



Variable Cards

Advanced Journey With Ada

Carriage
Gustavo A. Hoffmann
Robert A. Duff

Carriage
A Flight in Progress

Push
LEARN.
ADACORE.COM

**Advanced Journey With Ada: A
Flight In Progress**

Release 2024-07

**Gustavo A. Hoffmann
and Robert A. Duff**

Jul 20, 2024

CONTENTS

I	Data types	3
1	Types	5
1.1	Scalar Types	5
1.1.1	Ranges	6
1.1.2	Predecessor and Successor	6
1.1.3	Scalar To String Conversion	9
1.1.4	Width attribute	10
1.1.5	Base	11
1.2	Enumerations	15
1.2.1	Enumerations as functions	15
1.2.2	Enumeration overloading	18
1.2.3	Position and Internal Code	22
1.3	Definite and Indefinite Subtypes	23
1.3.1	Constrained Attribute	31
1.4	Incomplete types	34
1.5	Type view	35
1.5.1	Non-Record Private Types	38
1.6	Type conversion	42
1.6.1	Value conversion	43
1.6.2	View conversion	52
1.6.3	Implicit conversions	56
1.6.4	Conversion of other types	57
1.7	Qualified Expressions	61
1.7.1	Verifying subtypes	62
1.8	Default initial values	63
1.9	Deferred Constants	66
1.10	User-defined literals	68
2	Types and Representation	77
2.1	Enumeration Representation Clauses	77
2.2	Data Representation	79
2.2.1	Sizes	79
2.2.2	Alignment	87
2.2.3	Overlapping Storage	90
2.2.4	Packed Representation	93
2.3	Record Representation and storage clauses	97
2.3.1	Storage Place Attributes	99
2.3.2	Using Representation Clauses	101
2.3.3	Derived Types And Representation Clauses	103
2.3.4	Representation on Bit Level	104
2.4	Changing Data Representation	106
2.4.1	Restrictions	110
2.5	Valid Attribute	113
2.6	Unchecked Union	116

2.7	Shared variable control	122
2.7.1	Volatile	122
2.7.2	Independent	124
2.7.3	Atomic	129
2.8	Addresses	131
2.8.1	Address attribute	132
2.8.2	Address aspect	133
2.8.3	Address comparison	135
2.8.4	Address to integer conversion	136
2.8.5	Address arithmetic	137
2.9	Discarding names	140
3	Records	143
3.1	Default Initialization	143
3.1.1	Dependencies	144
3.1.2	Initialization Order	144
3.1.3	Evaluation	146
3.1.4	Defaults and object declaration	147
3.1.5	Advanced Usages	149
3.2	Mutually dependent types	149
3.3	Null records	151
3.3.1	Simple Prototyping	153
3.3.2	Extending the prototype	154
3.3.3	More complex applications	156
3.3.4	Implementing the API	157
3.3.5	Tagged null records	159
3.4	Per-Object Expressions	160
3.4.1	Default value	163
3.4.2	Restrictions	164
4	Aggregates	167
4.1	Container Aggregates	167
4.2	Record aggregates	170
4.2.1	<>	172
4.2.2	others	176
4.2.3	Record discriminants	178
4.3	Full coverage rules for Aggregates	180
4.4	Array aggregates	182
4.4.1	Positional and named array aggregates	182
4.4.2	Null array aggregate	185
4.4.3	, <>, others	187
4.4.4	188
4.4.5	Missing components	189
4.4.6	Iterated component association	190
4.4.7	Multidimensional array aggregates	192
4.4.8	<> and default values	197
4.5	Extension Aggregates	202
4.5.1	Assignments to objects of derived types	202
4.5.2	Example: Points	203
4.5.3	Using extension aggregates	205
4.5.4	More extension aggregates	206
4.5.5	with others	206
4.5.6	with null record	207
4.5.7	Extension aggregates and descendent types	208
4.6	Delta Aggregates	209
4.6.1	Delta Aggregates for Tagged Records	209
4.6.2	Delta Aggregates for Non-Tagged Records	212
4.6.3	Delta Aggregates for Arrays	213

5	Arrays	217
5.1	Unconstrained Arrays	217
5.1.1	Unconstrained Arrays vs. Vectors	218
5.2	Multidimensional Arrays	219
5.2.1	Unconstrained Multidimensional Arrays	224
5.2.2	Arrays of arrays	225
6	Strings	229
6.1	Wide and Wide-Wide Strings	229
6.1.1	Text I/O	231
6.1.2	Wide and Wide-Wide String Handling	233
6.1.3	Bounded and Unbounded Wide and Wide-Wide Strings	235
6.2	String Encoding	236
6.2.1	UTF-8 encoding and decoding	236
6.2.2	UTF-8 size and length	238
6.2.3	UTF-8 encoding in source-code files	240
6.2.4	UTF-16 encoding and decoding	244
6.3	Image attribute	246
6.3.1	Overview	246
6.3.2	Type 'Image and Obj 'Image	247
6.3.3	Wider versions of Image	248
6.3.4	Image attribute for non-scalar types	249
6.3.5	Image attribute for tagged types	251
6.3.6	Image attribute for task and protected types	252
6.4	Put_Image aspect	253
6.4.1	Overview	254
6.4.2	Complete Example of Put_Image	255
6.4.3	Relation to the Image attribute	256
6.4.4	Put_Image and derived types	257
6.4.5	Put_Image and tagged types	259
6.5	Universal text buffer	261
6.5.1	Overview	261
6.5.2	Additional procedures	262
7	Numerics	265
7.1	Modular Types	265
7.1.1	Modulus Attribute	265
7.1.2	Mod Attribute	266
7.1.3	Operations on modular types	268
7.2	Numeric Literals	270
7.2.1	Classification	270
7.2.2	Features and Flexibility	272
7.3	Floating-Point Types	277
7.3.1	Representation-oriented attributes	277
7.3.2	Primitive function attributes	283
7.4	Fixed-Point Types	291
7.4.1	Attributes of fixed-point types	291
7.4.2	Attributes of decimal fixed-point types	297
7.5	Big Numbers	299
7.5.1	Overview	300
7.5.2	Factorial	302
7.5.3	Conversions	305
7.5.4	Other features of big integers	312
7.5.5	Other operators for big integers	313
7.5.6	Big real and quotients	314
7.5.7	Range checks	315

