddf1cdfdaefa873ad66b9e47e03058
```

Build output

```
custom_recs.ads:6:21: warning: coextension will not be deallocated when its␣
↪associated owner is deallocated [enabled by default]
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;
```

Listing 27: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Default_Expression_Limited_Type
MD5: ae872f1dec64b8e955f04789ca4db218
```

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 28: show_access_discriminants.adb

```
with Custom_Recs; use Custom_Recs;

procedure Show_Access_Discriminants is
   R : Rec;
   --  ^^^
   --  This triggers "new Integer'(0)", so an
   --  integer object is allocated and stored in
   --  the R.IA discriminant.
begin
   Show (R);

   --  R gets out of scope here, and the object
   --  allocated via new hasn't been deallocated.
end Show_Access_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Default_Expression_Limited_Type
MD5: f5d9dee26044ccab2193ab419638de79
```

Build output

```
custom_recs.ads:6:21: warning: coextension will not be deallocated when its␣
↪associated owner is deallocated [enabled by default]
```

Runtime output

```
R.IA =  0
R.I  =  0
```

In this case, the allocated object won't be deallocated when R gets out of scope!

16.3.2 Benefits of Access Discriminants

Access discriminants have the same benefits that we've already seen earlier while discussing *discriminants as access values* (page 603). 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 29: 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 30: 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);
      Put_Line ("Age  = "
                & Integer'Image (P.Age));
   end Show;

end Persons;
```

Listing 31: show_person.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Persons
MD5: e918db3790c7ffeeb7c0f54ced9f48b9
```

Build output

```
show_person.adb:5:16: error: non-local pointer cannot point to local object
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 32: persons.ads

```
package Persons is

   --  Declaring a discriminant with an anonymous
   --  access type:
   type Person (Name : access String) is record
```

(continues on next page)

(continued from previous page)

```
      Age : Integer;
   end record;

   procedure Show (P : Person);

end Persons;
```

Listing 33: show_person.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Persons
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.

16.3.3 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:

Listing 34: show_person.adb

```
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;
```

(continues on next page)

(continued from previous page)

```

   P : Person := Local_Init;
begin
   Show (P);
end Show_Person;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↪Discriminants.Persons
MD5: 9c8d2aebf60b8bb19e455cb6bc5730eb
```

Build output

```
show_person.adb:8:07: error: access discriminant in return object would be a␣
↪dangling reference
gprbuild: *** compilation phase failed
```

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.

16.4 Self-reference

Previously, we've seen that we can declare *self-references* (page 620) using named access types. We can do the same with anonymous access types. Let's revisit the code example that implements linked lists:

Listing 35: linked_lists.ads

```
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

   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;
```

(continues on next page)

(continued from previous page)

```ada

   type List is access all Component;

end Linked_Lists;
```

Listing 36: linked_lists.adb

```ada
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"
            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;
```

(continues on next page)

(continued from previous page)

```
        Put_Line ("]");
     end if;
  end Show;

end Linked_Lists;
```

Listing 37: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Self_
↪Reference.Linked_List_Example
MD5: 98b9b2ce6fac3064326e6345520dc650
```

Runtime output

```
[ 1  2  3  4  5  6  7 ]
```

Here, in the declaration of the Component type (in the private part of the generic Linked_Lists package), we declare Next as an anonymous access type that refers to the Component type. (Note that at this point, we haven't finished the declaration of the Component type yet, but we're already using it as the designated subtype of an anonymous access type.) Then, we declare List as a general access type (with Component as the designated subtype).

It's worth mentioning that the List type and the anonymous **access** Component type aren't the same type, although they share the same designated subtype. Therefore, in the implementation of the Append_Rear procedure, we have to use type conversion to convert from the anonymous **access** Component type to the (named) List type.

16.5 Mutually dependent types using anonymous access types

In the section on *mutually dependent types using access types* (page 623), we've seen a code example that was using named access types. We could now rewrite it using anonymous access types:

Listing 38: mutually_dependent.ads

```
package Mutually_Dependent is

   type T2;

   type T1 is record
      B : access T2;
   end record;

   type T2 is record
      A : access T1;
   end record;

end Mutually_Dependent;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Mutually_
↪Dependent_Anonymous_Access_Types.Example
MD5: 09f869d99b9c16882554588bb806a113
```

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.

16.6 Access parameters

In the previous chapter, we talked about *parameters as access values* (page 610). 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):

Listing 39: names.ads

```
package Names is

   function Init (S1, S2 : String)
                  return access String;
   --             ^^^^^^^^^^^^^^^^^^^^
   --  Anonymous access type as the access
   --  result type.

   procedure Show (N : access constant String);
   --                  ^^^^^^^^^^^^^^^^^^^^^^
   --  Anonymous access type as a parameter type.

end Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.Names
MD5: 622a76c4b133ed2715f18c175694cbe2
```

In this example, we have a string as the access result type of the `Init` function, and another string as the access parameter of the Show procedure.

This is the complete code example:

Listing 40: names.ads

```
package Names is

   function Init (S1, S2 : String)
                  return access String;

   procedure Show (N : access constant String);

private

   function Init (S1, S2 : String)
                  return access String is
     (new String'(S1 & "-" & S2));

end Names;
```

Listing 41: names.adb

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

package body Names is

   procedure Show (N : access constant String) is
   begin
      Put_Line ("Name: " & N.all);
   end Show;

end Names;
```

Listing 42: show_names.adb

```
with Names; use Names;

procedure Show_Names is
   N : access String := Init ("Lily", "Ann");
begin
   Show (N);
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.Names
MD5: 9fe629f29de2898f2b82d9146b22fd1a
```

Runtime output

```
Name: Lily-Ann
```

Note that we're not using the `in` parameter mode in the Show procedure above. Usually, this parameter mode can be omitted, as it is the default parameter mode — `procedure P (I : Integer)` is the same as `procedure P (I : in Integer)`. However, in the case of the Show procedure, the `in` parameter mode isn't just optionally absent. In fact, for access parameters, the parameter mode is always implied as `in`, so writing it explicitly is actually forbidden. In other words, we can only write `N : access String` or `N : access constant String`, but we cannot write `N : in access String` or `N : in access constant String`.

For further reading...

When we discussed *parameters as access values* (page 610) 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 43: 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 44: 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 45: show_names.adb

```ada
with Names; use Names;

procedure Show_Names is
   N : String := Init ("Lily", "Ann");
begin
   Show (N);
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Anonymous_Access_Parameters.Names_String
MD5: 643f193999ef8de9bcefb11d9bdd21d7
```

Runtime output

```
Name: Lily-Ann
```

Although we're using simple strings instead of access types in this version of the code example, we're still getting a similar behavior. However, there is a small, yet important difference in the way the string returned by `Init` is being allocated: while the previous implementation (which was using an access result type) was allocating the string on the heap, we're now allocating the string on the stack.

Later on, we talk about the *accessibility rules in the case of access parameters* (page 757).

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 738). Another time when access parameters are vital is for inherited primitive operations for tagged types. We discuss this *later on* (page 741).

In the Ada Reference Manual

- 3.10 Access Types[298]

16.6.1 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 46: operations_c.h

```
void add_one(int *p_i);
```

Listing 47: operations_c.c

```
void add_one(int *p_i)
{
    *p_i = *p_i + 1;
}
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
MD5: 3270f3b2415266a203a6f4c605c3831b
```

We could map the **int** * parameter of add_one to **access Integer** in the Ada specification:

```
procedure Add_One (IA : access Integer)
  with Import, Convention => C;
```

This is a complete code example:

Listing 48: operations.ads

```
package Operations is

   procedure Add_One (IA : access Integer)
     with Import, Convention => C;

end Operations;
```

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

Listing 49: show_operations.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
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 50: operations.ads

```
package Operations is

   procedure Add_One
    (IA : aliased in out Integer)
      with Import, Convention => C;

end Operations;
```

Listing 51: show_operations.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
MD5: 2c5a81b8d77f0fff8a73f7912be6b6fe
```

However, there are situations where aliased objects cannot be used. For example, suppose we want to allocate memory inside a C function. In this case, the pointer to that memory block must be mapped to an access type in Ada.

Let's extend the previous C code example and introduce the `alloc_integer` and `dealloc_integer` functions, which allocate and deallocate an integer value:

Listing 52: operations_c.h

```
int * alloc_integer();

void dealloc_integer(int *p_i);

void add_one(int *p_i);
```

Listing 53: operations_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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
MD5: ec6dea12d0a948489cce21b0cc0a1ad2
```

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 54: operations.ads

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
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:

Listing 55: show_operations.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
MD5: b2b96a166926528bc44059b56e31fb55
```

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[299]

16.6.2 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:

Listing 56: inherited_primitives.ads

```
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;

```

(continues on next page)

[299] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

(continued from previous page)

```
private

   type T is tagged null record;

   type T_Child is new T with null record;

end Inherited_Primitives;
```

Listing 57: 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 58: 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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.Inherited_Primitives
MD5: 8235b21caa9f1f105f533d74d891adfe
```

Build output

```
show_inherited_primitives.adb:9:10: error: expected type "T_Access" defined at␣
↪inherited_primitives.ads:5
show_inherited_primitives.adb:9:10: error: found type "T_Child_Access" defined at␣
↪inherited_primitives.ads:12
gprbuild: *** compilation phase failed
```

In this example, Proc is not a primitive of type T because it's referring to type T_Access, not type T. This means that Proc isn't inherited when we derive the T_Child type. Therefore, when we call `Proc (Obj_Child)`, a compilation error occurs because the compiler expects type T_Access — there's no `Proc (N : T_Child_Access)` that could be used here.

If we replace T_Access in the Proc procedure with an an access parameter (`access T`), the subprogram becomes a primitive of T:

Listing 59: inherited_primitives.ads

```
package Inherited_Primitives is

   type T is tagged private;
```

(continues on next page)

(continued from previous page)

```ada

   procedure Proc (N : access T);
   --  Proc is a primitive of type T.

   type T_Child is new T with private;

private

   type T is tagged null record;

   type T_Child is new T with null record;

end Inherited_Primitives;
```

Listing 60: inherited_primitives.adb

```ada
package body Inherited_Primitives is

   procedure Proc (N : access T) is null;

end Inherited_Primitives;
```

Listing 61: show_inherited_primitives.adb

```ada
with Inherited_Primitives;
use  Inherited_Primitives;

procedure Show_Inherited_Primitives is
   Obj       : access T       := new T;
   Obj_Child : access T_Child := new T_Child;
begin
   Proc (Obj);
   Proc (Obj_Child);
   --    ^^^^^^^^^
   --    OK: Proc is inherited!
end Show_Inherited_Primitives;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.Inherited_Primitives
MD5: a7e9b8bc92e346758cc4ade43bb4b02d
```

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[300]

300 http://www.ada-auth.org/standards/22rm/html/RM-3-9-2.html

16.7 User-Defined References

Implicit dereferencing (page 625) 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 725); and
- indicate the name of the reference discriminant when specifying the `Implicit_Dereference` aspect.

Let's see a simple example:

Listing 62: show_user_defined_reference.adb

```
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.Ref.Id
   --                          or:
   --                           R.Ref.all.Id
end Show_User_Defined_Reference;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.Simple_User_Defined_References
MD5: 33eaa7e8e75b4eb56d64dcc17e2932aa
```

Runtime output

```
ID:  42
```

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:

```
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:

```
--  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[301]

[301] http://www.ada-auth.org/standards/22rm/html/RM-4-1-5.html

16.7.1 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 63: info.ads

```
package Info is
   type Id_Number (Id : Positive) is
     tagged private;

   procedure Show (R : Id_Number);
private
   type Id_Number (Id : Positive) is
     tagged null record;
end Info;
```

Listing 64: info.adb

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

package body Info is

   procedure Show (R : Id_Number) is
   begin
      Put_Line ("ID: " & Positive'Image (R.Id));
   end Show;

end Info;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.Dereferencing_Tagged_Types
MD5: 4de65094963450dc3a7505dbf93c2551
```

Then, let's declare a reference type and a reference object in the test application:

Listing 65: show_user_defined_reference.adb

```
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)

(continued from previous page)

```
end Show_User_Defined_Reference;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.Dereferencing_Tagged_Types
MD5: 9c5dfc4f2b8e085efde9e61689243f70
```

Runtime output

```
ID:  42
```

Here, we can call the Show procedure by simply writing `R.Show` instead of `R.Ref.all.Show`.

16.7.2 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 66: 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 67: 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;

end National_Date_Info;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.National_Dates
MD5: 90fd6740d701025e1d5f30c9751a528d
```

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 68: 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 69: 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
        National_Date_Reference'(Ref => null);
      --  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
      --   Returning reference object with a null
      --   reference in case the country wasn't
```

(continues on next page)

(continued from previous page)

```
      --   found. This will trigger an exception
      --   if we try to dereference it.
   end Find;

end National_Date_Containers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.National_Dates
MD5: ec37ae93a7052c4bc731b2a7be0763ab
```

Package `National_Date_Containers` contains the `National_Dates` type, which is an array type for declaring containers that we use to store the national day information. We can also see the declaration of the `National_Date_Reference` type, which is the reference type returned by the `Find` function when looking for a specific country in the container.

> **Important**
>
> We're declaring the container type (`National_Dates`) as an array type just to simplify the code. In many cases, however, this approach isn't recommended! Instead, we should use a private type in order to encapsulate — and better protect — the information stored in the actual container.

Finally, let's see a test application that stores information for some countries into the `Nat_Dates` container and displays the information for a specific country:

Listing 70: show_national_dates.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.National_Dates
MD5: 771ecb91e8f890d4bb9b08115ae833f4
```

Runtime output

```
Country: FRA
Year:     1789
```

Here, we call the Find function to retrieve a reference object, whose reference (access value) has the national day information of France. We then implicitly dereference it to get the tagged object (of National_Date type) and display its information by calling the Show procedure.

Relevant topics

The National_Date_Containers package was implemented specifically as an accompanying package for the National_Date_Info package. It is possible, however, to generalize it, so that we can reuse the container for other record types. In fact, this is actually very straightforward:

Listing 71: generic_containers.ads

```ada
generic
   type T is private;
   type T_Access is access T;
   type T_Cmp is private;
   with function Matches (E    : T_Access;
                          Elem : T_Cmp)
                          return Boolean;
package Generic_Containers is

   type Ref_Type (Ref : access T) is
     tagged limited null record
       with Implicit_Dereference => Ref;

   type Container is
     array (Positive range <>) of
       T_Access;

   function Find (Cont : Container;
                  Elem : T_Cmp)
                  return Ref_Type;

end Generic_Containers;
```

Listing 72: generic_containers.adb

```ada
package body Generic_Containers is

   function Find (Cont : Container;
                  Elem : T_Cmp)
                  return Ref_Type is
   begin
      for I in Cont'Range loop
         if Matches (Cont (I), Elem) then
            return Ref_Type'(Ref => Cont (I));
         end if;
      end loop;

      return Ref_Type'(Ref => null);
   end Find;

end Generic_Containers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.National_Dates
MD5: 94c23a48131a47439b5b41e985c3d6c1
```

When comparing the Generic_Containers package to the National_Date_Containers package, we see that the main difference is the addition of the Matches function, which indicates whether the current element we're evaluating in the for-loop of the Find function is the one we're looking for.

In the main application, we can implement the Matches function and declare the National_Date_Containers package as an instance of the Generic_Containers package:

Listing 73: show_national_dates.adb

```
with Generic_Containers;
with National_Date_Info; use National_Date_Info;

procedure Show_National_Dates is

   function Matches_Country
     (E    : National_Date_Access;
      Elem : Country_Code)
      return Boolean is
        (E.Country = Elem);

   package National_Date_Containers is new
     Generic_Containers
       (T        => National_Date,
        T_Access => National_Date_Access,
        T_Cmp    => Country_Code,
        Matches  => Matches_Country);

   use National_Date_Containers;

   subtype National_Dates is Container;

   Nat_Dates : constant
                 National_Dates (1 .. 5) :=
     (new National_Date'("USA",
                         Time'(1776,  7,  4)),
      new National_Date'("FRA",
                         Time'(1789,  7, 14)),
      new National_Date'("DEU",
                         Time'(1990, 10,  3)),
      new National_Date'("SPA",
                         Time'(1492, 10, 12)),
      new National_Date'("BRA",
                         Time'(1822,  9,  7)));

begin
   Find (Nat_Dates, "FRA").Show;
end Show_National_Dates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_
↪Defined_References.National_Dates
MD5: f4dac1fed69b9bccce5dccbf17844adc
```

Runtime output

```
Country: FRA
Year:     1789
```

Here, we instantiate the `Generic_Containers` package with the `Matches_Country` function, which is an expression function that compares the country component of the current `National_Date` reference with the name of the country we desire to learn about.

This generalized approach is actually used for the standard containers from the `Ada.Containers` packages. For example, the `Ada.Containers.Vectors` is specified as follows:

```
with Ada.Iterator_Interfaces;

generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
                      return Boolean is <>;
package Ada.Containers.Vectors
  with Preelaborate, Remote_Types,
       Nonblocking,
       Global => in out synchronized is

   -- OMITTED

   type Reference_Type
     (Element : not null access Element_Type) is
       private
         with Implicit_Dereference => Element,
              Nonblocking,
              Global => in out synchronized,
              Default_Initial_Condition =>
                (raise Program_Error);

   -- 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)
      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[302]

16.8 Anonymous Access Types and Accessibility Rules

In general, the *accessibility rules* (page 647) 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 647) to make use of anonymous access types:

Listing 74: library_level.ads

```
package Library_Level is

   L0_AO  : access Integer;

   L0_Var : aliased Integer;

end Library_Level;
```

Listing 75: show_library_level.adb

```
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;
```

(continues on next page)

[302] http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html

(continued from previous page)

```ada
   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.

   Test;
end Show_Library_Level;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Accessibility_Levels_Rules_Introduction.Accessibility_Library_Level
MD5: 255bdecebdaa735408db082edd583a0c
```

Build output

```
show_library_level.adb:13:16: error: non-local pointer cannot point to local object
show_library_level.adb:30:13: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed
```

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 `L0_AO := L1_Var'Access` is illegal because we're trying to assign to an access object of less deep level.

However, assignment such as `L0_AO := L1_AO` are possible now: we don't get a type mismatch — as we did with named access types — because both objects are of anonymous access types. Note that the accessibility level cannot be determined at compile time: `L1_AO` can hold an access value at library level (which would make the assignment legal) or at a deeper level. Therefore, the compiler introduces an accessibility check here.

However, the accessibility check used in `L0_AO := L1_AO` fails at runtime because the corresponding access value (`L1_Var'Access`) is of a deeper level than `L0_AO`, which is illegal. (If you comment out the `L1_AO := L1_Var'Access` assignment prior to the `L0_AO := L1_AO` assignment, this accessibility check doesn't fail anymore.)

16.8.1 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 647) and add anonymous access types to it:

Listing 76: 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 77: 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);

```

(continues on next page)

(continued from previous page)

```ada
   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
   --                (anonymous type)
   --              to
   --              L0 access object (named type)

   L0_IA := L0_Integer_Access (L1_AO);
   --       ^^^^^^^^^^^^^^^^^
   --       ILLEGAL: conversion / assignment from
   --                L1 access object
   --                  (anonymous type)
   --                to
   --                L0 access object (named type)
   --                (accessibility check fails)

   L1_IA := L1_Integer_Access (L0_AO);
   --       ^^^^^^^^^^^^^^^^^
   --       LEGAL: conversion / assignment from
   --              L0 access object
   --                (anonymous type)
```

(continues on next page)

(continued from previous page)

```
   --              to
   --              L1 access object (named type)

   L1_IA := L1_Integer_Access (L1_AO);
   --       ^^^^^^^^^^^^^^^^^
   --       LEGAL: conversion / assignment from
   --              L1 access object
   --                (anonymous type)
   --              to
   --              L1 access object (named type)
end Show_Library_Level;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Accessibility_Levels_Rules_Introduction.Accessibility_Named_Anonymous_Access_
↪Type_Conversions
MD5: a2e73bb0ed543bc4973850c80f951039
```

Build output

```
show_library_level.adb:28:13: error: cannot convert local pointer to non-local␣
↪access type
gprbuild: *** compilation phase failed
```

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 `L0_AO := L1_IA` and the `L0_IA := L0_Integer_Access (L1_AO)` assignments. Otherwise, mixing those access objects doesn't impose additional hurdles.

16.8.2 Accessibility rules on access parameters

In the previous chapter, we saw that the accessibility rules also apply to *access values as subprogram parameters* (page 650). 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 78: names.ads

```
package Names is

   procedure Show (N : access constant String);

end Names;
```

Listing 79: 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 : access constant String) is
   begin
```

(continues on next page)

(continued from previous page)

```
      --  for I in N'Range loop
      --     N (I) := To_Lower (N (I));
      --  end loop;
      Put_Line ("Name: " & N.all);
   end Show;

end Names;
```

Listing 80: show_names.adb

```
with Names; use Names;

procedure Show_Names is
   S : aliased String := "John";
begin
   Show (S'Access);
end Show_Names;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
↪Levels_Rules_Introduction.Accessibility_Checks_Parameters
MD5: aa930ba9be3264d01eb9115d27b884eb
```

Runtime output

```
Name: John
```

As we've seen in the previous chapter, compilation fails when we use named access types in this code example. In the case of access parameters, using S'Access doesn't make the compilation fail, nor does the accessibility check fail at runtime because S is still in scope when we call the Show procedure.

16.9 Anonymous Access-To-Subprograms

In the previous chapter, we talked about *named access-to-subprogram types* (page 677). Now, we'll see that the anonymous version of those types isn't much different from the named version.

Let's start our discussion by declaring a subprogram parameter using an anonymous access-to-procedure type:

Listing 81: anonymous_access_to_subprogram.ads

```
package Anonymous_Access_To_Subprogram is

   procedure Proc
     (P : access procedure (I : in out Integer));

end Anonymous_Access_To_Subprogram;
```

Listing 82: anonymous_access_to_subprogram.adb

```
package body Anonymous_Access_To_Subprogram is

   procedure Proc
     (P : access procedure (I : in out Integer))
   is
```

(continues on next page)

(continued from previous page)

```
      I : Integer := 0;
   begin
      P (I);
   end Proc;

end Anonymous_Access_To_Subprogram;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
MD5: 2cbe76d7e23905d575bd27e29d5e3175
```

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 83: add_ten.ads

```
procedure Add_Ten (I : in out Integer);
```

Listing 84: add_ten.adb

```
procedure Add_Ten (I : in out Integer) is
begin
   I := I + 10;
end Add_Ten;
```

Listing 85: add_twenty.ads

```
procedure Add_Twenty (I : in out Integer);
```

Listing 86: add_twenty.adb

```
procedure Add_Twenty (I : in out Integer) is
begin
   I := I + 20;
end Add_Twenty;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
MD5: 50eaeaf27caaa9618b35ecdf8acc11fe
```

Finally, this is our test application:

Listing 87: show_anonymous_access_to_subprograms.adb

```
with Anonymous_Access_To_Subprogram;
use  Anonymous_Access_To_Subprogram;

with Add_Ten;

procedure Show_Anonymous_Access_To_Subprograms is
begin
```

(continues on next page)

(continued from previous page)

```
   Proc (Add_Ten'Access);
   --            ^ Getting access to Add_Ten
   --              procedure and passing it
   --              to Proc
end Show_Anonymous_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
MD5: 13143ccf9620d26031484ba160a58fe1
```

Here, we get access to the Add_Ten procedure and pass it to the Proc procedure. Note that this implementation is not different from the *example for named access-to-subprogram types* (page 679). 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 767).

In the Ada Reference Manual

- 3.10 Access Types[303]

16.9.1 Examples of anonymous access-to-subprogram usage

In the section about *named access-to-subprogram types* (page 677), we've seen a couple of different usages for those types. In all those examples we discussed, we could instead have used anonymous access-to-subprogram types. Let's see a code example that illustrates that:

Listing 88: all_anonymous_access_to_subprogram.ads

```
package All_Anonymous_Access_To_Subprogram is

   --
   --  Anonymous access-to-subprogram as
   --  subprogram parameter:
   --
   procedure Proc
     (P : access procedure (I : in out Integer));

   --
   --  Anonymous access-to-subprogram in
   --  array type declaration:
   --
   type Access_To_Procedure_Array is
     array (Positive range <>) of
       access procedure (I : in out Integer);

   protected type Protected_Integer is

     procedure Mult_Ten;

     procedure Mult_Twenty;

   private
     I : Integer := 1;
```

(continues on next page)

[303] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

(continued from previous page)

```
   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 89: all_anonymous_access_to_subprogram.adb

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

package body All_Anonymous_Access_To_Subprogram is

   procedure Proc
     (P : access procedure (I : in out Integer))
   is
      I : Integer := 0;
   begin
      Put_Line
        ("Calling procedure for Proc...");
      P (I);
      Put_Line ("Finished.");
   end Proc;

   procedure Process
     (R : in out
            Rec_Access_To_Procedure_Discriminant)
   is
   begin
      Put_Line
```

(continues on next page)

(continued from previous page)

```
        ("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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
MD5: 628dcfdc5fe9b712f33fa044057093c2
```

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 90: show_anonymous_access_to_subprograms.adb

```
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
```

(continues on next page)

(continued from previous page)

```
   --
   --  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:
   --
   procedure Process_Integer is new
     Gen_Process (T      => Integer,
                  Proc_T => Add_Twenty'Access);

   --
   --  Object (APP) of anonymous
   --  access-to-protected-subprogram:
   --
   PI  : Protected_Integer;
   APP : constant access protected procedure :=
           PI.Mult_Ten'Access;

   Some_Int : Integer := 0;
begin
   Put_Line ("Some_Int: " & Some_Int'Image);

   --
```

(continues on next page)

(continued from previous page)

```
   --  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:
   --
   Process (RD);
   Put_Line ("RD.I: " & RD.I'Image);

   --
   --  Using procedure instantiated with
   --  formal anonymous access-to-subprogram:
   --
   Process_Integer (Some_Int);
   Put_Line ("Some_Int: " & Some_Int'Image);

   --
```

(continues on next page)

(continued from previous page)

```
   --  Using object of anonymous
   --  access-to-protected-subprogram type:
   --
   APP.all;
end Show_Anonymous_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
MD5: ec770c17e880a98fd2e9ab0110d4a858
```

Runtime output

```
Some_Int:  0
Calling procedure for Proc...
Finished.
Calling procedure for Proc...
Finished.
Some_Int:  30
Calling procedure from PA array...
Some_Int:  40
Some_Int:  60
Finished.
Calling procedure from PAA array...
Some_Int:  70
Some_Int:  90
Finished.
Some_Int:  90
Some_Int:  100
Calling procedure for Rec_Access_To_Procedure_Discriminant type...
Finished.
RD.I:  20
Calling procedure for Gen_Process...
Finished.
Some_Int:  120
```

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 677), 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.

16.9.2 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 715), which have *many drawbacks* (page 718) — 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 767). 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.)

16.9.3 Readability

Note that readability suffers if you use a *cascade* of anonymous access-to-subprograms. For example:

Listing 91: 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 92: 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;
```

Listing 93: readability_issue.adb

```ada
with Readability_Issue.Functions;
use  Readability_Issue.Functions;

package body Readability_Issue is

   function F
     return access
       function (A : Integer)
                 return access
                   function (B : Float)
                       return Integer is
     (Select_Conversion'Access);

```

(continues on next page)

(continued from previous page)

```
end Readability_Issue;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Subprograms.Readability_Issue
MD5: 9e2ac58942c97b44c0d847c28e39bd11
```

In this example, the definition of F might compile fine, but it's simply too long to be readable. Not only that: we need to carry this *chain* to other functions as well — such as the Select_Conversion function above. Also, using these functions in an application is not straightforward:

Listing 94: 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
        := F;
   F2 : access function (B : Float)
                         return Integer
        := F1 (2);
   I  : Integer := F2 (0.1);
begin
   I := F1 (2) (0.1);
end Show_Readability_Issue;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_To_Subprograms.Readability_Issue
MD5: 80267b1d673663e3cacba0c4978e6abf
```

Therefore, our recommendation is to avoid this kind of *access cascading* by carefully designing your application. In general, you won't need that.

16.10 Accessibility Rules and Anonymous Access-To-Subprograms

In principle, the *accessibility rules for anonymous access types* (page 753) 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 702) 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[304]

[304] http://www.ada-auth.org/standards/22rm/html/RM-3-10.html

16.10.1 Named vs. anonymous access-to-subprograms

Let's see an example of a named access-to-subprogram type:

Listing 95: show_access_to_subprogram_error.adb

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

procedure Show_Access_To_Subprogram_Error is

   type PI is access
     procedure (I : in out Integer);

   P : PI;

   I : Integer := 0;
begin
   declare
      procedure Add_One (I : in out Integer) is
      begin
         I := I + 1;
      end Add_One;
   begin
      P := Add_One'Access;
   end;
end Show_Access_To_Subprogram_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Accessibility_Rules_Anonymous_Access_To_Subprograms.Simple_Example_Named
MD5: 41c36426112e799210b7704dd43b6217
```

Build output

```
show_access_to_subprogram_error.adb:18:12: error: subprogram must not be deeper␣
↪than access type
gprbuild: *** compilation phase failed
```

In this example, we get a compilation error because the lifetime of the Add_One procedure is shorter than the access type PI.

In contrast, using an anonymous access-to-subprogram type eliminates the compilation error, i.e. the assignment `P := Add_One'Access` becomes legal:

Listing 96: 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.
```

(continues on next page)

(continued from previous page)

```
   end;
end Show_Access_To_Subprogram_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Accessibility_Rules_Anonymous_Access_To_Subprograms.Simple_Example_Anonymous
MD5: a5eeb4a716b4f6a932dd74c580a07b66
```

Runtime output

```
raised PROGRAM_ERROR : show_access_to_subprogram_error.adb:14 accessibility check␣
↪failed
```

In this case, the compiler introduces an accessibility check, which fails at runtime because the lifetime of Add_One is shorter than the lifetime of the access object P.

16.10.2 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 97: access_to_subprogram_types.ads

```
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 98: access_to_subprogram_types.adb

```
package body Access_To_Subprogram_Types is

   procedure Process
     (Arr : in out Integer_Array;
      P   :        Process_Procedure) is
```

(continues on next page)

(continued from previous page)

```
   begin
      P (Arr);
   end Process;

end Access_To_Subprogram_Types;
```

Listing 99: 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

   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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪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 100: 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;
```

Listing 101: access_to_subprogram_types.adb

```
package body Access_To_Subprogram_Types is

   procedure Process
     (Arr : in out Integer_Array;
      P   : access procedure
              (Arr : in out Integer_Array)) is
   begin
      P (Arr);
   end Process;

end Access_To_Subprogram_Types;
```

Listing 102: 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

   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
```

(continues on next page)

(continued from previous page)

```
      (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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Accessibility_Rules_Anonymous_Access_To_Subprograms.Access_To_Subprogram_
↪Parameter_Anonymous
MD5: a500e0a864f0adadc1d6823c1f50bd64
```

Runtime output

```
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`.

16.10.3 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[305]
- A.18.4 Maps[306]
- A.18.7 Sets[307]

[305] http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html
[306] http://www.ada-auth.org/standards/22rm/html/RM-A-18-4.html
[307] http://www.ada-auth.org/standards/22rm/html/RM-A-18-7.html

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 103: 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 104: 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
```

(continues on next page)

(continued from previous page)

```
   begin
      for I in D.S'First .. D.Curr loop
         Proc (D.S (I));
      end loop;
   end Iterate;

end Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↪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 105: float_data_processing.ads

```
with Data_Processing;

package Float_Data_Processing is
  new Data_Processing (Element => Float);
```

Listing 106: show_access_to_subprograms.adb

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

with Float_Data_Processing;
use  Float_Data_Processing;

procedure Show_Access_To_Subprograms is

   procedure Display (F : Float) is
   begin
      Put_Line ("F :" & Float'Image (F));
   end Display;

   D : Data_Container (5);
begin
    Append (D, 1.0);
    Append (D, 2.0);
    Append (D, 3.0);

    Iterate (D, Display'Access);
end Show_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↪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 107: 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);

   procedure Iterate
     (D    : Data_Container;
      Proc : not null access
               procedure (E : Element));

private

   type Data_Container_Storage is
     array (Positive range <>) of Element;

   type Data_Container (Last : Positive) is
   record
      S    : Data_Container_Storage (1 .. Last);
      Curr : Natural := 0;
   end record;

end Data_Processing;
```

Listing 108: 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
```

(continues on next page)

(continued from previous page)

```
   begin
      for I in D.S'First .. D.Curr loop
         Proc (D.S (I));
      end loop;
   end Iterate;

end Data_Processing;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Anonymous
MD5: fa56595ef1734f2f07ad719c36dfd8b5
```

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 109: float_data_processing.ads

```
with Data_Processing;

package Float_Data_Processing is
  new Data_Processing (Element => Float);
```

Listing 110: show_access_to_subprograms.adb

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

with Float_Data_Processing;
use  Float_Data_Processing;

procedure Show_Access_To_Subprograms is

   procedure Display (F : Float) is
   begin
      Put_Line ("F :" & Float'Image (F));
   end Display;

   D : Data_Container (5);
begin
    Append (D, 1.0);
    Append (D, 2.0);
    Append (D, 3.0);

    Iterate (D, Display'Access);
end Show_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↪Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Anonymous
MD5: 64ee435aac5f2817b7d9cecf538a1e4c
```

Runtime output

```
F : 1.00000E+00
F : 2.00000E+00
F : 3.00000E+00
```

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`.

CHAPTER

SEVENTEEN

LIMITED TYPES

So far, we discussed nonlimited types in most cases. In this chapter, we discuss limited types.

We can think of limited types as an easy way to avoid inappropriate semantics. For example, a lock should not be copied — neither directly, via assignment, nor with pass-by-copy. Similarly, a *file*, which is really a file descriptor, should not be copied. In this chapter, we'll see example of unwanted side-effects that arise if we don't use limited types for these cases.

17.1 Assignment and equality

Limited types have the following restrictions, which we discussed in the Introduction to Ada[308] 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

```
package Nonlimited_Types is

   type Simple_Rec is private;

   type Integer_Access is access Integer;

   function Init (I : Integer) return Simple_Rec;

   procedure Set (E : Simple_Rec;
                  I : Integer);

   procedure Show (E      : Simple_Rec;
                   E_Name : String);

private

   type Simple_Rec is record
      V : Integer_Access;
```

(continues on next page)

308 https://learn.adacore.com/courses/intro-to-ada/chapters/privacy.html#intro-ada-limited-types

(continued from previous page)

```
   end record;

end Nonlimited_Types;
```

Listing 2: nonlimited_types.adb

```
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;
```

Listing 3: show_wrong_assignment_equality.adb

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

procedure Show_Wrong_Assignment_Equality is
   A, B : Simple_Rec := Init (0);

   procedure Show_Compare is
   begin
      if A = B then
         Put_Line ("A = B");
      else
         Put_Line ("A /= B");
      end if;
   end Show_Compare;
begin

   Put_Line ("A := Init (0); A := Init (0);");
   Show (A, "A");
   Show (B, "B");
   Show_Compare;
   Put_Line ("--------");

   Put_Line ("Set (A, 2); Set (B, 3);");
   Set (A, 2);
```

(continues on next page)

(continued from previous page)

```
   Set (B, 3);

   Show (A, "A");
   Show (B, "B");
   Put_Line ("--------");

   Put_Line ("B := A");
   B := A;

   Show (A, "A");
   Show (B, "B");
   Show_Compare;
   Put_Line ("--------");

   Put_Line ("Set (B, 7);");
   Set (B, 7);

   Show (A, "A");
   Show (B, "B");
   Show_Compare;
   Put_Line ("--------");

end Show_Wrong_Assignment_Equality;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
 ↪Equality.Wrong_Assignment_Equality
MD5: 72cf7145cd26a8628580c5a837d9cb61
```

Runtime output

```
A := Init (0); A := Init (0);
A.V.all =  0
B.V.all =  0
A /= B
--------
Set (A, 2); Set (B, 3);
A.V.all =  2
B.V.all =  3
--------
B := A
A.V.all =  2
B.V.all =  2
A = B
--------
Set (B, 7);
A.V.all =  7
B.V.all =  7
A = B
--------
```

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[309]

17.1.1 Assignments

Assignments are forbidden when using objects of limited types. For example:

Listing 4: limited_types.ads

```
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;
```

Listing 5: limited_types.adb

```
package body Limited_Types is

   function Init (I : Integer) return Simple_Rec
   is
   begin
```

(continues on next page)

[309] http://www.ada-auth.org/standards/22rm/html/RM-7-5.html

(continued from previous page)

```
      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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
↪Equality.Assignment
MD5: 019c16f7feac896fd8c37d40d0522dc8
```

Build output

```
show_limited_assignment.adb:6:04: error: left hand of assignment must not be␣
↪limited type
gprbuild: *** compilation phase failed
```

In this example, we declare the limited private type Simple_Rec and two objects of this type (A and B) in the Show_Limited_Assignment procedure. (We discuss more about limited private types *later* (page 787)).

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;
```

Listing 8: 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;

   procedure Copy (From :        Simple_Rec;
                   To   : in out Simple_Rec)
   is
   begin
      --  Copying record components
      To.V.all := From.V.all;
   end Copy;

end Limited_Types;
```

Listing 9: show_limited_assignment.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
↪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.

17.1.2 Equality

Limited types don't have a predefined equality operator. For example:

Listing 10: 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;

private
```

(continues on next page)

(continued from previous page)

```

   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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
↪Equality.Equality
MD5: dad31b5e36de0b3b7824f723a60e5aa0
```

Build output

```
show_limited_equality.adb:8:09: error: there is no applicable operator "=" for␣
↪private type "Simple_Rec" defined at limited_types.ads:5
gprbuild: *** compilation phase failed
```

As expected, the comparison `A = B` triggers a compilation error because no predefined `=` operator is available for the `Simple_Rec` type. If we want to be able to compare objects of this type, we have to implement the `=` operator ourselves. For example, we can do that for the `Simple_Rec` type:

Listing 13: limited_types.ads

```
package Limited_Types is

   type Integer_Access is access Integer;

```

(continues on next page)

(continued from previous page)

```
   type Simple_Rec is limited private;

   function Init (I : Integer) return Simple_Rec;

   function "=" (Left, Right : Simple_Rec)
                 return Boolean;

private

   type Simple_Rec is limited record
      V : Integer_Access;
   end record;

end Limited_Types;
```

Listing 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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
↪Equality.Equality
MD5: f56b2229443a5e4e33c402b41b02d318
```

Runtime output

```
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 replacements when required.

In other languages

In C++, you can overload the assignment operator. For example:

```
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:

```
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;
```

17.2 Limited private types

As we've seen in code examples from the previous section, we can apply *information hiding* (page 43) to limited types. In other words, we can declare a type as **limited private** instead of just **limited**. For example:

Listing 16: simple_recs.ads

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Private
MD5: ececb364f5365a74db43952e9421dee0
```

In this case, in addition to the fact that assignments are forbidden for objects of this type (because Rec is limited), we cannot access the record components.

Note that in this example, both partial and full views of the Rec record are of limited type. In the next sections, we discuss how the partial and full views can have non-matching declarations.

In the Ada Reference Manual

- 7.5 Limited Types[310]

17.2.1 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Non_Record_Limited_Error
MD5: c155e02d809caf28352cbbb579deb861
```

However, we've mentioned *in a previous chapter* (page 45) 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)

[310] http://www.ada-auth.org/standards/22rm/html/RM-7-5.html

(continued from previous page)

```
      limited private;

   type Limited_Integer_Array_2 is
     limited private;

private

   type Limited_Enumeration is (Off, On);

   type Limited_Integer is new Integer;

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

   type Limited_Integer_Array_2 is
     new Integer_Array (1 .. 2);

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Non_Record_Limited
MD5: 9e65b56a5cb3d7a3da11c7f63ee9bb19
```

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.

17.2.2 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:

Listing 19: simple_recs.ads

```
package Simple_Recs is

   type Rec is limited private;
   --  Partial view of Rec is limited

private

   type Rec is record
   --  Full view of Rec is nonlimited
      I : Integer;
   end record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Partial_Full_View
MD5: 5d0dbc3e87531476856f0ac1f9b22c78
```

In this case, only the partial view of Rec is limited, while its full view is nonlimited. When deriving from Rec, the view of the derived type is the same as for the parent type:

Listing 20: simple_recs-child.ads

```
package Simple_Recs.Child
is
   type Rec_Derived is new Rec;
   --  As for its parent, the
   --  partial view of Rec_Derived
   --  is limited, but the full view
   --  is nonlimited.

end Simple_Recs.Child;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Partial_Full_View
MD5: fdf0ffa87ac2b8766830bf8e17ac7b5e
```

Clients must nevertheless comply with their partial view, and treat the type as if it is in fact limited. In other words, if you use the Rec type in a subprogram or package outside of the Simple_Recs package (or its child packages), the type is limited from that perspective:

Listing 21: use_rec_in_subprogram.adb

```
with Simple_Recs; use Simple_Recs;

procedure Use_Rec_In_Subprogram is
   R1, R2 : Rec;
begin
   R1.I := 1;
   R2   := R1;
end Use_Rec_In_Subprogram;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Partial_Full_View
MD5: f0af323a951853b97a2b67ce9b13e732
```

Build output

```
use_rec_in_subprogram.adb:6:04: error: invalid prefix in selected component "R1"
use_rec_in_subprogram.adb:7:04: error: left hand of assignment must not be limited␣
↪type
gprbuild: *** compilation phase failed
```

Here, compilation fails because the type Rec is limited from the procedure's perspective.

Limitations

Note that the opposite — declaring a type as **private** and its full full view as **limited private** — is not possible. For example:

Listing 22: simple_recs.ads

```
package Simple_Recs is

   type Rec is private;

private

```

(continues on next page)

(continued from previous page)

```
   type Rec is limited record
      I : Integer;
   end record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Partial_Full_View
MD5: ec1c8a2dcf3cc2c49b1497cf4c9d3a5a
```

Build output

```
use_rec_in_subprogram.adb:6:04: error: invalid prefix in selected component "R1"
simple_recs.ads:7:09: error: completion of nonlimited type cannot be limited
gprbuild: *** compilation phase failed
```

As expected, we get a compilation error in this case. The issue is that the partial view cannot be allowed to mislead the client about what's possible. In this case, if the partial view allows assignment, then the full view must actually provide assignment. But the partial view can restrict what is actually possible, so a limited partial view need not be completed in the full view as a limited type.

In addition, tagged limited private types cannot have a nonlimited full view. For example:

Listing 23: simple_recs.ads

```
package Simple_Recs is

   type Rec is tagged limited private;

private

   type Rec is tagged record
      I : Integer;
   end record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Partial_Full_View
MD5: cadb9ca1346a98fb65f9059fdb29f865
```

Build output

```
simple_recs-child.ads:3:28: error: type derived from tagged type must have␣
↪extension
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.

17.2.3 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

```
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

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Non_Limited_Partial_Full_View
MD5: 24b75bb97ddd485bd6825bb8647607c1
```

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.

17.2.4 Limited private component

Another example mentioned by the Ada Reference Manual (7.3.1[311], 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:

Listing 26: limited_nonlimited_arrays.ads

```ada
package Limited_Nonlimited_Arrays is

   type Limited_Private is
     limited private;

   function Init return Limited_Private;

   --  The array type Limited_Private_Array
   --  is limited because the type of its
   --  component is limited.
   type Limited_Private_Array is
     array (Positive range <>) of
       Limited_Private;

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;
```

Listing 27: limited_nonlimited_arrays.adb

```ada
package body Limited_Nonlimited_Arrays is

   function Init return Limited_Private is
     ((A => 1));

end Limited_Nonlimited_Arrays;
```

311 http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html

Listing 28: show_limited_nonlimited_array.adb

```
with Limited_Nonlimited_Arrays;
use  Limited_Nonlimited_Arrays;

procedure Show_Limited_Nonlimited_Array is
   A3 : Limited_Private_Array (1 .. 2) :=
          (others => Init);
   A4 : Limited_Private_Array (1 .. 2);
begin
   --  ERROR: this assignment is illegal because
   --  Limited_Private_Array is limited, as
   --  its component is limited at this point.
   A4 := A3;
end Show_Limited_Nonlimited_Array;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Limited_Nonlimited_Array
MD5: 211670e99e6e3a63a785bb2dde255b58
```

Build output

```
show_limited_nonlimited_array.adb:12:04: error: left hand of assignment must not␣
↪be limited type
show_limited_nonlimited_array.adb:12:04: error: component type "Limited_Private"␣
↪of subtype of "Limited_Private_Array" is limited
gprbuild: *** compilation phase failed
```

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[312]

17.2.5 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

```
package Simple_Recs is

   type Rec is tagged limited private;

private

   type Rec is tagged limited record
      I : Integer;
   end record;
```

(continues on next page)

[312] http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html

(continued from previous page)

```

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↪Types.Tagged_Limited_Private_Types
MD5: bee48bd7e0d70ddfd288c0de5e21b039
```

Here, the tagged Rec type is limited both in its partial and full views. Any mismatch in one of the views triggers a compilation error. (As an exercise, you may remove any of the **limited** keywords from the code example and try to compile it.)

> **For further reading...**
>
> This rule is for the sake of dynamic dispatching and classwide types. The compiler must not allow any of the types in a derivation class — the set of types related by inheritance — to be different regarding assignment and equality (and thus inequality). That's necessary because we are meant to be able to manipulate objects of any type in the entire set of types via the partial view presented by the root type, without knowing which specific tagged type is involved.

17.3 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

```
package Simple_Recs is

   type Rec is limited record
      I : Integer;
   end record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Explicitly_Limited_
↪Types.Explicitly_Limited_Types
MD5: de73a20140628420830ed9fe0b2dedb5
```

The Rec type is also explicitly limited when it's declared limited in the private type's completion (in the package's private part):

Listing 31: simple_recs.ads

```
package Simple_Recs is

   type Rec is limited private;

private

```

(continues on next page)

(continued from previous page)

```
   type Rec is limited record
      I : Integer;
   end record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Explicitly_Limited_
↪Types.Explicitly_Limited_Types
MD5: ececb364f5365a74db43952e9421dee0
```

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 789), 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:

Listing 32: simple_recs.ads

```
package Simple_Recs is

   type Rec is limited private;

private

   type Rec is record
      I : Integer;
   end record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Explicitly_Limited_
↪Types.Explicitly_Limited_Types
MD5: bd54dec4f9b67d3d14d80511b3ac311f
```

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 794) are always explicitly limited types — because, as we've learned before, they cannot have a nonlimited type declaration in its full view.

In the Ada Reference Manual

- 6.2 Formal Parameter Modes[313]
- 6.4.1 Parameter Associations[314]
- 7.5 Limited Types[315]

313 http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
314 http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
315 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html

17.4 Subtypes of Limited Types

We can declare subtypes of limited types. For example:

Listing 33: simple_recs.ads

```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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Limited_Subtype
MD5: 2a82c3c96fad2a01b9a8c15912d4b974
```

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:

Listing 34: test_limitedness.adb

```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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Limited_Subtype
MD5: c24d07be96f27298a97e18d955cc6161
```

Build output

```
test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

As expected, compilations fails because `Limited_Integer_Array_2` is a limited (sub)type.

17.5 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

```
package Simple_Recs is

   type Rec is limited null record;

   type Rec_Derived is new Rec;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Limited_Type
MD5: cd23dfb69645ba5f1ebfdd65ee761ebe
```

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:

```
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

```
with Simple_Recs; use Simple_Recs;

procedure Test_Limitedness is
   Dummy_1, Dummy_2 : Rec_Derived;
begin
   Dummy_2 := Dummy_1;
end Test_Limitedness;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Limited_Type
MD5: ce1b5fc8c96c4ede0cc6768b84296b51
```

Build output

```
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

```
package Simple_Recs is

   type Rec is null record;

   type Rec_Derived is limited new Rec;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Limited_Type_Nonlimited_Ancestor
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[316]
> - 7.5 Limited Types[317]

17.5.1 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;
```

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;

```

(continues on next page)

316 http://www.ada-auth.org/standards/22rm/html/RM-7-3.html
317 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html

(continued from previous page)

```
procedure Test_Limitedness is
   Dummy_1, Dummy_2 : Rec_Derived;
begin
   Dummy_2 := Dummy_1;
end Test_Limitedness;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Limited_Private_Type
MD5: c6eed14520589b9c1e11c17bd6179c19
```

Build output

```
test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

Here, Rec_Derived is a limited type derived from the (limited private) Rec type. We can verify that Rec_Derived type is limited because the compilation of the Test_Limitedness procedure fails.

17.5.2 Deriving from non-explicitly limited private types

Up to this point, we have discussed *explicitly limited types* (page 795). 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

```
package Simple_Recs is

   type Rec is limited private;

private

   type Rec is null record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Limited_Private_Type
MD5: 30a2a88ff46b7e528bb8d75d3d6ad6ce
```

Build output

```
simple_recs.ads:1: Simple_Recs cannot be used as a main program
gprbind: invocation of gnatbind failed
gprbuild: unable to bind simple_recs.ads
```

Here, Rec_Derived is a limited type because the partial view of Rec is limited. The fact that the full view of Rec is nonlimited doesn't affect the Rec_Derived type — as we can verify with the compilation error in the Test_Limitedness procedure.

Note, however, that a derived type becomes nonlimited in the **private part or the body** of a child package if it isn't explicitly limited. In this sense, the derived type inherits the *nonlimitedness* of the parent's full view. For example, because we're declaring Rec_Derived as **is**

new Rec in the child package (`Simple_Recs.Ext`), we're saying that `Rec_Derived` is limited *outside* this package, but nonlimited in the private part and body of the `Simple_Recs.Ext` package. We can verify this by copying the code from the `Test_Limitedness` procedure to a new procedure in the body of the `Simple_Recs.Ext` package:

Listing 42: simple_recs-ext.ads

```
package Simple_Recs.Ext
  with Elaborate_Body is

  --  Rec_Derived is derived from Rec, which is a
  --  limited private type that is nonlimited in
  --  its full view.
  --
  --  Rec_Derived isn't explicitly limited.
  --  Therefore, it's nonlimited in the private
  --  part of Simple_Recs.Ext and its package
  --  body.
  --
  type Rec_Derived is new Rec;

end Simple_Recs.Ext;
```

Listing 43: simple_recs-ext.adb

```
package body Simple_Recs.Ext is

   procedure Test_Child_Limitedness is
      Dummy_1, Dummy_2 : Rec_Derived;
   begin
      --  Here, Rec_Derived is a nonlimited
      --  type because Rec is nonlimited in
      --  its full view.

      Dummy_2 := Dummy_1;
   end Test_Child_Limitedness;

end Simple_Recs.Ext;
```

Listing 44: test_limitedness.adb

```
--  We copied the code to the
--  Test_Child_Limitedness procedure (in the
--  body of the Simple_Recs.Ext package) and
--  commented it out here.
--
--  You may uncomment the code to verify
--  that Rec_Derived is limited in this
--  procedure.
--

--  with Simple_Recs.Ext; use Simple_Recs.Ext;

procedure Test_Limitedness is
   --  Dummy_1, Dummy_2 : Rec_Derived;
begin
   --  Dummy_2 := Dummy_1;
   null;
end Test_Limitedness;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Limited_Private_Type
MD5: f480cd05afff622e451684a0293cb982
```

In the `Test_Child_Limitedness` procedure of the `Simple_Recs.Ext` package, we can use the `Rec_Derived` as a nonlimited type because its ancestor Rec is nonlimited in its full view. (*As we've learned before* (page 791), 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 795).

Let's look at an example:

Listing 45: simple_recs.ads

```ada
package Simple_Recs is

   type Tagged_Rec is tagged limited private;

private

   type Tagged_Rec is tagged limited null record;

end Simple_Recs;
```

Listing 46: simple_recs-ext.ads

```ada
package Simple_Recs.Ext is

   type Rec_Derived is new
     Tagged_Rec with private;

private

   type Rec_Derived is new
     Tagged_Rec with null record;

end Simple_Recs.Ext;
```

Listing 47: test_limitedness.adb

```ada
with Simple_Recs.Ext; use Simple_Recs.Ext;

procedure Test_Limitedness is
   Dummy_1, Dummy_2 : Rec_Derived;
begin
   Dummy_2 := Dummy_1;
end Test_Limitedness;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Tagged_Limited_Private_Type
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

```ada
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Tagged_Limited_Private_Type
MD5: b82a58a4bf9701b321000c52bf121977
```

Build output

```
simple_recs-ext.ads:1: Simple_Recs.ext cannot be used as a main program
gprbind: invocation of gnatbind failed
gprbuild: unable to bind simple_recs-ext.ads
```

(Obviously, if we include the **limited** keyword in the partial view of the derived type, we must include it in its full view as well.)

Deriving from limited interfaces

The rules for limited interfaces are different from the ones for limited tagged types. In contrast to the rule we've seen in the previous section, a type that is derived from a limited type isn't automatically limited. In other words, it does **not** inherit the *limitedness* from the interface. For example:

Listing 49: simple_recs.ads

```ada
package Simple_Recs is

   type Limited_IF is limited interface;

end Simple_Recs;
```

Listing 50: simple_recs-ext.ads

```
package Simple_Recs.Ext is

   type Rec_Derived is new
     Limited_IF with private;

private

   type Rec_Derived is new
     Limited_IF with null record;

end Simple_Recs.Ext;
```

Listing 51: 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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Interface_Limited_Private
MD5: d9cf0bd26b86d0caec82eff2a2ec6ead
```

Here, Rec_Derived is derived from the limited Limited_IF interface. As we can see, the Test_Limitedness compiles fine because Rec_Derived is nonlimited.

Of course, if we want Rec_Derived to be limited, we can make this explicit in the type declaration:

Listing 52: simple_recs-ext.ads

```
package Simple_Recs.Ext is

   type Rec_Derived is limited new
     Limited_IF with private;

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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↪Limited_Types.Derived_Interface_Limited_Private
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.

17.6 Immutably Limited Types

According to the Annotated Ada Reference Manual (7.5, 8.b/3)[318], "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 789), 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* (page 795)
- tagged limited types (i.e. with the keywords **tagged limited**);
- *tagged limited private types* (page 794);
- limited private type that have at least one *access discriminant* (page 725) 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:

Listing 54: show_immutably_limited_types.ads

```ada
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;
```

(continues on next page)

318 http://www.ada-auth.org/standards/22aarm/html/AA-7-5.html

(continued from previous page)

```

   --
   --  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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Immutably_Limited_
↪Types.Example
MD5: 6bcb9582a10eedc96040ab11cd320153
```

Build output

```
show_immutably_limited_types.ads:31:30: warning: coextension will not be␣
↪deallocated when its associated owner is deallocated [enabled by default]
```

In the Show_Immutably_Limited_Types package above, we see multiple instances of immutably limited types. (The comments in the source code indicate each type.)

In the Ada Reference Manual

- 7.5 Limited Types[319]

17.6.1 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* (page 789). In addition, we have nonsynchronized limited interface types. As mentioned earlier in this chapter, a *type derived from a nonsynchronized limited interface* (page 803), can be nonlimited, so it's not immutably limited.

In the Ada Reference Manual

- 7.3.1 Private Operations[320]
- 7.5 Limited Types[321]

319 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html
320 http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html
321 http://www.ada-auth.org/standards/22rm/html/RM-7-5.html

17.7 Limited Types with Discriminants

In this section, we look into the implications of using discriminants with limited types. Actually, most of the topics mentioned here have already been covered in different sections of previous chapters, as well as in this chapter. Therefore, this section is in most parts just a review of what we've already discussed.

Let's start with a simple example:

Listing 56: simple_recs.ads

```
package Simple_Recs is

   type Rec (L : Positive)
     is limited null record;

end Simple_Recs;
```

Listing 57: test_limitedness.adb

```
with Simple_Recs; use Simple_Recs;

procedure Test_Limitedness is
   Dummy_1 : Rec (2);
   Dummy_2 : Rec (3);
begin
   Dummy_2 := Dummy_1;
   --  ^^^^^^^^^^^^^^
   --  ERRORS:
   --    1. Cannot assign objects of
   --       limited types.
   --    2. Cannot assign objects with
   --       different discriminants.
end Test_Limitedness;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Simple_Example
MD5: 7b4a62c0341becf16f59e163b4359397
```

Build output

```
test_limitedness.adb:7:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

In this example, we see the declaration of the limited type Rec, which has the discriminant L. For objects of type Rec, we not only have the typical restrictions that *equality and assignment aren't available* (page 782), but we also have the restriction that we won't be able to assign objects with different discriminants.

In the Ada Reference Manual

- 3.7 Discriminants[322]

[322] http://www.ada-auth.org/standards/12rm/html/RM-3-7.html

17.7.1 Default Expressions

On the other hand, there are restrictions that apply to nonlimited types with discriminants, but not to limited types with discriminants. This concerns mostly default expressions, which are generally allowed for discriminants of limited types.

Discriminants of tagged limited types

As we've discussed previously, we can use default expressions for discriminants of tagged limited types. Let's see an example:

Listing 58: recs.ads

```
package Recs is

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     tagged limited null record;

end Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Discriminant_Default_Value_Tagged_TYpe
MD5: ebd28ee124c6a84765c61ea609ba0595
```

Obviously, the same applies to *tagged limited private types* (page 794):

Listing 59: recs.ads

```
package Recs is

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     tagged limited private;

private

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     tagged limited null record;

end Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Discriminant_Default_Value_Tagged_TYpe
MD5: dd3d5a25ad9d050f7e7467d859dd9e14
```

In the case of tagged, nonlimited types, using default expressions in this context isn't allowed.

Access discriminant

Similarly, when using limited types, we can specify default expressions for *access discriminants* (page 725):

Listing 60: custom_recs.ads

```
package Custom_Recs is

   --  Specifying a default expression for
   --  an access discriminant:
   type Rec (IA : access Integer :=
                  new Integer'(0)) is limited
   record
      I : Integer := IA.all;
   end record;

end Custom_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Access_Discriminant_Default_Expression
MD5: 23703d9dc80e9f1c8fe237c76b9dd6b0
```

Build output

```
custom_recs.ads:6:21: warning: coextension will not be deallocated when its␣
↪associated owner is deallocated [enabled by default]
```

In fact, *as we've discussed before* (page 727), this isn't possible for nonlimited types.

Note, however, that we can only assign a default expression to an access discriminant of an *immutably limited type* (page 805).

Discriminants of nontagged limited types

In addition to tagged limited types, we can use default expressions for discriminants of nontagged limited types. Let's see an example:

Listing 61: recs.ads

```
package Recs is

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     limited null record;

end Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Discriminant_Default_Value_Tagged_TYpe
MD5: 8d189b814c6afb4f060e9a41558c18c6
```

Obviously, the same applies to *limited private types* (page 787):

Listing 62: recs.ads

```
package Recs is

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     limited private;

```

(continues on next page)

(continued from previous page)

```
private

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     limited null record;

end Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Discriminant_Default_Value_Tagged_TYpe
MD5: 536d7dfbe84c818cb94a8b972e3d77cb
```

Note that using default expressions for discriminants of nonlimited, nontagged types is OK as well.

Mutable subtypes and Limitedness

As we've mentioned before, an unconstrained discriminated subtype with defaults is called a mutable subtype. An important feature of mutable subtypes is that it allows changing the discriminants of an object, e.g. via assignments. However, as we know, we cannot assign to objects of limited types. Therefore, in essence, a type should be nonlimited to be considered a mutable subtype.

Let's look at a code example:

Listing 63: recs.ads

```
package Recs is

   type LTT (L : Positive := 1;
             M : Positive := 2) is
     limited null record;

   function Init (L : Positive;
                  M : Positive)
                  return LTT is
     ((L => L, M => M));

   procedure Copy (From :        LTT;
                   To   : in out LTT);

end Recs;
```

Listing 64: recs.adb

```
package body Recs is

   procedure Copy (From :        LTT;
                   To   : in out LTT) is
   begin
      To := Init (L => From.L,
                  M => From.M);
      --  ERROR: cannot assign to object of
      --         limited type

      To.L := From.L;
      To.M := From.M;
      --  ERROR: cannot change discriminants
```

(continues on next page)

(continued from previous page)

```
   end Copy;

end Recs;
```

Listing 65: show.adb

```
with Recs; use Recs;

procedure Show is
   A : LTT;
   B : LTT := Init (10, 12);
begin
   Copy (From => B, To => A);
end Show;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Discriminant_Default_Value_Tagged_TYpe
MD5: e8dfb1e99e33923aa4023428ecb17372
```

Build output

```
recs.adb:6:07: error: left hand of assignment must not be limited type
recs.adb:11:09: error: assignment to discriminant not allowed
recs.adb:12:09: error: assignment to discriminant not allowed
gprbuild: *** compilation phase failed
```

As we can see in the Copy procedure, it's not possible to properly assign to the target object. Using Init is forbidden because the assignment is not initializing the target object — as we're not declaring To at this point. Also, changing the individual discriminants is forbidden as well. Therefore, we don't have any means to change the discriminants of the target object. (In contrast, if LTT was a nonlimited type, we would be able to implement Copy by using the call to the Init function.)

17.7.2 Limited private type with unknown discriminants

We can declare limited private types with *unknown discriminants* (page 221). Let's see an example:

Listing 66: limited_private_unknown_discriminants.ads

```
package Limited_Private_Unknown_Discriminants is

   type Rec (<>) is limited private;

private

   type Rec is limited
   record
      I : Integer;
   end record;

end Limited_Private_Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Limited_Private_Unknown_Discriminants
MD5: 74184919132a084da76bd3e1445c22e5
```

In this example, we declare type Rec, which has unknown discriminants.

As we mentioned earlier, when we use a private type with unknown discriminants, we gain extra control over its initialization. In addition, if we declare those types as limited, we gain even more control. In fact, this is what the Annotated Ada Reference Manual (3.7, 26.b/2)[323] says:

> "A subtype with unknown discriminants is indefinite, and hence an object of such a subtype needs explicit initialization. A limited private type with unknown discriminants is 'extremely' limited; objects of such a type can be initialized only by subprograms (either procedures with a parameter of the type, or a function returning the type) declared in the package. Subprograms declared elsewhere can operate on and even return the type, but they can only initialize the object by calling (ultimately) a subprogram in the package declaring the type. Such a type is useful for keeping complete control over object creation within the package declaring the type."

Let's reuse a code example from the *previous section on unknown discriminants* (page 223) and use limited types:

Listing 67: limited_private_unknown_discriminants.ads

```
package Limited_Private_Unknown_Discriminants is

   type Rec (<>) is limited private;

   function Init return Rec;

private

   type Rec is limited
   record
      I : Integer;
   end record;

   function Init return Rec is
     ((I => 0));

end Limited_Private_Unknown_Discriminants;
```

Listing 68: show_constructor_function.adb

```
with Limited_Private_Unknown_Discriminants;
use  Limited_Private_Unknown_Discriminants;

procedure Show_Constructor_Function is
   R : Rec := Init;
begin
   null;
end Show_Constructor_Function;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↪Limited_Private_Unknown_Discriminants
MD5: f4b1de2a83837e2e52b0b57214f0eaf9
```

A function such as Init is called a *constructor function for limited types* (page 819). We discuss this topic in more detail later on.

[323] http://www.ada-auth.org/standards/22aarm/html/AA-3-7.html

17.8 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 69: simple_recs.ads

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Record_Components_
↪Limited_Type.Record_Components_Limited_Type
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 70: simple_recs.ads

```
package Simple_Recs is

   type Int_Rec is limited record
      V : Integer;
   end record;

   type Rec is record
      IR : Int_Rec;
   end record;

end Simple_Recs;
```

Listing 71: show_implicitly_limited.adb

```
with Simple_Recs; use Simple_Recs;

procedure Show_Implicitly_Limited is
   A, B : Rec;
begin
   B := A;
end Show_Implicitly_Limited;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Record_Components_
↪Limited_Type.Record_Components_Limited_Type
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[324]

17.9 Limited types and aggregates

Note

This section was originally written by Robert A. Duff and published as Gem #1: Limited Types in Ada 2005[325] and Gem #2[326].

In this section, we focus on using aggregates to initialize limited types.

Historically

Prior to Ada 2005, aggregates were illegal for limited types. Therefore, we would be faced with a difficult choice: Make the type limited, and initialize it like this:

Listing 72: 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;
```

[324] http://www.ada-auth.org/standards/22rm/html/RM-3-8.html
[325] https://www.adacore.com/gems/gem-1
[326] https://www.adacore.com/gems/gem-2

Listing 73: show_non_aggregate_init.adb

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

with Persons; use Persons;

procedure Show_Non_Aggregate_Init is
   X : Limited_Person;
begin
   X.Name := To_Unbounded_String ("John Doe");
   X.Age := 25;
end Show_Non_Aggregate_Init;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Aggregates.Full_Coverage_Rules_Limited_Ada95
MD5: fd3dcb6251f7b6912dafcca052932be2
```

which has the maintenance problem the full coverage rules are supposed to prevent. Or, make the type nonlimited, and gain the benefits of aggregates, but lose the ability to prevent copies.

17.9.1 Full coverage rules for limited types

Previously, we discussed *full coverage rules for aggregates* (page 264). 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 74: persons.ads

```
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
      Self : Limited_Person_Access :=
               Limited_Person'Unchecked_Access;
      Name : Unbounded_String;
      Age  : Natural;
      Shoe_Size : Positive;
   end record;

end Persons;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Aggregates.Full_Coverage_Rules_Limited
MD5: b8ece44a10d512061cb138be21e42034
```

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 75: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Aggregates.Full_Coverage_Rules_Limited
MD5: ded40ff29b53ea5528efba94efaadbec
```

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.Self` to be `X'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:

Listing 76: show_aggregate_init.adb

```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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Aggregates.Full_Coverage_Rules_Limited
MD5: 793ee000fd777d0aa5c15e16132ec411
```

It seems uncomfortable to set the value of Self to the wrong value (**null**) and then correct it. It also seems annoying that we have a (correct) default value for Self, but prior to Ada 2005, we couldn't use defaults with aggregates. Since Ada 2005, a new syntax in aggregates is available: <> means "use the default value, if any". Therefore, we can replace Self => **null** by Self => <>.

Important

Note that using <> in an aggregate can be dangerous, because it can leave some components uninitialized. <> means "use the default value". If the type of a component is scalar, and there is no record-component default, then there is no default value.

For example, if we have an aggregate of type **String**, like this:

Listing 77: show_string_box_init.adb

```
procedure Show_String_Box_Init is
    Uninitialized_Const_Str : constant String :=
                                (1 .. 10 => <>);
begin
   null;
end Show_String_Box_Init;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Aggregates.String_Box_Init
MD5: 28931ced4e1113d55bdc9dc64b42f70a
```

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 78: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Aggregates.Dangerous_String
MD5: 6c26e9c8d5d031d4e6eac1ac8458f17e
```

Build output

```
show_dangerous_string.adb:2:05: warning: variable "Uninitialized_String_Var" is
read but never assigned [-gnatwv]
```

As always, one must be careful about uninitialized scalar objects.

17.10 Constructor functions for limited types

Note

This section was originally written by Robert A. Duff and published as Gem #3[327].

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:

Listing 79: p.ads

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

package P is
   task type Some_Task_Type;

   protected type Some_Protected_Type is
      --  dummy type
   end Some_Protected_Type;

   type T (<>) is limited private;
   function Make_T (Name : String) return T;
   --        ^^^^^^
   --  constructor function
private
   type T is limited
      record
         Name    : Unbounded_String;
         My_Task : Some_Task_Type;
         My_Prot : Some_Protected_Type;
      end record;
end P;
```

[327] https://www.adacore.com/gems/gem-3

Listing 80: p.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
↪Functions_Limited_Types.Constructor_Functions
MD5: 2e73eea0ba7852d45ba96dc1f6fae14d
```

Given the above, clients can say:

Listing 81: show_constructor_function.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
↪Functions_Limited_Types.Constructor_Functions
MD5: 52801fafbd58fedbf268a6704008627b
```

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 82: 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 :=
```

(continues on next page)

(continued from previous page)

```
      Make_Rumplestiltskin;
begin
   null;
end Show_Rumplestiltskin_Constructor;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
↪Functions_Limited_Types.Constructor_Functions
MD5: d8d9e9f22a0f2f034057fe97f75eacfe
```

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.

As we discussed before (page 812), the (<>) on type T means that it has *unknown discriminants* from the point of view of the client. This is a trick that prevents clients from creating default-initialized objects (that is, X : T; is illegal). Thus clients must call Make_T whenever an object of type T is created, giving package P full control over initialization of objects.

Ideally, limited and nonlimited types should be just the same, except for the essential difference: you can't copy limited objects (and there's no language-defined equality operator). By allowing functions and aggregates for limited types, we're very close to this goal. Some languages have a specific feature called *constructor*. In Ada, a *constructor* is just a function that creates a new object.

Historically

Prior to Ada 2005, *constructors* only worked for nonlimited types. For limited types, the only way to *construct* on declaration was via default values, which limits you to one constructor. And the only way to pass parameters to that construction was via discriminants.

Consider the following package:

Listing 83: aux.ads

```ada
with Ada.Containers.Ordered_Sets;

package Aux is
   generic
      with package OS is new
        Ada.Containers.Ordered_Sets (<>);
   function Gen_Singleton_Set
     (Element : OS.Element_Type)
      return OS.Set;
end Aux;
```

Listing 84: aux.adb

```
package body Aux is
   function Gen_Singleton_Set
     (Element : OS.Element_Type)
      return OS.Set
   is
   begin
      return S : OS.Set := OS.Empty_Set do
         S.Insert (Element);
      end return;
   end Gen_Singleton_Set;
end Aux;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
↪Functions_Limited_Types.Constructor_Functions_2
MD5: b715ae504c49ed59b7fd5ead4cc7bbb4
```

Since Ada 2005, we can say:

Listing 85: show_set_decl.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
↪Functions_Limited_Types.Constructor_Functions_2
MD5: 443fc3390b0f3e5516d91c80f16bed3f
```

whether or not Set is limited. `This_Set : Set := Empty_Set;` seems clearer than:

Listing 86: show_set_decl.adb

```
with Ada.Containers.Ordered_Sets;

procedure Show_Set_Decl is

   package Integer_Sets is new
     Ada.Containers.Ordered_Sets
       (Element_Type => Integer);
   use Integer_Sets;

   This_Set : Set;
begin
   null;
end Show_Set_Decl;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
↪Functions_Limited_Types.Constructor_Functions_2
MD5: e5b6c0e148cfdb1987ab3002ec1f53bd
```

which might mean "default-initialize to the empty set" or might mean "leave it uninitialized, and we'll initialize it in later".

17.11 Return objects

17.11.1 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[328].

Previously, we discussed *extended return statements* (page 462). 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 87: task_construct_error.ads

```
package Task_Construct_Error is

   task type Task_Type (Discriminant : Integer);

   function Make_Task (Val : Integer)
                       return Task_Type;

end Task_Construct_Error;
```

Listing 88: task_construct_error.adb

```
package body Task_Construct_Error is

   task body Task_Type is
   begin
      null;
```

(continues on next page)

328 https://www.adacore.com/gems/ada-gem-10

(continued from previous page)

```
   end Task_Type;

   function Make_Task (Val : Integer)
                       return Task_Type
   is
      Result : Task_Type
                 (Discriminant => Val * 3);
   begin
      --  some statements...
      return Result; -- Illegal!
   end Make_Task;

end Task_Construct_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_
↪Statements_Limited_Types.Extended_Return_Limited_Error
MD5: f55b1c367d2931ece4d352d209fe6b3b
```

The return statement here is illegal, because Result is local to Make_Task, and returning it would involve a copy, which makes no sense (which is why task types are limited). Since Ada 2005, we can write constructor functions for task types:

Listing 89: 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 90: task_construct.adb

```
package body Task_Construct is

   task body Task_Type is
   begin
      null;
   end Task_Type;

   function Make_Task (Val : Integer)
                       return Task_Type is
   begin
      return Result : Task_Type
                        (Discriminant => Val * 3)
      do
         --  some statements...
         null;
      end return;
   end Make_Task;

end Task_Construct;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_
```

(continues on next page)

(continued from previous page)

```
↪Statements_Limited_Types.Extended_Return_Limited
MD5: c91a24f09a76aef1c25d1a55bcbee910
```

If we call it like this:

Listing 91: show_task_construct.adb

```ada
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_
↪Statements_Limited_Types.Extended_Return_Limited
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.

17.11.2 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`:

Listing 92: limited_types.ads

```ada
package Limited_Types is

   type Integer_Access is access Integer;

   type Simple_Rec is limited private;

   function Init (I : Integer) return Simple_Rec;

private

   type Simple_Rec is limited record
      V : Integer_Access;
   end record;

end Limited_Types;
```

Listing 93: 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 94: 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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_
↪Statements_Limited_Types.Initialization_Return_Do
MD5: 67235f804206e07fa4eba3a45cc1096f
```

Runtime output

```
E'Address (Init):  00007FFD2287A1D8
A'Address (local): 00007FFD2287A1D8
----
```

(continues on next page)

(continued from previous page)

```
E'Address (Init):  00007FFD2287A1D0
B'Address (local): 00007FFD2287A1D0
```

When running this code example and comparing the address of the object E in the Init function and the object that is being initialized in the Show_Limited_Init procedure, we see that the return object E (of the Init function) and the local object in the 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 Simple_Rec nonlimited:

Listing 95: non_limited_types.ads

```ada
package Non_Limited_Types is

   type Integer_Access is access Integer;

   type Simple_Rec is private;

   function Init (I : Integer)
                  return Simple_Rec;

private

   type Simple_Rec is record
      V : Integer_Access;
   end record;

end Non_Limited_Types;
```

Listing 96: non_limited_types.adb

```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 return;
   end Init;

end Non_Limited_Types;
```

Listing 97: show_non_limited_init_by_copy.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_
↪Types.Extended_Return_Statements_Limited_Types.
↪Initialization_Return_Copy
MD5: 6e224b64b90dabdf5064c70364fa80cb
```

Runtime output

```
E'Address (Init):  00007FFE90B49890
A'Address (local): 00007FFE90B49988
----
E'Address (Init):  00007FFE90B49890
B'Address (local): 00007FFE90B49980
```

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.

17.12 Building objects from constructors

Note

This section was originally written by Robert A. Duff and published as Gem #11: Limited Types in Ada 2005[329].

We've earlier seen examples of constructor functions for limited types similar to this:

[329] https://www.adacore.com/gems/ada-gem-11

Listing 98: p.ads

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

package P is
   task type Some_Task_Type;

   protected type Some_Protected_Type is
      --  dummy type
   end Some_Protected_Type;

   type T is limited private;
   function Make_T (Name : String) return T;
   --        ^^^^^^
   --  constructor function
private
   type T is limited
      record
         Name    : Unbounded_String;
         My_Task : Some_Task_Type;
         My_Prot : Some_Protected_Type;
      end record;
end P;
```

Listing 99: p.adb

```
package body P is

   task body Some_Task_Type is
   begin
      null;
   end Some_Task_Type;

   protected body Some_Protected_Type is
   end Some_Protected_Type;

   function Make_T (Name : String) return T is
   begin
      return (Name   =>
                To_Unbounded_String (Name),
              others => <>);
   end Make_T;

end P;
```

Listing 100: p-aux.ads

```
package P.Aux is
   function Make_Rumplestiltskin return T;
end P.Aux;
```

Listing 101: p-aux.adb

```
package body P.Aux is

   function Make_Rumplestiltskin return T is
   begin
      return Make_T (Name => "Rumplestiltskin");
   end Make_Rumplestiltskin;
```

(continues on next page)

(continued from previous page)

```

end P.Aux;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↪From_Constructors.Building_Objs_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 102: 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: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↪From_Constructors.Building_Objs_From_Constructors
MD5: 2fe193516df6452eccece8132660f8e5
```

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 103: 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: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↪From_Constructors.Building_Objs_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:

Listing 104: show_heap_constructor.adb

```ada
with P;     use P;
with P.Aux; use P.Aux;
```

(continues on next page)

(continued from previous page)

```

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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↪From_Constructors.Building_Objs_From_Constructors
MD5: 8eb794884f1dfbdbedf1bc4369f45cf8
```

The result of the function is built directly in the heap-allocated object, which will be finalized when the scope of T_Ref is left (long after Heap_Alloc returns).

We can create another limited type with a component of type T, and use an aggregate:

Listing 105: show_outer_type.adb

```
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;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↪From_Constructors.Building_Objs_From_Constructors
MD5: 00817649406492b79977d67eb0fd3955
```

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 106: show_illegal_constructor.adb

```
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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↪From_Constructors.Building_Objs_From_Constructors
MD5: f7b0c78e9fbe2e104b82dfff25ac3e3a
```

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 107: show_illegal_constructor.adb

```
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;
```

17.13 Limited types as parameter

Previously, we saw that *parameters can be passed by copy or by reference* (page 465). Also, we discussed the concept of by-copy and by-reference types. *Explicitly limited types* (page 795) are by-reference types. Consequently, parameters of these types are always passed by reference.

For further reading...

As an example of the importance of this rule, consider the case of a lock (as an abstract data type). If such a lock object were passed by copy, the `Acquire` and `Release` operations would be working on copies of this object, not on the original one. This would lead to timing-dependent bugs.

Let's reuse an example of an explicitly limited type:

Listing 108: simple_recs.ads

```
package Simple_Recs is

   type Rec is limited record
      I : Integer;
   end record;

end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Parameters.Explicitly_Limited_Types
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:

Listing 109: simple_recs.ads

```
with System;

package Simple_Recs is

   type Rec is limited record
      I : Integer;
   end record;

   procedure Proc (R : in out Rec;
                   A :    out System.Address);

end Simple_Recs;
```

Listing 110: simple_recs.adb

```
package body Simple_Recs is

   procedure Proc (R : in out Rec;
                   A :    out System.Address) is
   begin
      R.I := 0;
      A   := R'Address;
   end Proc;

end Simple_Recs;
```

Listing 111: test.adb

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

procedure Test is
   R : Rec;

```

(continues on next page)

(continued from previous page)

```
   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

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↪Parameters.Explicitly_Limited_Types
MD5: d4fe2bb47d2223ef013d22aa305403e5
```

Runtime output

```
R'Address (Proc): 00007FFCB8ABA9DC
R'Address (Test): 00007FFCB8ABA9DC
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[330]
- 6.4.1 Parameter Associations[331]
- 7.5 Limited Types[332]

[330] http://www.ada-auth.org/standards/22rm/html/RM-6-2.html
[331] http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html
[332] http://www.ada-auth.org/standards/22rm/html/RM-7-5.html

CHAPTER

EIGHTEEN

CONTROLLED TYPES

18.1 Overview

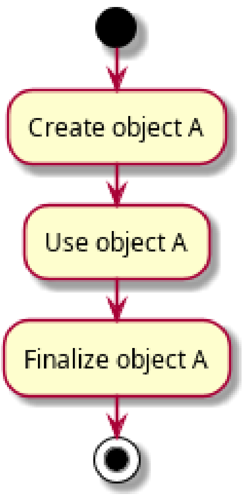
In this section, we introduce the concept of controlled types. We start with a review of lifetime of objects and discuss how controlled types allow us to control the initialization, post-copy (e.g. assignment) adjustment and finalization of objects.

Relevant topics

- Assignment and Finalization[333]

18.1.1 Lifetime of objects

We already talked about the lifetime of objects[334] previously in the context of *access types* (page 645). Again, we assume you understand the concept. In any case, let's quickly review the typical lifetime of an object:



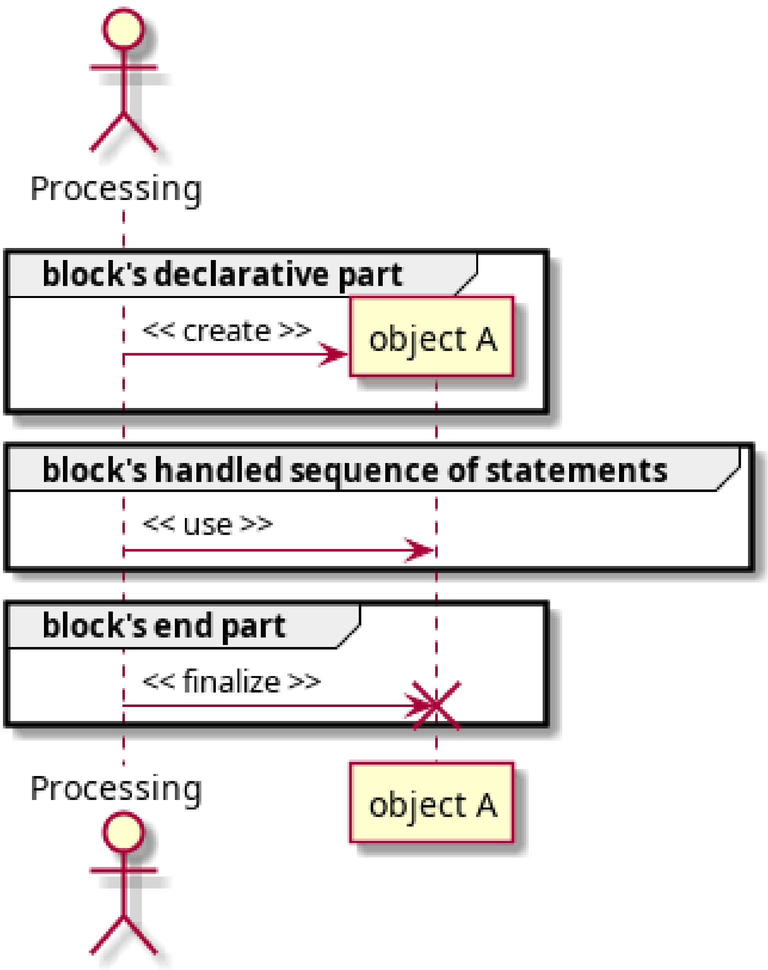
In simple terms, an object A is first created before we can make use of it. When object A is about to get out of scope, it is finalized. Note that finalization might not entail any actual code execution — but it often does.

Let's analyze the lifetime of object A in a procedure P:

[333] http://www.ada-auth.org/standards/22rm/html/RM-7-6.html
[334] https://en.wikipedia.org/wiki/Variable_(computer_science)#Scope_and_extent

```
procedure P is
   A : T;
begin
   P2 (A);
end P;
```

We could visualize the lifetime as follows:



In other words, object A is created in the declarative part of P and then it's used in P's sequence of statements. Finally, A is finalized when P ends.

18.1.2 Initialization of objects

Typically, right after an object A is created, it is still uninitialized. Therefore, we have to explicitly initialize it with a meaningful initial value — or with the value returned by a function call, for example. Similarly, when an object A is about to get out of scope, it is going to be finalized (i.e. destroyed) and its contents are then lost forever.

As we know, for some standard Ada types, objects are initialized by default. For example, objects of access types are initialized by default to **null**. Likewise, we can declare *types with default initial value* (page 69):

Listing 1: main.adb

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

```

(continues on next page)

(continued from previous page)

```
procedure Main is
   type Int is new Integer
     with Default_Value => 42;

   I  : Int;
   AI : access Int;
begin
   Put_Line ("I : "
             & I'Image);
   Put_Line ("AI : "
             & AI'Image);
end Main;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.
↪Default_Initialization
MD5: 14a5929f0635f0f7843c883bab9021d8
```

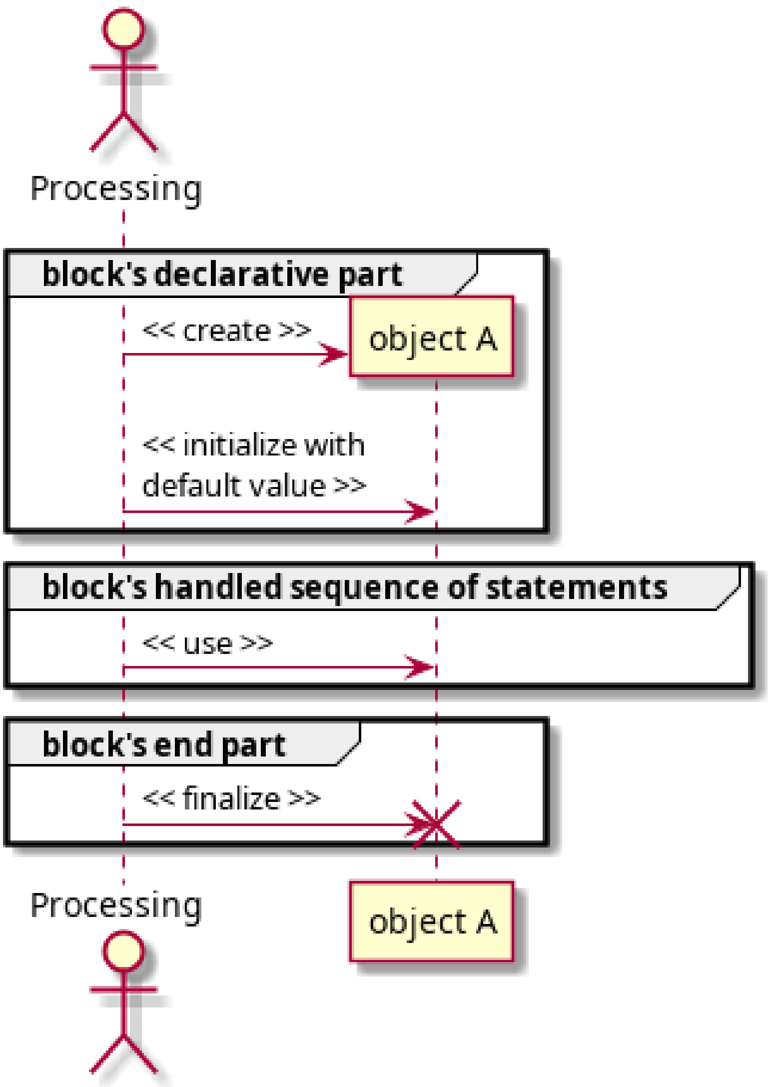
Build output

```
main.adb:8:04: warning: variable "AI" is read but never assigned [-gnatwv]
```

Runtime output

```
I :  42
AI : null
```

In this case, we can visualize the lifetime of those objects as follows:



Even though these default initialization methods provide some control over the objects, they might not be enough in certain situations. Also, we don't have any means to perform useful operations right before an object gets out of scope.

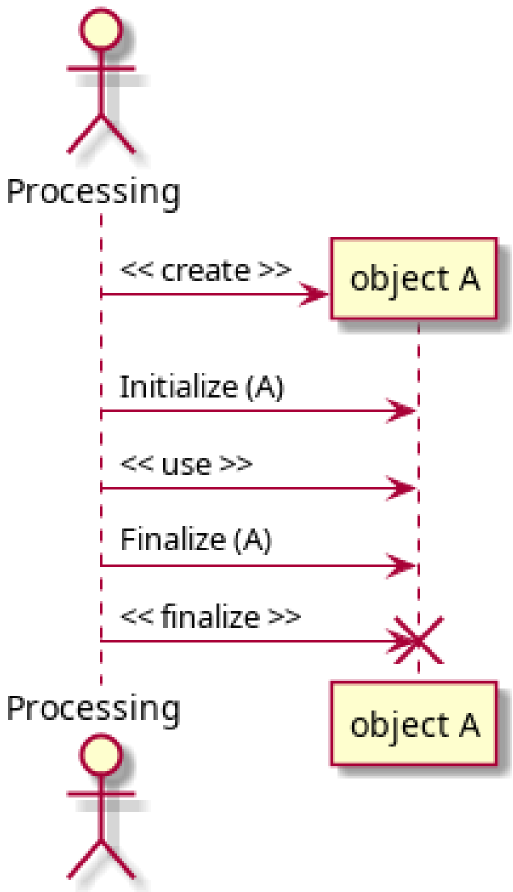
For further reading...

In general, record types have a very good default initialization capability. They're the most common completion for private types, so the facility is often used. In this sense, default initialization is the first choice, as it's guaranteed and requires nothing of the client. In addition, it's cheap at run-time compared to controlled types.

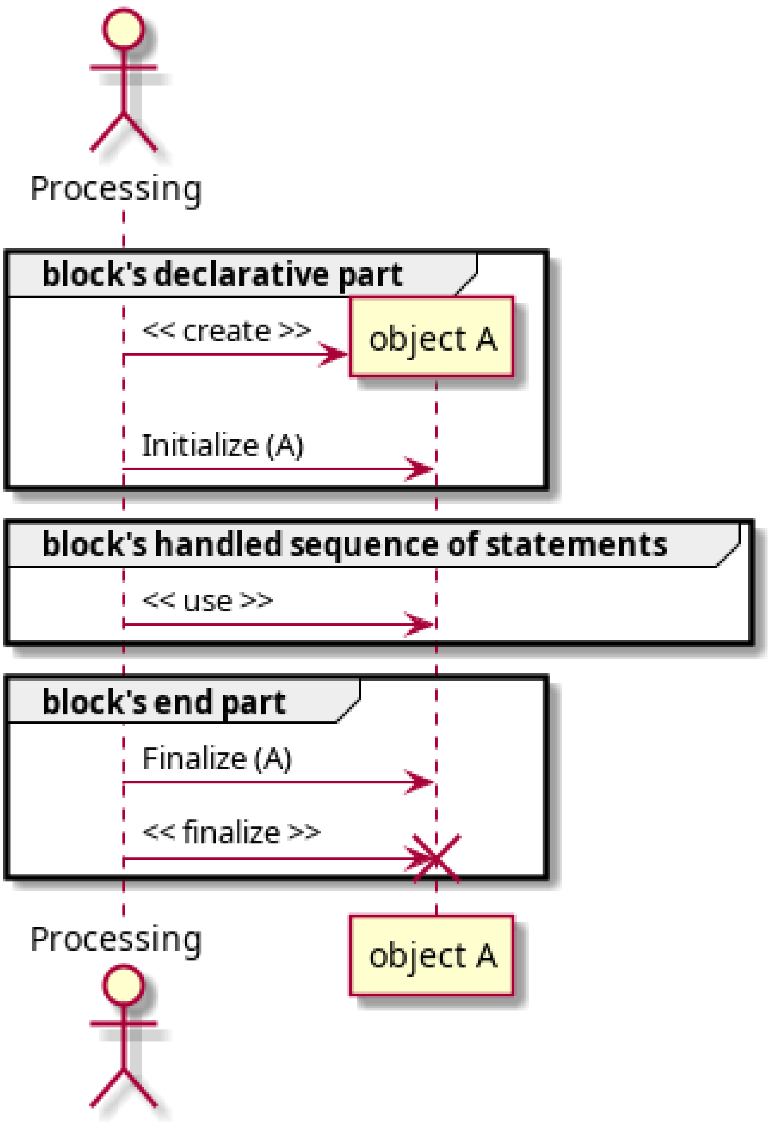
18.1.3 Controlled objects

Controlled objects allow us to better control the initialization and finalization of an object. For any controlled object `A`, an `Initialize (A)` procedure is called right *after* the object is created, and a `Finalize (A)` procedure is called right *before* the object is actually finalized.

We can visualize the lifetime of controlled objects as follows:



In the context of a block statement, the lifetime becomes:



Let's look at a simple example:

Listing 2: simple_controlled_types.ads

```
with Ada.Finalization;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Dummy (E : T);

private

   type T is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out T);

   overriding
```

(continues on next page)

(continued from previous page)

```
   procedure Finalize (E : in out T);

end Simple_Controlled_Types;
```

Listing 3: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy...)");
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize...");
   end Initialize;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize...");
   end Finalize;

end Simple_Controlled_Types;
```

Listing 4: show_controlled_types.adb

```
with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Controlled_Types is
   A : T;
   --
   --  This declaration roughly
   --  corresponds to:
   --
   --     A : T;
   --  begin
   --     Initialize (A);
   --
begin
   Dummy (A);

   --  When A is about to get out of
   --  scope:
   --
   --  Finalize (A);
   --
end Show_Controlled_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.Simple_
↪Example
MD5: 24f95418bb8c439648ab9dba9f0c953a
```

Runtime output

```
Initialize...
(Dummy...)
Finalize...
```

When we run this application, we see the user messages indicating the calls to Initialize and Finalize.

For further reading...

Note that if a controlled object isn't used in the application, the compiler might optimize it out. In this case, procedures Initialize and Finalize won't be called for this object, as it doesn't actually exist. You can see this effect by replacing the call to Dummy (A) in the Show_Controlled_Types procedure by a null statement (**null**).

18.1.4 Adjustment of controlled objects

An assignment is a full bit-wise copy of the entire right-hand side to the entire left-hand side. When copying controlled objects, however, we might need to adjust the target object. This is made possible by overriding the Adjust procedure, which is called right after the copy to an object has been performed. (As we'll see later on, *limited controlled types* (page 844) do not offer an Adjust procedure.)

The deep copy[335] of objects is a typical example where adjustments are necessary. When we assign an object B to an object A, we're essentially doing a shallow copy[336]. If we have references to other objects in the source object B, those references will be copied as well, so both target A and source B will be referring to the same objects. When performing a deep copy, however, we want the information from the dereferenced objects to be copied, not the references themselves. Therefore, we have to first allocate new objects for the target object A and copy the information from the original references — the ones we copied from the source object B — to the new objects. This kind of processing can be performed in the Adjust procedure.

As an example, let's extend the previous code example and override the Adjust procedure:

Listing 5: simple_controlled_types.ads

```
with Ada.Finalization;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Dummy (E : T);

private

   type T is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out T);

   overriding
   procedure Adjust (E : in out T);

```

(continues on next page)

335 https://en.wikipedia.org/wiki/Object_copying#Deep_copy
336 https://en.wikipedia.org/wiki/Object_copying#Shallow_copy

(continued from previous page)

```
   overriding
   procedure Finalize (E : in out T);

end Simple_Controlled_Types;
```

Listing 6: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy...)");
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize...");
   end Initialize;

   procedure Adjust (E : in out T) is
   begin
      Put_Line ("Adjust...");
   end Adjust;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize...");
   end Finalize;

end Simple_Controlled_Types;
```

Listing 7: show_controlled_types.adb

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

with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Controlled_Types is
   A, B : T;
begin
   Put_Line ("A := B");
   A := B;

   Dummy (A);
   Dummy (B);
end Show_Controlled_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.Simple_
↪Example_2
MD5: 4f4575dab6c9b384ea0cbd8bf9701850
```

Runtime output

```
Initialize...
Initialize...
```

(continues on next page)

(continued from previous page)

```
A := B
Finalize...
Adjust...
(Dummy...)
(Dummy...)
Finalize...
Finalize...
```

When running this application, we see that the `Adjust` procedure is called for object A — right after B is copied to A as part of the `A := B` assignment. We discuss more about this procedure *later on* (page 858).

18.1.5 Limited controlled types

Ada offers controlled types in two flavors: nonlimited controlled types — such as the ones we've seen so far — and limited controlled types. Both types are declared in the `Ada.Finalization` package.

The only difference between these types is that limited controlled types don't have an `Adjust` procedure that could be overridden, as limited types *do not permit direct copies of objects to be made via assignments* (page 782). (Obviously, both controlled and limited controlled types provide `Initialize` and `Finalize` procedures.)

The following table summarizes the information:

Type	Name	Initialize	Finalize	Adjust
Nonlimited Controlled	**Controlled**	Yes	Yes	Yes
Limited controlled	`Limited_Controlled`	Yes	Yes	Not available

18.1.6 Simple Example with ID

Although the previous code examples indicated that `Initialize`, `Finalize` and `Adjust` are called as we expect for controlled objects, they didn't show us exactly how those objects are actually handled. In this section, we discuss this by analyzing a code example that assigns a unique ID to each controlled object.

Let's start with the complete code example:

Listing 8: simple_controlled_types.ads

```
with Ada.Finalization;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Show (E    : T;
                   Name : String);

private

   protected Id_Gen is
      procedure New_Id (Id_Out : out Positive);
   private
      Id : Natural := 0;
   end Id_Gen;

   type T is new
```

(continues on next page)

(continued from previous page)

```
      Ada.Finalization.Controlled with
    record
       Id : Positive;
    end record;

    overriding
    procedure Initialize (E : in out T);

    overriding
    procedure Adjust (E : in out T);

    overriding
    procedure Finalize (E : in out T);

end Simple_Controlled_Types;
```

Listing 9: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   protected body Id_Gen is

      procedure New_Id (Id_Out : out Positive) is
      begin
         Id := Id + 1;
         Id_Out := Id;
      end New_Id;

   end Id_Gen;

   procedure Initialize (E : in out T) is
   begin
      Id_Gen.New_Id (E.Id);
      Put_Line ("Initialize: ID => "
                & E.Id'Image);
   end Initialize;

   procedure Adjust (E : in out T) is
      Prev_Id : constant Positive := E.Id;
   begin
      Id_Gen.New_Id (E.Id);
      Put_Line ("Adjust:     ID => "
                & E.Id'Image);
      Put_Line ("    (Previous ID => "
                & Prev_Id'Image
                & ")");
   end Adjust;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize:   ID => "
                & E.Id'Image);
   end Finalize;

   procedure Show (E    : T;
                   Name : String) is
   begin
      Put_Line ("Obj. " & Name
                & ": ID => "
```

(continues on next page)

(continued from previous page)

```
                & E.Id'Image);
   end Show;

end Simple_Controlled_Types;
```

Listing 10: show_controlled_types.adb

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

with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Controlled_Types is
   A, B : T;
   --
   --  Declaration corresponds to:
   --
   --  declare
   --     A, B : T;
   --  begin
   --     Initialize (A);
   --     Initialize (B);
   --  end;
begin
   Put_Line ("--------");
   Show (A, "A");
   Show (B, "B");

   Put_Line ("--------");
   Put_Line ("A := B;");

   A := B;
   --  Statement corresponds to:
   --
   --  Finalize (A);
   --  A := B;
   --  Adjust (A);

   Put_Line ("--------");
   Show (A, "A");
   Show (B, "B");
   Put_Line ("--------");

   --  When A and B get out of scope::
   --
   --  Finalize (A);
   --  Finalize (B);
   --
end Show_Controlled_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.Simple_
↪Example_With_Id
MD5: f7b490041616a1b309184086ceef1b24
```

Runtime output

```
Initialize: ID =>  1
Initialize: ID =>  2
--------
```

(continues on next page)

(continued from previous page)

```
Obj. A: ID =>  1
Obj. B: ID =>  2
--------
A := B;
Finalize:   ID =>  1
Adjust:     ID =>  3
    (Previous ID =>  2)
--------
Obj. A: ID =>  3
Obj. B: ID =>  2
--------
Finalize:   ID =>  2
Finalize:   ID =>  3
```

In contrast to the previous versions of the `Simple_Controlled_Types` package, type T now has an `Id` component. Moreover, we use a protected object `Id_Gen` that provides us with a unique ID to keep track of each controlled object. Basically, we assign an ID to each controlled object (right after it is created) via the call to `Initialize`. Similarly, this ID is updated via the calls to `Adjust`. Besides, we now have a Show procedure that displays the ID of a controlled object.

When running the application, we see that the calls to `Initialize`, `Adjust` and `Finalize` happen as expected. In addition, we see the objects' ID, which we will now analyze in order to understand how each object is actually handled.

First, we see the two calls to `Initialize` for objects A and B. Object A's ID is 1, and object B's ID is 2. This is later confirmed by the calls to Show.

The `A := B` assignment triggers two procedure calls: a call to `Finalize (A)` and a call to `Adjust (A)`. In fact, this assignment can be described as follows:

1. `Finalize (A)` is called before the actual copy;
2. B's data is copied to object A;
3. `Adjust (A)` is called after that copy.

We can confirm this via the object ID: the object we handle in the call to `Finalize (A)` has an ID of 1, and the object we handle in the call to `Adjust (A)` has an ID of 2 (which originates from the copy of B to A) and is later changed (*adjusted*) to 3. Again, we can verify the correct IDs by looking at the output of the calls to Show.

Note that the call to `Finalize (A)` (before the copy of B's data) indicates that the previous version of object A is being finalized, i.e. it's as though the original object A is going to be destroyed and its contents are going to be lost. Actually, the object's contents are just overwritten, but the call to `Finalize` allows us to make proper adjustments to the object before the previous information is lost.

Finally, the new version of object A (the one whose ID is 3) and object B are finalized via the calls to `Finalize (A)` and `Finalize (B)` before the `Show_Controlled_Types` procedure ends.

18.2 Initialization

In this section, we cover some details about the initialization of controlled types. Most of those details are related to the initialization order. In principle, as stated in the Ada Reference Manual, "`Initialize` and other initialization operations are done in an arbitrary order," except in the situations that we describe later on.

Relevant topics

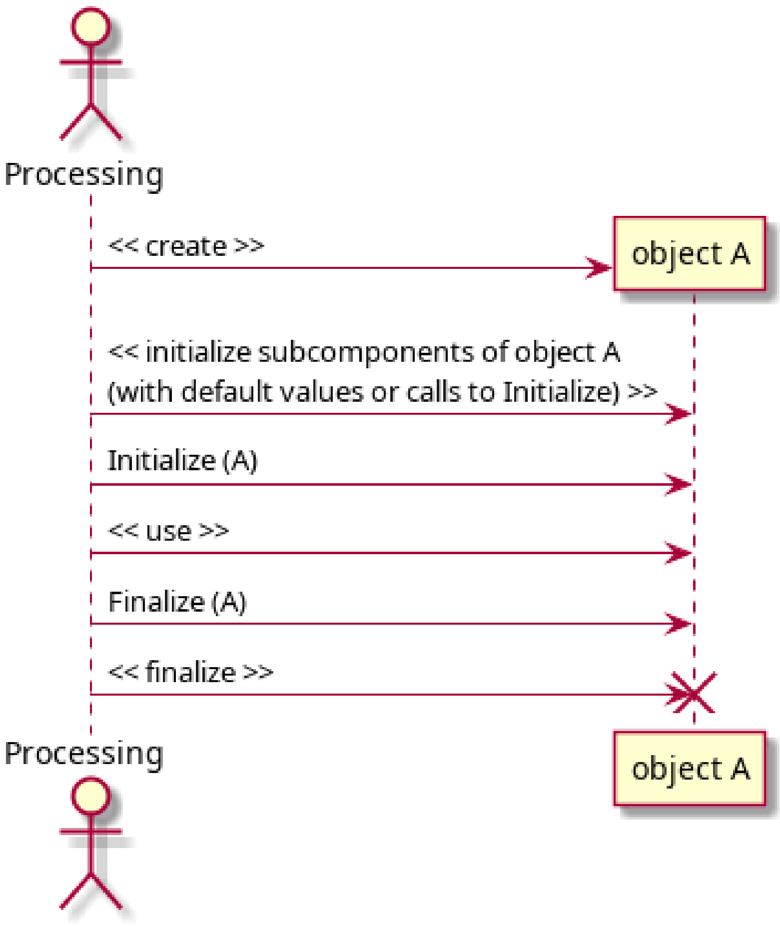
- Assignment and Finalization[337]

18.2.1 Subcomponents

We've seen before that default initialization is a way of controlling the initialization of arbitrary types. In the case of controlled types, the default initialization of its subcomponents always takes places before the call to `Initialize`.

Similarly, a controlled type might have subcomponents of controlled types. These subcomponents are initialized by a call to the `Initialize` procedure of each of those controlled types.

We can visualize the lifetime as follows:



In order to see this effect, let's start by implementing two controlled types: Sub_1 and Sub_2:

Listing 11: subs.ads

```
with Ada.Finalization;

package Subs is

```

(continues on next page)

[337] http://www.ada-auth.org/standards/22rm/html/RM-7-6.html

(continued from previous page)

```
   type Sub_1 is tagged private;

   type Sub_2 is tagged private;

private

   type Sub_1 is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out Sub_1);

   type Sub_2 is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out Sub_2);

end Subs;
```

Listing 12: subs.adb

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

package body Subs is

   procedure Initialize (E : in out Sub_1) is
   begin
      Put_Line ("Initialize: Sub_1...");
   end Initialize;

   procedure Initialize (E : in out Sub_2) is
   begin
      Put_Line ("Initialize: Sub_2...");
   end Initialize;

end Subs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
↪Controlled_Initialization
MD5: f6a7676e82294a62965157d2ffd4ae3b
```

Now, let's use those controlled types as components of a type T. In addition, let's declare an integer component I with default initialization. This is how the complete code looks like:

Listing 13: simple_controlled_types.ads

```
with Ada.Finalization;

with Subs; use Subs;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Dummy (E : T);

```

(continues on next page)

(continued from previous page)

```
private

   function Default_Init return Integer;

   type T is new
     Ada.Finalization.Controlled with
   record
      S1 : Sub_1;
      S2 : Sub_2;
      I  : Integer := Default_Init;
   end record;

   overriding
   procedure Initialize (E : in out T);

end Simple_Controlled_Types;
```

Listing 14: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   function Default_Init return Integer is
   begin
      Put_Line ("Default_Init: Integer...");
      return 42;
   end Default_Init;

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy: T...)");
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize: T...");
   end Initialize;

end Simple_Controlled_Types;
```

Listing 15: show_controlled_types.adb

```
with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Controlled_Types is
   A : T;
begin
   Dummy (A);
end Show_Controlled_Types;
```

Code block metadata

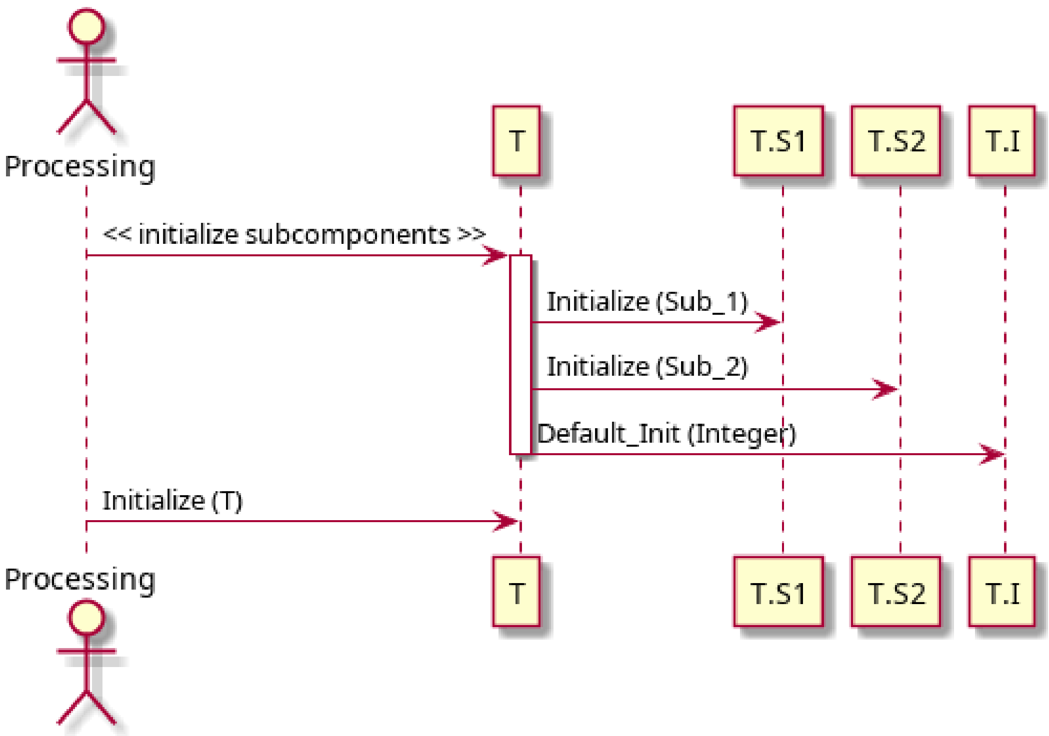
```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
↪Controlled_Initialization
MD5: 39d0efa76c056ac8190573c86f17c890
```

Runtime output

```
Initialize: Sub_1...
Initialize: Sub_2...
Default_Init: Integer...
Initialize: T...
(Dummy: T...)
```

When we run this application, we see that the Sub_1 and Sub_2 components are initialized by calls to their respective `Initialize` procedures, and the `I` component is initialized with its default value (via a call to the `Default_Init` function). Finally, after all subcomponents of type T have been initialized, the `Initialize` procedure is called for the type T itself.

This diagram shows the initialization sequence:



18.2.2 Components with access discriminants

Record types with access discriminants are a special case. In fact, according to the Ada Reference Manual, "if an object has a component with an access discriminant constrained by a *per-object expression* (page 244), `Initialize` is applied to this component after any components that do not have such discriminants. For an object with several components with such a discriminant, `Initialize` is applied to them in order of their component declarations."

Let's see a code example. First, we implement another package with controlled types:

Listing 16: selections.ads

```
with Ada.Finalization;

package Selections is

   type Selection is private;

   type Selection_1 (S : access Selection) is
     tagged private;
```

(continues on next page)

(continued from previous page)

```

   type Selection_2 (S : access Selection) is
     tagged private;

private

   type Selection is null record;

   type Selection_1 (S : access Selection) is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize
     (E : in out Selection_1);

   type Selection_2 (S : access Selection) is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize
     (E : in out Selection_2);

end Selections;
```

Listing 17: selections.adb

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

package body Selections is

   procedure Initialize
     (E : in out Selection_1) is
   begin
      Put_Line ("Initialize: Selection_1...");
   end Initialize;

   procedure Initialize
     (E : in out Selection_2) is
   begin
      Put_Line ("Initialize: Selection_2...");
   end Initialize;

end Selections;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
 ↪Controlled_Initialization
MD5: 01c3639ebd52d37856e77ccfeb057d1b
```

In this example, we see the declaration of the Selection_1 and Selection_2 types, which are controlled types with an access discriminant of Selection type. Now, let's use these types in the declaration of the T type from the *previous example* (page 848) and add two new components (Sel_1 and Sel_2):

Listing 18: simple_controlled_types.ads

```
with Ada.Finalization;
```

(continues on next page)

(continued from previous page)

```

with Subs;       use Subs;
with Selections; use Selections;

package Simple_Controlled_Types is

   type T (S1 : access Selection;
           S2 : access Selection) is
     tagged private;

   procedure Dummy (E : T);

private

   function Default_Init return Integer;

   type T (S1 : access Selection;
           S2 : access Selection) is new
     Ada.Finalization.Controlled with
   record
      Sel_1 : Selection_1 (S1);
      Sel_2 : Selection_2 (S2);
      S_1   : Sub_1;
      I     : Integer := Default_Init;
   end record;

   overriding
   procedure Initialize (E : in out T);

end Simple_Controlled_Types;
```

Listing 19: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   function Default_Init return Integer is
   begin
      Put_Line ("Default_Init: Integer...");
      return 42;
   end Default_Init;

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy: T...)");
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize: T...");
   end Initialize;

end Simple_Controlled_Types;
```

Listing 20: show_controlled_types.adb

```
with Simple_Controlled_Types;
use  Simple_Controlled_Types;

```

(continues on next page)

(continued from previous page)

```
with Selections;
use  Selections;

procedure Show_Controlled_Types is
   S1, S2 : aliased Selection;
   A : T (S1'Access, S2'Access);
begin
   Dummy (A);
end Show_Controlled_Types;
```

Code block metadata

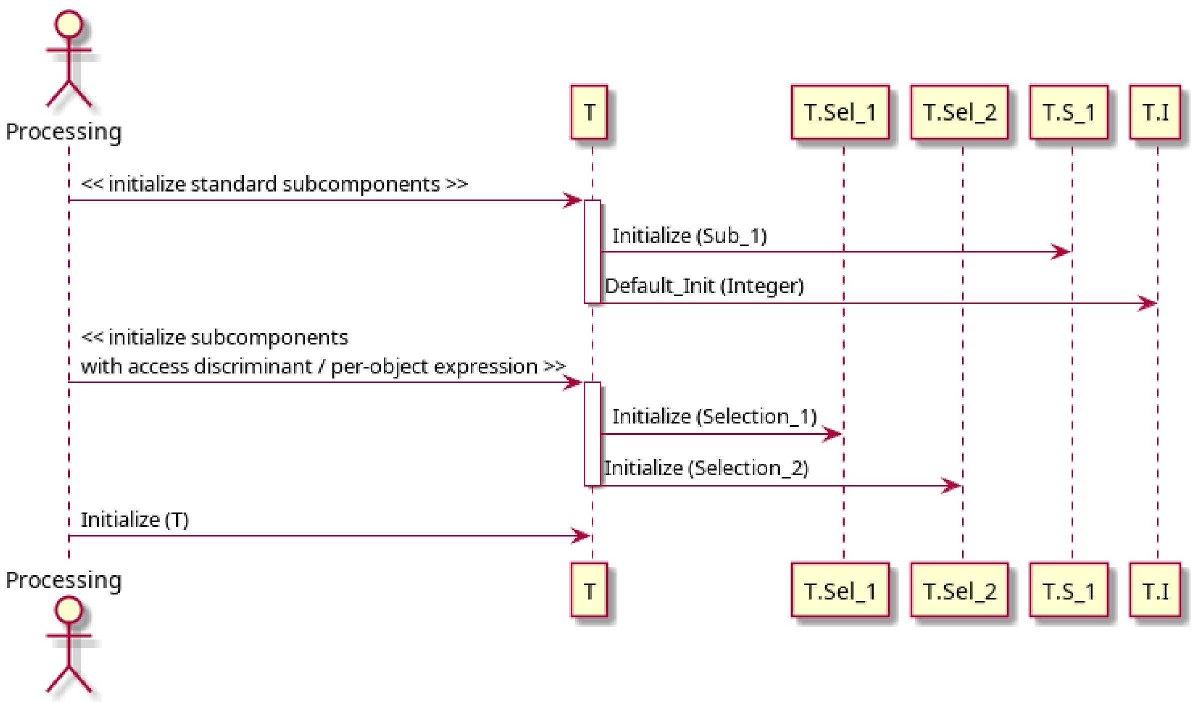
```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
↪Controlled_Initialization
MD5: 74f507b912ab746b70aec451a9bc8f74
```

Runtime output

```
Initialize: Sub_1...
Default_Init: Integer...
Initialize: Selection_1...
Initialize: Selection_2...
Initialize: T...
(Dummy: T...)
```

When running this example, we see that all other subcomponents — to be more precise, those subcomponents that require initialization — are initialized before the Sub_1 and Sub_2 components are initialized via calls to their corresponding `Initialize` procedure. Note that, although Sub_1 and Sub_2 are the last components to be initialized, they are still initialized before the call to the `Initialize` procedure of type T.

This diagram shows the initialization sequence:



18.2.3 Task activation

Components of task types also require special treatment. According to the Ada Reference Manual, "for an allocator, any task activations follow all calls on `Initialize`."

As always, let's analyze an example that illustrates this. First, we implement another package called `Workers` with a simple task type:

Listing 21: workers.ads

```
package Workers is

   task type Worker is
      entry Start;
      entry Stop;
   end Worker;

end Workers;
```

Listing 22: workers.adb

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

package body Workers is

   task body Worker is

      function Init return Integer is
      begin
         Put_Line ("Activating Worker task...");
         return 0;
      end Init;

      I : Integer := Init;
   begin

      accept Start do
        Put_Line ("Worker.Start accepted...");
         I := I + 1;
      end Start;

      accept Stop do
        Put_Line ("Worker.Stop accepted...");
         I := I - 1;
      end Stop;
   end Worker;

end Workers;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
↪Controlled_Initialization
MD5: 1d48a78f14a496c8cdadeab9d1bc9070
```

Let's extend the declaration of the T type from the *previous example* (page 851) and declare a new component of `Worker` type. Note that we have to change T to a limited controlled type because of this new component of task type. This is the updated code:

Listing 23: simple_controlled_types.ads

```
with Ada.Finalization;

with Subs;       use Subs;
with Selections; use Selections;
with Workers;    use Workers;

package Simple_Controlled_Types is

   type T (S : access Selection) is
     tagged limited private;

   procedure Start_Work (E : T);
   procedure Stop_Work (E : T);

private

   function Default_Init return Integer;

   type T (S : access Selection) is new
     Ada.Finalization.Limited_Controlled with
   record
      W     : Worker;
      Sel_1 : Selection_1 (S);
      S1    : Sub_1;
      I     : Integer := Default_Init;
   end record;

   overriding
   procedure Initialize (E : in out T);

end Simple_Controlled_Types;
```

Listing 24: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   function Default_Init return Integer is
   begin
      Put_Line ("Default_Init: Integer...");
      return 42;
   end Default_Init;

   procedure Start_Work (E : T) is
   begin
      --  Starting Worker task:
      E.W.Start;

   end Start_Work;

   procedure Stop_Work (E : T) is
   begin
      --  Stopping Worker task:
      E.W.Stop;
   end Stop_Work;

   procedure Initialize (E : in out T) is
   begin
```

(continues on next page)

(continued from previous page)

```
      Put_Line ("Initialize: T...");
   end Initialize;

end Simple_Controlled_Types;
```

Listing 25: show_controlled_types.adb

```
with Simple_Controlled_Types;
use  Simple_Controlled_Types;

with Selections; use Selections;

procedure Show_Controlled_Types is
   type T_Access is access T;

   S : aliased Selection;
   A : constant T_Access := new T (S'Access);
begin
   Start_Work (A.all);
   Stop_Work (A.all);
end Show_Controlled_Types;
```

Code block metadata

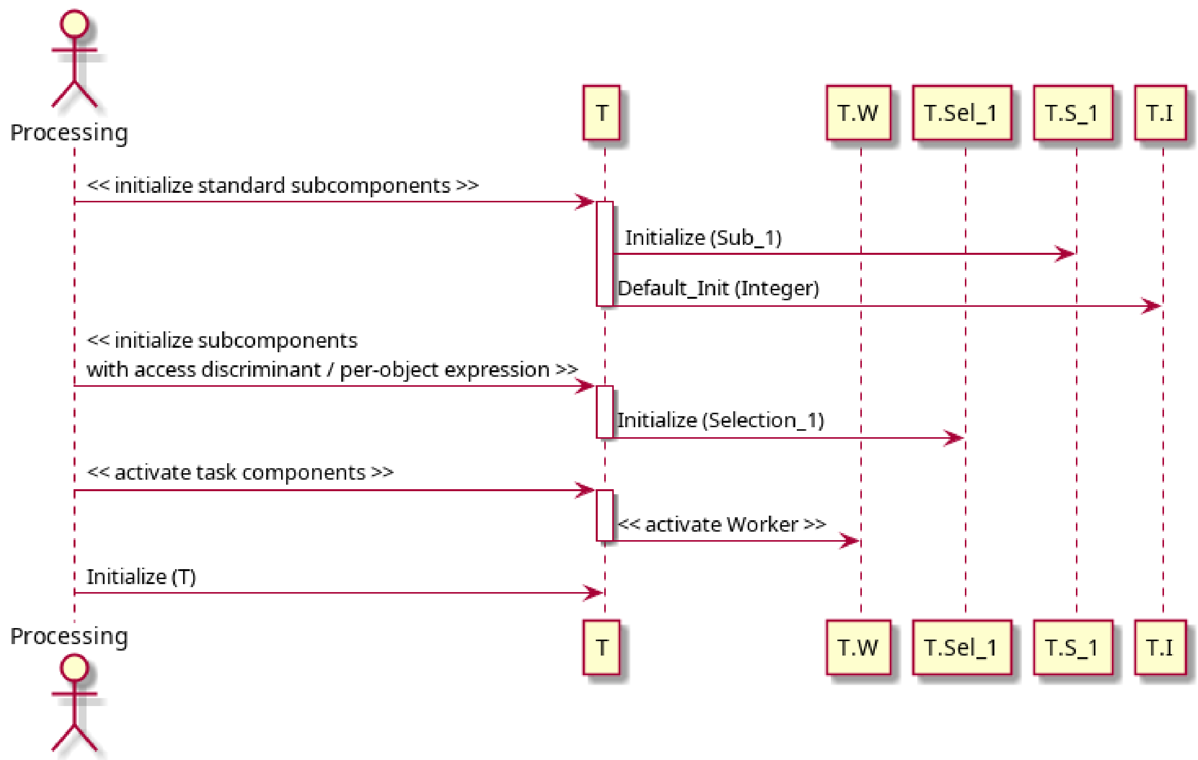
```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
↪Controlled_Initialization
MD5: f87adac74205d590ee66ce971918e642
```

Runtime output

```
Initialize: Sub_1...
Default_Init: Integer...
Initialize: Selection_1...
Activating Worker task...
Initialize: T...
Worker.Start accepted...
Worker.Stop accepted...
```

When we run this application, we see that the W component is activated only after all other subcomponents of type T have been initialized.

This diagram shows the initialization sequence:



18.3 Assignment

We already talked about *adjustments* (page 842) previously. As we already mentioned, an actual assignment is a full bit-wise copy of the entire right-hand side to the entire left-hand side, so the adjustment (via a call to `Adjust`) is a way to "work around" that, when necessary. In this section, we'll look into some details about the adjustment of controlled types.

Relevant topics

- Assignment and Finalization[338]

18.3.1 Assignment using anonymous object

The Ada Reference Manual[339] mentions that an anonymous object is created during the assignment of objects of controlled type. A simple `A := B` operation for nonlimited controlled types can be expanded to the following illustrative code:

```
procedure P is
   A, B: Some_Controlled_Type;
begin
   --
   --  A := B;
   --
   B_To_A_Assignment : declare
      Anon_Obj : Some_Controlled_Type;
   begin
      Anon_Obj := B;
```

(continues on next page)

338 http://www.ada-auth.org/standards/22rm/html/RM-7-6.html
339 http://www.ada-auth.org/standards/22rm/html/RM-7-6.html

(continued from previous page)

```
      Adjust (Anon_Obj);
      Finalize (A);
      A := Anon_Obj;
      Finalize (Anon_Obj);
   end B_To_A_Assignment;
end P;
```

The first assignment happens to the anonymous object Anon_Obj. After the adjustment of Anon_Obj and the finalization of the original version of A, the actual assignment to A can take place — and Anon_Obj can be discarded after it has been properly finalized. With this strategy, we have a chance to finalize the original version of A before the assignment overwrites the object.

Of course, this expanded code isn't really efficient, and the compiler has some freedom to improve the performance of the generated machine code. Whenever possible, it'll typically optimize the anonymous object out and build the object in place. (The Ada Reference Manual[340] describes the rules when this is possible or not.)

Also, the A := Anon_Obj statement in the code above doesn't necessarily translate to an actual assignment in the generated machine code. Typically, a compiler may treat Anon_Obj as the new A and destroy the original version of A (i.e. the object that used to be A). In this case, the code becomes something like this:

```
procedure P is
   A, B: Some_Controlled_Type;
begin
   --
   --  A := B;
   --
   B_To_A_Assignment : declare
      Anon_Obj : Some_Controlled_Type;
   begin
      Anon_Obj := B;
      Finalize (A);
      Adjust (Anon_Obj);
      declare
         A : Some_Controlled_Type renames Anon_Obj;
      begin
         --  Now, we treat Anon_Obj as the new A.
         --  Further processing continues here...

      end;
   end B_To_A_Assignment;
end P;
```

In some cases, the compiler is required to build the object in place. A typical example is when an object of controlled type is initialized by assigning an aggregate to it:

```
C: constant Some_Controlled_Type :=
    (Ada.Finalization.Controlled with ...);
--  C is built in place,
--  no anonymous object is used here.
```

Also, it's possible that Adjust and Finalize aren't called at all. Consider an assignment like this: A := A;. In this case, since the object on both sides is the same, the compiler is allowed to simply skip the assignment and not do anything.

For more details about possible optimizations and compiler behavior, please refer to the Ada Reference Manual[341] .

[340] http://www.ada-auth.org/standards/22rm/html/RM-7-6.html
[341] http://www.ada-auth.org/standards/22rm/html/RM-7-6.html

In general, the advice is simple: use `Adjust` and `Finalize` solely for their intended purposes. In other words, don't implement extraneous side-effects into those procedures, as they might not be called at run-time.

18.3.2 Adjustment of subcomponents

In principle, the order in which components are adjusted is arbitrary. However, adjustments of subcomponents will happen before the adjustment of the component itself. The subcomponents must be adjusted before the enclosing object because the semantics of the adjustment of the whole might depend on the states of the parts (the subcomponents), so those states must already be in place.

Let's revisit a *previous code example* (page 848). First, we override the `Adjust` procedure of the Sub_1 and Sub_2 types from the Subs package.

Listing 26: subs.ads

```
with Ada.Finalization;

package Subs is

   type Sub_1 is tagged private;

   type Sub_2 is tagged private;

private

   type Sub_1 is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out Sub_1);

   overriding
   procedure Adjust (E : in out Sub_1);

   overriding
   procedure Finalize (E : in out Sub_1);

   type Sub_2 is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out Sub_2);

   overriding
   procedure Adjust (E : in out Sub_2);

   overriding
   procedure Finalize (E : in out Sub_2);

end Subs;
```

Listing 27: subs.adb

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

package body Subs is

```

(continues on next page)

(continued from previous page)

```
   procedure Initialize (E : in out Sub_1) is
   begin
      Put_Line ("Initialize: Sub_1...");
   end Initialize;

   procedure Adjust (E : in out Sub_1) is
   begin
      Put_Line ("Adjust: Sub_1...");
   end Adjust;

   procedure Finalize (E : in out Sub_1) is
   begin
      Put_Line ("Finalize: Sub_1...");
   end Finalize;

   procedure Initialize (E : in out Sub_2) is
   begin
      Put_Line ("Initialize: Sub_2...");
   end Initialize;

   procedure Adjust (E : in out Sub_2) is
   begin
      Put_Line ("Adjust: Sub_2...");
   end Adjust;

   procedure Finalize (E : in out Sub_2) is
   begin
      Put_Line ("Finalize: Sub_2...");
   end Finalize;

end Subs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Adjustment.
↪Controlled_Initialization
MD5: 110d88543a7a897ba433c90f6c2a881c
```

Next, we override the `Adjust` procedure of the T type from the `Simple_Controlled_Types` package:

Listing 28: simple_controlled_types.ads

```
with Ada.Finalization;

with Subs; use Subs;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Dummy (E : T);

private

   function Default_Init return Integer;

   type T is new
     Ada.Finalization.Controlled with
   record
      S1 : Sub_1;
```

(continues on next page)

(continued from previous page)

```
      S2 : Sub_2;
      I  : Integer := Default_Init;
   end record;

   overriding
   procedure Initialize (E : in out T);

   overriding
   procedure Adjust (E : in out T);

   overriding
   procedure Finalize (E : in out T);

end Simple_Controlled_Types;
```

Listing 29: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   function Default_Init return Integer is
   begin
      Put_Line ("Default_Init: Integer...");
      return 42;
   end Default_Init;

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy: T...)");
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize: T...");
   end Initialize;

   procedure Adjust (E : in out T) is
   begin
      Put_Line ("Adjust: T...");
   end Adjust;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize: T...");
   end Finalize;

end Simple_Controlled_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Adjustment.
 ↪Controlled_Initialization
MD5: 9fb392305df70734994cffe612cb3869
```

Finally, this is the main application:

Listing 30: show_controlled_types.adb

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

```

(continues on next page)

(continued from previous page)

```
with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Controlled_Types is
   A, B : T;
begin
   Dummy (A);

   Put_Line ("----------");
   Put_Line ("A := B");
   A := B;
   Put_Line ("----------");
end Show_Controlled_Types;
```

Code block metadata

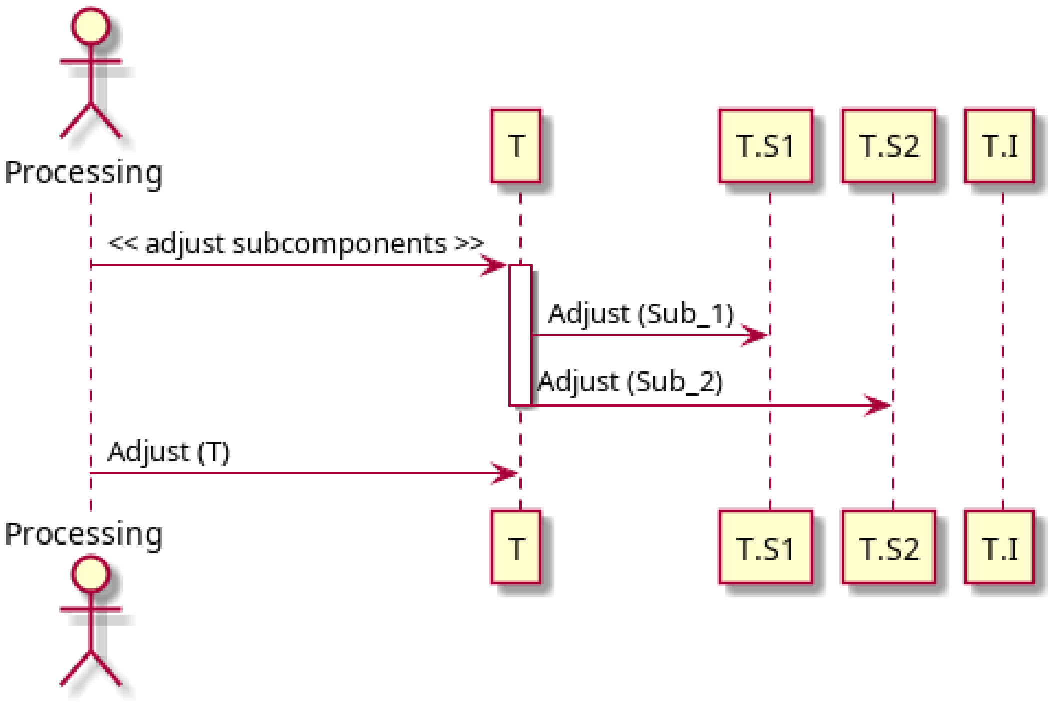
```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Adjustment.
↪Controlled_Initialization
MD5: 1ceaa50cbb18b9f1f997246a614e3a90
```

Runtime output

```
Initialize: Sub_1...
Initialize: Sub_2...
Default_Init: Integer...
Initialize: T...
Initialize: Sub_1...
Initialize: Sub_2...
Default_Init: Integer...
Initialize: T...
(Dummy: T...)
----------
A := B
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
Adjust: Sub_1...
Adjust: Sub_2...
Adjust: T...
----------
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
```

When running this code, we see that the S1 and S2 components are adjusted before the adjustment of the parent type T takes place.

This diagram shows the adjustment sequence:



18.4 Finalization

We mentioned finalization — and the `Finalize` procedure — at the *beginning of the chapter* (page 838). In this section, we discuss the topic in more detail.

Relevant topics

- Assignment and Finalization[342]
- Completion and Finalization[343]

18.4.1 Normal and abnormal completion

When a subprogram has just executed its last statement, normal completion of this subprogram has been reached. At this point, finalization starts. In the case of controlled objects, this means that the `Finalize` procedure is called for those objects. (As we've already seen *an example of normal completion* (page 840) at the beginning of the chapter, we won't repeat it here, as we assume you are already familiar with the concept.)

When an exception is raised or due to an abort, however, a subprogram has an abnormal completion. We discuss more about exception handling and finalization *later on* (page 873).

18.4.2 Finalization via unchecked deallocation

When performing unchecked deallocation of a controlled type, the `Finalize` procedure is called right before the actual memory for the controlled object is deallocated.

Let's see a simple example:

342 http://www.ada-auth.org/standards/22rm/html/RM-7-6.html
343 http://www.ada-auth.org/standards/22rm/html/RM-7-6-1.html

Listing 31: simple_controlled_types.ads

```
with Ada.Finalization;
with Ada.Unchecked_Deallocation;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Dummy (E : T);

   type T_Access is access T;

   procedure Free (A : in out T_Access);

private

   type T is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Finalize (E : in out T);

   procedure Free_T_Access is
     new Ada.Unchecked_Deallocation
       (Object => T,
        Name   => T_Access);

   procedure Free (A : in out T_Access)
     renames Free_T_Access;

end Simple_Controlled_Types;
```

Listing 32: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy T...)");
   end Dummy;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize T...");
   end Finalize;

end Simple_Controlled_Types;
```

Listing 33: show_controlled_types.adb

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

with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Controlled_Types is
   A : T_Access := new T;
```

(continues on next page)

(continued from previous page)

```
begin
   Dummy (A.all);

   Free (A);
   --  At this point, Finalize (A.all)
   --  will be called before the actual
   --  deallocation.

   Put_Line ("We've just freed A.");
end Show_Controlled_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↪Unchecked_Deallocation
MD5: b9388699ee396430f689fe88df41fc32
```

Runtime output

```
(Dummy T...)
Finalize T...
We've just freed A.
```

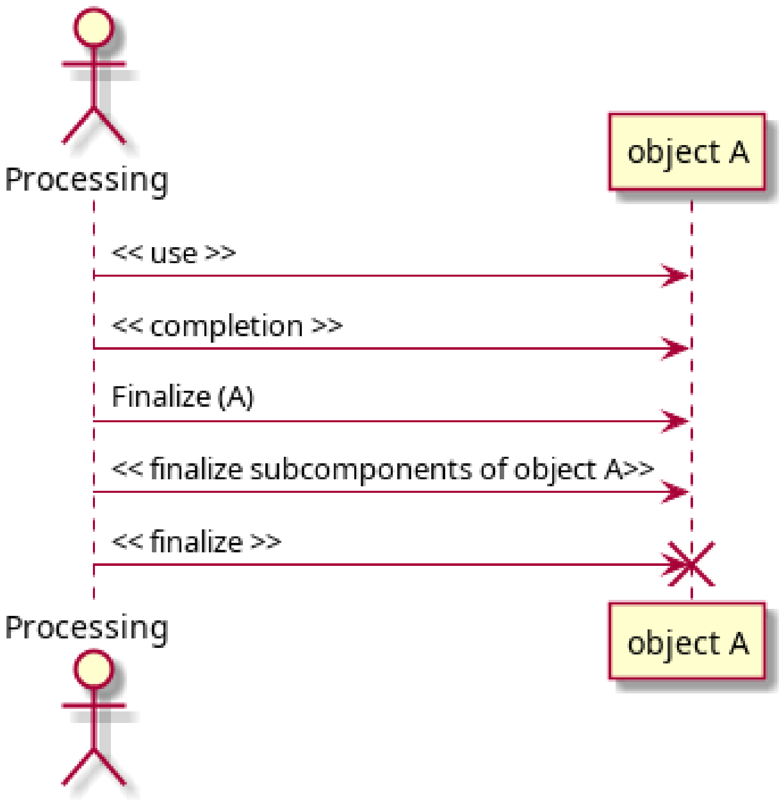
In this example, we see that a call to `Finalize` (for type `T`) is triggered by the call to `Free` for the `A` object — at this point, we haven't reached the end of the main procedure (`Show_Controlled_Types`) yet. After the call to `Free`, the object originally referenced by `A` has been completely finalized — and deallocated.

When the main procedure completes (after the call to `Put_Line` in that procedure), we would normally see the calls to `Finalize` for controlled objects. However, at this point, we obviously don't have a second call to the `Finalize` procedure for type `T`, as the object referenced by `A` has already been finalized and freed.

18.4.3 Subcomponents

As we've seen in the section about *initialization of subcomponents* (page 848), subcomponents of a controlled type are initialized by a call to their corresponding `Initialize` procedure before the call to `Initialize` for the parent controlled type. In the case of finalization, the reverse order is applied: first, finalization of the parent type takes place, and then the finalization of the subcomponents.

We can visualize the lifetime as follows:



Let's show a code example by revisiting the previous implementation of the controlled types Sub_1 and Sub_2, and adapting it:

Listing 34: subs.ads

```
with Ada.Finalization;

package Subs is

   type Sub_1 is tagged private;

   type Sub_2 is tagged private;

private

   type Sub_1 is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Finalize (E : in out Sub_1);

   type Sub_2 is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Finalize (E : in out Sub_2);

end Subs;
```

Listing 35: subs.adb

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

package body Subs is

   procedure Finalize (E : in out Sub_1) is
   begin
      Put_Line ("Finalize: Sub_1...");
   end Finalize;

   procedure Finalize (E : in out Sub_2) is
   begin
      Put_Line ("Finalize: Sub_2...");
   end Finalize;

end Subs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↪Controlled_Initialization
MD5: 565f0b13586c08e0cdfdc119bcb28780
```

Now, let's use those controlled types as components of a type T:

Listing 36: simple_controlled_types.ads

```
with Ada.Finalization;

with Subs; use Subs;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Dummy (E : T);

private

   type T is new
     Ada.Finalization.Controlled with
   record
      S1 : Sub_1;
      S2 : Sub_2;
   end record;

   overriding
   procedure Finalize (E : in out T);

end Simple_Controlled_Types;
```

Listing 37: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy: T...)");
```

(continues on next page)

(continued from previous page)

```
   end Dummy;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize: T...");
   end Finalize;

end Simple_Controlled_Types;
```

Listing 38: show_controlled_types.adb

```
with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Controlled_Types is
   A : T;
begin
   Dummy (A);
end Show_Controlled_Types;
```

Code block metadata

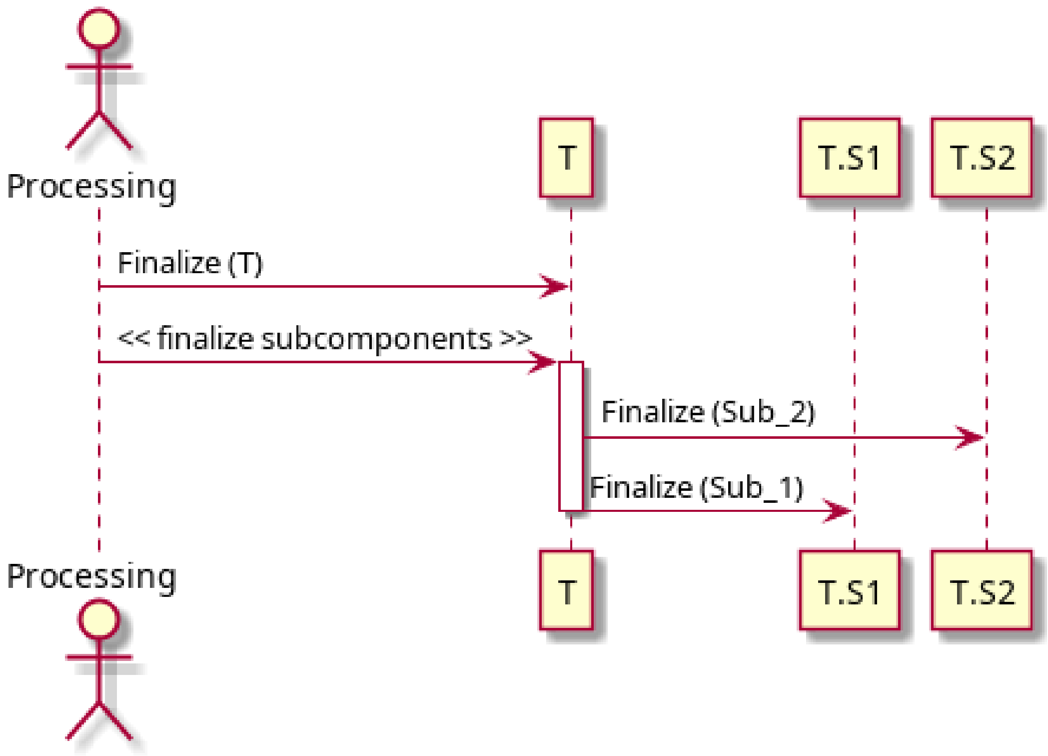
```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↪Controlled_Initialization
MD5: 6feecb7c544f340bf4841034d7ab5f71
```

Runtime output

```
(Dummy: T...)
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
```

When we run this application, we see that the `Finalize` procedure is called for the type T itself — as the first step of the finalization of type T. Then, the Sub_2 and Sub_1 components are finalized by calls to their respective `Finalize` procedures.

This diagram shows the finalization sequence:



18.4.4 Components with access discriminants

We already discussed the *initialization of components with access discriminants constrained by a per-object expression* (page 851). In the case of the finalization of such components, they are finalized before any components that do not fall into this category — in the reverse order of their component declarations — but after the finalization of the parent type.

Let's revisit a *previous code example* (page 851) and adapt it to demonstrate the finalization of components with access discriminants. First, we implement another package with controlled types:

Listing 39: selections.ads

```
with Ada.Finalization;

package Selections is

   type Selection is private;

   type Selection_1 (S : access Selection) is
     tagged private;

   type Selection_2 (S : access Selection) is
     tagged private;

private

   type Selection is null record;

   type Selection_1 (S : access Selection) is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Finalize
```

(continues on next page)

(continued from previous page)

```
      (E : in out Selection_1);

   type Selection_2 (S : access Selection) is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Finalize
     (E : in out Selection_2);

end Selections;
```

Listing 40: selections.adb

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

package body Selections is

   procedure Finalize
     (E : in out Selection_1) is
   begin
      Put_Line ("Finalize: Selection_1...");
   end Finalize;

   procedure Finalize
     (E : in out Selection_2) is
   begin
      Put_Line ("Finalize: Selection_2...");
   end Finalize;

end Selections;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↪Controlled_Initialization
MD5: d1d35eb7ea62742fb130fbf05d898989
```

In this example, we see the declaration of the `Selection_1` and `Selection_2` types, which are controlled types with an access discriminant of `Selection` type. Now, let's use these types in the declaration of a type T and add two new components — `Sel_1` and `Sel_2`:

Listing 41: simple_controlled_types.ads

```
with Ada.Finalization;

with Subs;       use Subs;
with Selections; use Selections;

package Simple_Controlled_Types is

   type T (S1 : access Selection;
           S2 : access Selection) is
     tagged private;

   procedure Dummy (E : T);

private

   type T (S1 : access Selection;
```

(continues on next page)

(continued from previous page)

```
           S2 : access Selection) is new
     Ada.Finalization.Controlled with
   record
      Sel_1 : Selection_1 (S1);
      Sel_2 : Selection_2 (S2);
      S_1   : Sub_1;
   end record;

   overriding
   procedure Finalize (E : in out T);

end Simple_Controlled_Types;
```

Listing 42: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy: T...)");
   end Dummy;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize: T...");
   end Finalize;

end Simple_Controlled_Types;
```

Listing 43: show_controlled_types.adb

```
with Simple_Controlled_Types;
use  Simple_Controlled_Types;

with Selections;
use  Selections;

procedure Show_Controlled_Types is
   S1, S2 : aliased Selection;
   A : T (S1'Access, S2'Access);
begin
   Dummy (A);
end Show_Controlled_Types;
```

Code block metadata

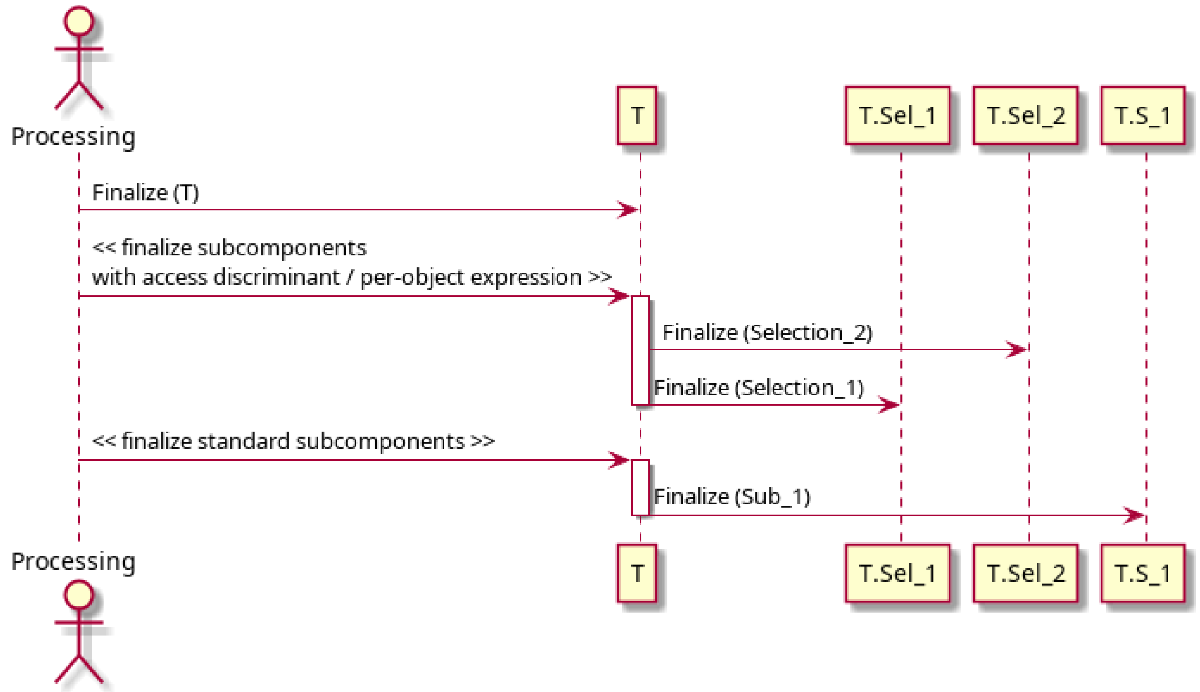
```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↪Controlled_Initialization
MD5: e421a750f11ade3b4df98569c71b904a
```

Runtime output

```
(Dummy: T...)
Finalize: T...
Finalize: Selection_2...
Finalize: Selection_1...
Finalize: Sub_1...
```

When we run this example, we see that the Finalize procedure of type T is called as the first step. Then, the `Finalize` procedure is called for the components with an access discriminant constrained by a *per-object expression* (page 244) — in this case, `Sel_2` and `Sel_1` (of `Selection_2` and `Selection_1` types, respectively). Finally, the `Sub_1` component is finalized.

This diagram shows the finalization sequence:



18.5 Controlled Types and Exception Handling

In the previous section, we mainly focused on the normal completion of controlled types. However, when control is transferred out of the normal execution path due to an abort or an exception being raised, we speak of abnormal completion. In this section, we focus on those cases.

Let's start with a simple example:

Listing 44: simple_controlled_types.ads

```
with Ada.Finalization;

package Simple_Controlled_Types is

   type T is tagged private;

   procedure Dummy (E : T);

private

   type T is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out T);

   overriding
```

(continues on next page)

(continued from previous page)

```
   procedure Finalize (E : in out T);

end Simple_Controlled_Types;
```

Listing 45: simple_controlled_types.adb

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

package body Simple_Controlled_Types is

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy...)");
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize...");
   end Initialize;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize...");
   end Finalize;

end Simple_Controlled_Types;
```

Listing 46: show_simple_exception.adb

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

with Simple_Controlled_Types;
use  Simple_Controlled_Types;

procedure Show_Simple_Exception is
   A : T;

   function Int_Last return Integer is
     (Integer'Last);

   Cnt : Positive := Int_Last;
begin
   Cnt := Cnt + 1;

   Dummy (A);

   Put_Line (Cnt'Image);

   --  When A is about to get out of
   --  scope:
   --
   --  Finalize (A);
   --
end Show_Simple_Exception;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_
↪Handling.Simple_Exception
MD5: 9461f420f091f058e6ea1ee419b2a5c6
```

Runtime output

```
Initialize...
Finalize...

raised CONSTRAINT_ERROR : show_simple_exception.adb:14 overflow check failed
```

In this example, we're forcing an overflow to happen in the `Show_Simple_Exception` by adding one to the integer variable `Cnt`, which already has the value `Integer'Last`. The corresponding *overflow check* (page 519) raises the `Constraint_Error`.

However, *before* this exception is raised, the finalization of the controlled object `A` is performed. In this sense, we have normal completion of the controlled type — even though an exception is being raised.

For further reading...

We already talked about the *allocation check* (page 523), which may raise a `Program_Error` exception. In the code example for that section, we used controlled types. Feel free to revisit the example.

Relevant topics

- Completion and Finalization[344]

18.5.1 Exception raising in Initialize

If an exception is raised in the `Initialize` procedure, we have abnormal completion. Let's see an example:

Listing 47: ct_initialize_exception.ads

```
with Ada.Finalization;

package CT_Initialize_Exception is

   type T is tagged private;

   procedure Dummy (E : T);

private

   type T is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out T);

   overriding
   procedure Finalize (E : in out T);

end CT_Initialize_Exception;
```

[344] http://www.ada-auth.org/standards/22rm/html/RM-7-6-1.html

Listing 48: ct_initialize_exception.adb

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

package body CT_Initialize_Exception is

   function Int_Last return Integer is
     (Integer'Last);

   Cnt : Positive := Int_Last;

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy...)");
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize...");
      Cnt := Cnt + 1;
   end Initialize;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize...");
   end Finalize;

end CT_Initialize_Exception;
```

Listing 49: show_initialize_exception.adb

```
with CT_Initialize_Exception;
use  CT_Initialize_Exception;

procedure Show_Initialize_Exception is
   A : T;
begin
   Dummy (A);
end Show_Initialize_Exception;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_
 ↪Handling.CT_Initialize_Exception
MD5: 189a5fafafb01eba31a73c9237fa7aff
```

Runtime output

```
Initialize...

raised CONSTRAINT_ERROR : ct_initialize_exception.adb:18 overflow check failed
```

In the Show_Initialize_Exception procedure, we declare an object A of controlled type T. As we know, this declaration triggers a call to the Initialize procedure that we've implemented in the body of the CT_Initialize_Exception package. In the Initialize procedure, we're forcing an overflow to happen — by adding one to the Cnt variable, which already has the **Integer**'Last value.

This is an example of abnormal completion, as the control is transferred out of the Initialize procedure, and the corresponding Finalize procedure is never called for object A.

18.5.2 Bounded errors of controlled types

Bounded errors (page 506) are an important topic when talking about exception and controlled types. In general, if an exception is raised in the Adjust or Finalize procedure, this is considered a bounded error. If the bounded error is detected, the Program_Error exception is raised.

Note that the original exception raised in the Adjust or Finalize procedures could be any possible exception. For example, one of those procedures could raise a Constraint_Error exception. However, the actual exception that is raised at runtime is the Program_Error exception. This is because the bounded error, which raises the Program_Error exception, is more severe than the original exception coming from those procedures.

(The behavior is different when the Adjust or Finalize procedure is called explicitly, as we'll see later.)

Not every exception raised during an operation on controlled types is considered a bounded error. In fact, the case we've seen before, an *exception raised in the Initialize procedure* (page 875) is not a bounded error.

Here's a code example of a Constraint_Error exception being raised in the Finalize procedure:

Listing 50: ct_finalize_exception.ads

```
with Ada.Finalization;

package CT_Finalize_Exception is

   type T is tagged private;

   procedure Dummy (E : T);

   procedure Reset_Counter;

private

   type T is new
     Ada.Finalization.Controlled
       with null record;

   overriding
   procedure Initialize (E : in out T);

   overriding
   procedure Adjust (E : in out T);

   overriding
   procedure Finalize (E : in out T);

end CT_Finalize_Exception;
```

Listing 51: ct_finalize_exception.adb

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

package body CT_Finalize_Exception is

   Cnt : Integer := Integer'Last;

   procedure Dummy (E : T) is
   begin
      Put_Line ("(Dummy...)");
```

(continues on next page)

(continued from previous page)

```
   end Dummy;

   procedure Initialize (E : in out T) is
   begin
      Put_Line ("Initialize...");
   end Initialize;

   overriding
   procedure Adjust (E : in out T) is
   begin
      Put_Line ("Adjust...");
   end Adjust;

   procedure Finalize (E : in out T) is
   begin
      Put_Line ("Finalize...");
      Cnt := Cnt + 1;
   end Finalize;

   procedure Reset_Counter is
   begin
      Cnt := 0;
   end Reset_Counter;

end CT_Finalize_Exception;
```

Listing 52: show_finalize_exception.adb

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

with CT_Finalize_Exception;
use  CT_Finalize_Exception;

procedure Show_Finalize_Exception is
   A, B : T;
begin
   Dummy (A);

   --  When A is about to get out of
   --  scope:
   --
   --  Finalize (A);
   --
end Show_Finalize_Exception;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_
↪Handling.CT_Finalize_Exception
MD5: eacb64b0a9d68ce3484a3bda9b633495
```

Runtime output

```
Initialize...
Initialize...
(Dummy...)
Finalize...
Finalize...

raised PROGRAM_ERROR : show_finalize_exception.adb:6 finalize/adjust raised␣
↪exception
```

In this example, we're again forcing an overflow to happen (by adding one to the integer variable `Cnt`), this time in the `Finalize` procedure. When this procedure is implicitly called — when object `A` is about to get out of scope in the `Show_Finalize_Exception` procedure — the `Constraint_Error` exception is raised.

As we've just seen, having an exception be raised during an implicit call to the `Finalize` procedure is a bounded error. Therefore, we see that the `Program_Error` exception is raised at runtime instead of the original `Constraint_Error` exception.

As we hinted in the beginning, when the `Adjust` or the `Finalize` procedure is called *explicitly*, the exception raised in that procedure is *not* considered a bounded error. In this case, the original exception is raised.

To show an example of such an explicit call, let's first move the overriden procedures for type `T` (`Initialize`, `Adjust` and `Finalize`) out of the private part of the package `CT_Finalize_Exception`, so they are now visible to clients. This allows us to call the `Finalize` procedure explicitly:

Listing 53: ct_finalize_exception.ads

```
with Ada.Finalization;

package CT_Finalize_Exception is

   type T is new
     Ada.Finalization.Controlled
   with null record;

   overriding
   procedure Initialize (E : in out T);

   overriding
   procedure Adjust (E : in out T);

   overriding
   procedure Finalize (E : in out T);

   procedure Dummy (E : T);

   procedure Reset_Counter;

end CT_Finalize_Exception;
```

Listing 54: show_finalize_exception.adb

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

with CT_Finalize_Exception;
use  CT_Finalize_Exception;

procedure Show_Finalize_Exception is
   A : T;
begin
   Dummy (A);

   Finalize (A);

   Put_Line ("After Finalize");
exception
   when Constraint_Error =>
      Put_Line
        ("Constraint_Error is being handled...");
```

(continues on next page)

(continued from previous page)

```
      Reset_Counter;
end Show_Finalize_Exception;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_
↪Handling.CT_Finalize_Exception
MD5: f43133c997076c491a20117960be8807
```

Runtime output

```
Initialize...
(Dummy...)
Finalize...
Constraint_Error is being handled...
Finalize...
```

Now, we're calling the `Finalize` procedure explicitly in the `Show_Finalize_Exception` procedure. As we know, due to the operation on `I` in the `Finalize` procedure, the `Constraint_Error` exception is raised in the procedure. Because we're handling this exception in the `Show_Finalize_Exception` procedure, we see the corresponding user message ("Constraint_Error is being handled...") at runtime.

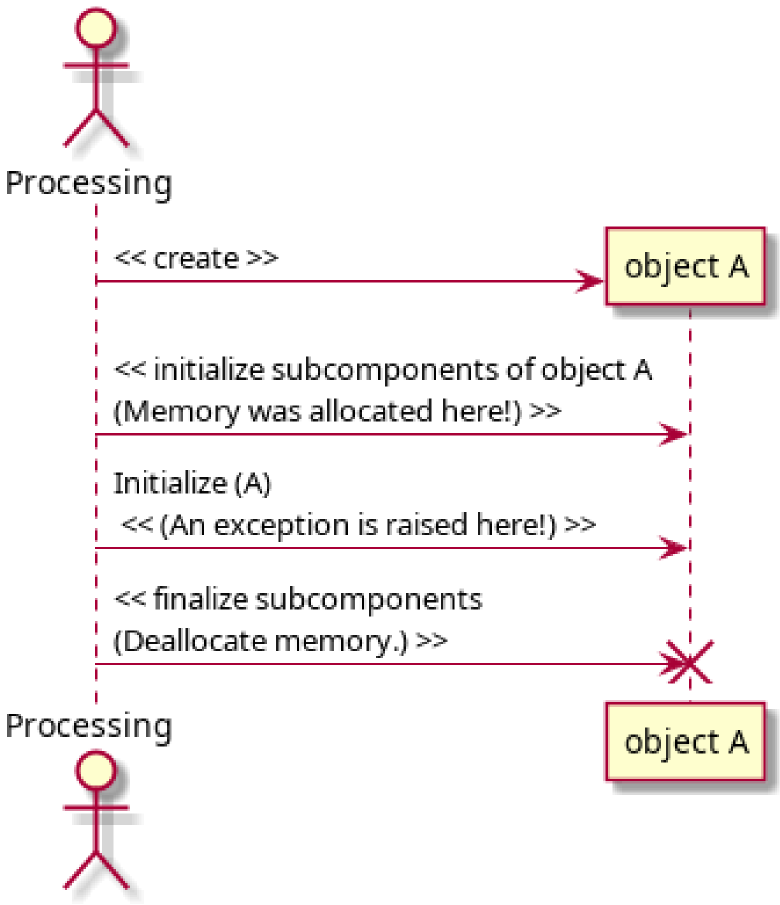
(Note that in the exception handling block, we're calling the `Reset_Counter` procedure. This prevents `Constraint_Error` from being raised in the next call to `Finalize`.)

18.5.3 Memory allocation and exceptions

When a memory block is allocated for controlled types and a bounded error occurs, there is no guarantee that this memory block will be deallocated. Roughly speaking, the compiler has the freedom — but not the obligation — to generate appropriate calls to `Finalize`, which may deallocate memory blocks.

For example, we've seen that *subcomponents of controlled type* (page 848) of a controlled object `A` are initialized before the initialization of object `A` takes place. Because memory might have been allocated for the subcomponents, the compiler can insert code that attempts to finalize those subcomponents, which in turn deallocates the memory blocks (if they were allocated in the first place).

We can visualize this strategy in the following diagram:



This strategy (of finalizing subcomponents that haven't raised exceptions) prevents memory leaks. However, this behavior very much depends on the compiler implementation. The Ada Reference Manual[345] delineates (in the "Implementation Permissions" section) the cases where the compiler is allowed — but not required — to finalize objects when exceptions are raised.

Because the actual behavior isn't defined, custom implementation of `Adjust` and `Finalize` procedures for controlled types should be designed very carefully in order to avoid exceptions, especially when memory is allocated in the `Initialize` procedure.

18.6 Applications of Controlled Types

In this section, we discuss applications of controlled types. In this context, it's important to remember that controlled types have an associated overhead, which can become non-negligible depending in which context the controlled objects are used. However, there are applications where utilizing controlled types is the best approach.

(Note that this overhead we've just mentioned is not specific to Ada. In fact, types similar to controlled types will be relatively expensive in any programming language. As an example, destructors in C++ may require a similar maintenance of state at run-time.)

[345] http://www.ada-auth.org/standards/22rm/html/RM-7-6-1.html

18.6.1 Encapsulating access type handling

Previously, when discussing *design strategies for access types* (page 669), we saw an example on using *limited controlled types to encapsulate access types* (page 672).

A more generalized example is the one of an unbounded stack. Because it's unbounded, it allows for increasing the stack's size *on demand*. We can implement this kind of stack by using access types. Let's look at a simple (unoptimized) implementation:

Listing 55: unbounded_stacks.ads

```
with Ada.Finalization;

generic
   Default_Chunk_Size : Positive := 5;
   type Element is private;
package Unbounded_Stacks is

   Stack_Underflow : exception;

   type Unbounded_Stack is private;

   procedure Push (S : in out Unbounded_Stack;
                   E :        Element);

   function Pop (S : in out Unbounded_Stack)
                 return Element;

   function Is_Empty (S : Unbounded_Stack)
                      return Boolean;

private

   type Element_Array is
     array (Positive range <>) of
       Element;

   type Element_Array_Access is
     access Element_Array;

   type Unbounded_Stack is new
     Ada.Finalization.Controlled with
      record
         Chunk_Size : Positive
           := Default_Chunk_Size;
         Data       : Element_Array_Access;
         Top        : Natural := 0;
      end record;

   procedure Initialize
     (S : in out Unbounded_Stack);

   procedure Adjust
     (S : in out Unbounded_Stack);

   procedure Finalize
     (S : in out Unbounded_Stack);

end Unbounded_Stacks;
```

Listing 56: unbounded_stacks.adb

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

with Ada.Unchecked_Deallocation;

package body Unbounded_Stacks is

   --
   --  LOCAL SUBPROGRAMS
   --

   procedure Free is
     new Ada.Unchecked_Deallocation
       (Object => Element_Array,
        Name   => Element_Array_Access);

   function Is_Full (S : Unbounded_Stack)
                     return Boolean is
   begin
      return S.Top = S.Data'Last;
   end Is_Full;

   procedure Reallocate_Data
     (To         : in out Element_Array_Access;
      From       :        Element_Array_Access;
      Max_Last   :        Positive;
      Valid_Last :        Positive) is
   begin
      To := new Element_Array (1 .. Max_Last);

      for I in 1 .. Valid_Last loop
         To (I) := From (I);
      end loop;
   end Reallocate_Data;

   procedure Increase_Size
     (S : in out Unbounded_Stack)
   is
      Old_Data : Element_Array_Access := S.Data;
      Old_Last : constant Positive
                 := Old_Data'Last;
      New_Last : constant Positive
                 := Old_Data'Last + S.Chunk_Size;
   begin
      Put_Line ("Increasing Unbounded_Stack "
                & "(1 .. "
                & Old_Last'Image
                & ") to (1 .. "
                & New_Last'Image
                & ")");

      Reallocate_Data
        (To         => S.Data,
         From       => Old_Data,
         Max_Last   => New_Last,
         Valid_Last => S.Top);

      Free (Old_Data);
   end Increase_Size;

```

(continues on next page)

(continued from previous page)

```
   --
   --  SUBPROGRAMS
   --

   procedure Push (S : in out Unbounded_Stack;
                   E :        Element) is
   begin
      if Is_Full (S) then
         Increase_Size (S);
      end if;

      S.Top := S.Top + 1;
      S.Data (S.Top) := E;
   end Push;

   function Pop (S : in out Unbounded_Stack)
                 return Element is
   begin
      return E : Element do
         if Is_Empty (S) then
            raise Stack_Underflow;
         end if;

         E := S.Data (S.Top);
         S.Top := S.Top - 1;
      end return;
   end Pop;

   function Is_Empty (S : Unbounded_Stack)
                      return Boolean is
   begin
      return S.Top = 0;
   end Is_Empty;

   --
   --  PRIVATE SUBPROGRAMS
   --

   procedure Initialize
     (S : in out Unbounded_Stack)
   is
      Last : constant Positive
               := S.Chunk_Size;
   begin
      Put_Line ("Initializing Unbounded_Stack "
                & "(1 .. "
                & Last'Image
                & ")");
      S.Data := new Element_Array
                      (1 .. S.Chunk_Size);
   end Initialize;

   procedure Allocate_Duplicate_Data
     (S : in out Unbounded_Stack)
   is
      Last : constant Positive
               := S.Data'Last;
   begin
      Put_Line ("Duplicating data for new "
                & "Unbounded_Stack (1 .. "
                & Last'Image
```

(continues on next page)

(continued from previous page)

```
               & ")");

      Reallocate_Data
        (To         => S.Data,
         From       => S.Data,
         Max_Last   => Last,
         Valid_Last => S.Top);
   end Allocate_Duplicate_Data;

   procedure Adjust
     (S : in out Unbounded_Stack)
   is
   begin
      Put_Line ("Adjusting Unbounded_Stack...");
      Allocate_Duplicate_Data (S);
   end Adjust;

   procedure Finalize
     (S : in out Unbounded_Stack)
   is
      Last : constant Positive
             := S.Data'Last;
   begin
      Put_Line ("Finalizing Unbounded_Stack "
                & "(1 .. "
                & Last'Image
                & ")");
      if S.Data /= null then
        Free (S.Data);
      end if;
   end Finalize;

end Unbounded_Stacks;
```

Listing 57: show_unbounded_stack.adb

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

with Unbounded_Stacks;

procedure Show_Unbounded_Stack is

   package Unbounded_Integer_Stacks is new
     Unbounded_Stacks (Element => Integer);
   use Unbounded_Integer_Stacks;

   procedure Print_Pop_Stack
      (S    : in out Unbounded_Stack;
       Name :        String)
   is
      V : Integer;
   begin
      Put_Line ("STACK: " & Name);
      Put ("= ");
      while not Is_Empty (S) loop
         V := Pop (S);
         Put (V'Image & " ");
      end loop;
      New_Line;
   end Print_Pop_Stack;

```

(continues on next page)

(continued from previous page)

```
   Stack   : Unbounded_Stack;
   Stack_2 : Unbounded_Stack;
begin
   for I in 1 .. 10 loop
      Push (Stack, I);
   end loop;

   Stack_2 := Stack;

   for I in 11 .. 20 loop
      Push (Stack, I);
   end loop;

   Print_Pop_Stack (Stack, "Stack");
   Print_Pop_Stack (Stack_2, "Stack_2");

end Show_Unbounded_Stack;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Applications.
↪Unbounded_Stacks
MD5: 22c795f2dfd2fbdf5468b54722d7126b
```

Runtime output

```
Initializing Unbounded_Stack (1 ..  5)
Initializing Unbounded_Stack (1 ..  5)
Increasing Unbounded_Stack (1 ..  5) to (1 ..  10)
Finalizing Unbounded_Stack (1 ..  5)
Adjusting Unbounded_Stack...
Duplicating data for new Unbounded_Stack (1 ..  10)
Increasing Unbounded_Stack (1 ..  10) to (1 ..  15)
Increasing Unbounded_Stack (1 ..  15) to (1 ..  20)
STACK: Stack
=  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1
STACK: Stack_2
=  10  9  8  7  6  5  4  3  2  1
Finalizing Unbounded_Stack (1 ..  10)
Finalizing Unbounded_Stack (1 ..  20)
```

Let's first focus on the Unbounded_Stack type from the Unbounded_Stacks package. The actual stack is implemented via the array that we allocate for the Data component. The initial allocation takes place in the Initialize procedure, which is called when an object of Unbounded_Stack type is created. The corresponding deallocation of the stack happens in the Finalize procedure.

In the Push procedure, we check whether the stack is full or not before storing a new element into the stack. If the stack is full, we call the Increase_Size procedure to *increase* the size of the array. This is actually done by calling the Reallocate_Data procedure, which allocates a new array for the stack and copies the original data to the new array.

Also, when copying an unbounded stack object to another object of this type, a call to the Adjust procedure is triggered — we do this by the assignment Stack_2 := Stack in the Show_Unbounded_Stack procedure. In the Adjust procedure, we call the Allocate_Duplicate_Data procedure to allocate a new array for the stack data and copy the data from the original stack. (Internally, the Allocate_Duplicate_Data procedure calls the Reallocate_Data procedure, which we already mentioned.)

By encapsulating the access type handling in controlled types, we can ensure that the access objects are handled correctly: no incorrect pointer usage or memory leak can happen when we use this strategy.

18.6.2 Encapsulating file handling

Controlled types can be used to encapsulate file handling, so that files are automatically created and closed. A common use-case is when a new file is expected to be created or opened when we declare the controlled object, and closed when the controlled object gets out of scope.

A simple example is the one of a logger, which we can use to write to a logfile by simple calls to Put_Line:

Listing 58: loggers.ads

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

package Loggers is

   type Logger (<>) is
     limited private;

   function Init (Filename : String)
                  return Logger;

   procedure Put_Line (L : Logger;
                       S : String);

private

   type Logger is new
     Ada.Finalization.Limited_Controlled with
      record
         Logfile : File_Type;
      end record;

   procedure Finalize
     (L : in out Logger);

end Loggers;
```

Listing 59: loggers.adb

```
package body Loggers is

   --
   --  SUBPROGRAMS
   --

   function Init (Filename : String)
                  return Logger is
   begin
      return L : Logger do
         Create (L.Logfile, Out_File, Filename);
      end return;
   end Init;

   procedure Put_Line (L : Logger;
                       S : String) is
   begin
      Put_Line ("Logger: Put_Line");
      Put_Line (L.Logfile, S);
   end Put_Line;

```

(continues on next page)

(continued from previous page)

```
   --
   --  PRIVATE SUBPROGRAMS
   --

   procedure Finalize
     (L : in out Logger) is
   begin
      Put_Line ("Finalizing Logger...");
      if Is_Open (L.Logfile) then
         Close (L.Logfile);
      end if;
   end Finalize;

end Loggers;
```

Listing 60: some_processing.adb

```
with Loggers; use Loggers;

procedure Some_Processing (Log : Logger) is
begin
   Put_Line (Log, "Some processing...");
end Some_Processing;
```

Listing 61: show_logger.adb

```
with Loggers;         use Loggers;
with Some_Processing;

procedure Show_Logger is
   Log : constant Logger := Init ("report.log");
begin
   Put_Line (Log, "Some info...");
   Some_Processing (Log);
end Show_Logger;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Applications.
↪Logger
MD5: 0ac4b5dff9ded8b64cb2b1f001e763fa
```

```
Some info...
Some processing...
```

Runtime output

```
Logger: Put_Line
Logger: Put_Line
Finalizing Logger...
```

The Logger type from the Loggers package has two subprograms:

- Init, which creates a logger object and creates a logfile *in the background*, and
- Put_Line, which writes a message to the logfile.

Note that we use the (<>) in the declaration of the Logger type to ensure that clients call the Init function. This allows us to specify the location of the logfile (as the Filename parameter).

Also, we can pass the logger to other subprograms and use it there. In this example, we pass the logger to the Some_Processing procedure and there, we the call Put_Line using the logger object.

Finally, as soon as the logger goes out of scope, the log is automatically closed via the call to Finalize.

For further reading...

Instead of enforcing a call to Init, we could have overridden the Initialize procedure and opened the logfile there. This approach, however, would have prevented the client from specifying the location of the logfile in a simple way. Specifying the filename as a type discriminant wouldn't work because we cannot use a string as a discriminant — as we mentioned *in a previous chapter* (page 200), we cannot use indefinite subtypes as discriminants.

If we had preferred this approach, we could generate a random name for the file in the Initialize procedure and store the file itself in a temporary directory indicated by the operating system. Alternatively, we could use the access to a string as a discriminant:

Listing 62: loggers.ads

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

package Loggers is

   type Logger (Filename : access String) is
     limited private;

   procedure Put_Line (L : Logger;
                       S : String);

private

   type Logger (Filename : access String) is new
     Ada.Finalization.Limited_Controlled with
      record
         Logfile : File_Type;
      end record;

   procedure Initialize
     (L : in out Logger);

   procedure Finalize
     (L : in out Logger);

end Loggers;
```

Listing 63: loggers.adb

```
package body Loggers is

   --
   --  SUBPROGRAMS
   --

   procedure Put_Line (L : Logger;
                       S : String) is
   begin
      Put_Line ("Logger: Put_Line");
      Put_Line (L.Logfile, S);
   end Put_Line;

   --
   --  PRIVATE SUBPROGRAMS
   --

   procedure Initialize
     (L : in out Logger) is
   begin
      Create (L.Logfile,
              Out_File,
              L.Filename.all);
   end Initialize;

   procedure Finalize
     (L : in out Logger) is
   begin
      Put_Line ("Finalizing Logger...");
      if Is_Open (L.Logfile) then
         Close (L.Logfile);
      end if;
   end Finalize;

end Loggers;
```

Listing 64: show_logger.adb

```
with Loggers;         use Loggers;
with Some_Processing;

procedure Show_Logger is
   Name : aliased String := "report.log";
   Log  : Logger (Name'Access);
begin
   Put_Line (Log, "Some info...");
   Some_Processing (Log);
end Show_Logger;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Applications.
↪Logger
MD5: d60ffbafd26d3d70a3d7807487dd95ab
```

```
Some info...
Some processing...
```

Runtime output

```
Logger: Put_Line
Logger: Put_Line
Finalizing Logger...
```

This approach works, but requires us to declare an aliased string (Name), which we can give access to in the declaration of the Log object.

By encapsulating the file handling in controlled types, we ensure that files are properly opened when we want to use them, and that the files are closed when they're not going to be used anymore.