II	Control Flow	317
8	Expressions	319
8.1	Expressions: Definition	319
8.1.1	Relations and simple expressions	319
8.1.2	Numeric expressions	322
8.1.3	Other expressions	323
8.1.4	Parenthesized expression	323
8.2	Conditional Expressions	326
8.3	Quantified Expressions	328
8.4	Declare Expressions	332
8.4.1	Restrictions in the declarative part	334
8.5	Reduction Expressions	336
8.5.1	Value sequences	338
8.5.2	Custom reducers	339
8.5.3	Other accumulator types	341
9	Statements	343
9.1	Simple and Compound Statements	343
9.2	Labels	343
9.2.1	Labels and goto statements	344
9.2.2	Use-case: Continue	345
9.2.3	Labels and compound statements	346
9.3	Exit loop statement	348
9.4	If, case and loop statements	349
9.4.1	Case statements and expressions	351
9.5	Block Statements	353
9.6	Extended return statement	354
9.6.1	Other usages of extended return statements	355
10	Subprograms	357
10.1	Parameter Modes and Associations	357
10.1.1	Formal Parameter Modes	357
10.1.2	By-copy and by-reference	358
10.1.3	Bounded errors	363
10.1.4	Aliased parameters	365
10.1.5	Parameter Associations	366
10.2	Operators	370
10.2.1	User-defined operators	370
10.3	Expression functions	376
10.4	Overloading	379
10.5	Operator Overloading	384
10.6	Operator Overriding	384
10.7	Nonreturning procedures	387
10.8	Inline subprograms	390
10.9	Null Procedures	392
10.9.1	Null procedures and overriding	393
11	Exceptions	397
11.1	Asserts	397
11.2	Assertion policies	399
11.3	Checks and exceptions	402
11.3.1	Access Check	402
11.3.2	Discriminant Check	404
11.3.3	Division Check	405
11.3.4	Index Check	406
11.3.5	Length Check	407
11.3.6	Overflow Check	407
11.3.7	Range Check	408

11.3.8 Tag Check	409
11.3.9 Accessibility Check	410
11.3.10 Allocation Check	411
11.3.11 Elaboration Check	413
11.3.12 Storage Check	414
11.4 Ada.Exceptions package	415
11.4.1 Retrieving exception information	415
11.4.2 Collecting exceptions	416
11.4.3 Debugging exceptions in the GNAT toolchain	420
11.5 Exception renaming	423
11.6 Out and Uninitialized	424
11.7 Suppressing checks	428
11.7.1 pragma Suppress	428
11.7.2 pragma Unsuppress	430
III Modular programming	433
12 Packages	435
12.1 Package renaming	435
12.1.1 Grouping packages	435
12.1.2 Child of renamed package	437
12.1.3 Backwards-compatibility via renaming	437
12.2 Private packages	438
12.2.1 Declaration and usage	439
12.2.2 Private sibling packages	441
12.2.3 Outside the package tree	443
12.3 Private with clauses	446
12.3.1 Definition and usage	446
12.3.2 Referring to private child package	448
12.4 Limited Visibility	450
12.4.1 Limited visibility and private with clauses	452
12.4.2 Limited visibility and other elements	453
12.5 Visibility	454
12.5.1 Automatic visibility	454
12.5.2 With clauses and visibility	455
12.5.3 Circular dependency	458
12.5.4 Private packages	460
12.6 Use type clause	462
12.6.1 Another use clause example	463
12.6.2 Visibility and Readability	464
12.6.3 use type	465
12.6.4 use all type	465
12.7 Use clauses and naming conflicts	466
12.7.1 Code example	466
12.7.2 Naming conflict	467
12.7.3 Circumventing naming conflicts	468
13 Subprograms and Modularity	473
13.1 Private subprograms	473
13.1.1 Private subprograms of a package	474
13.1.2 Private subprograms and private packages	475
IV Resource Management	479
14 Access Types	481
14.1 Access types: Terminology	481
14.1.1 Access type, designated subtype and profile	481

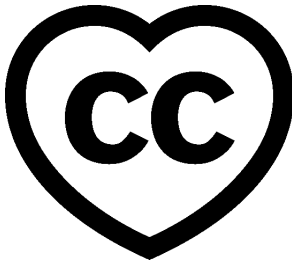
14.1.2	Access object and designated object	482
14.1.3	Access value and designated value	483
14.2	Access types: Allocation	483
14.2.1	Pool-specific access types	486
14.2.2	Multiple allocation	488
14.3	Discriminants as Access Values	492
14.3.1	Unconstrained type as designated subtype	494
14.3.2	Whole object assignments	497
14.4	Parameters as Access Values	498
14.4.1	Changing the referenced object	500
14.4.2	Replace the access value	502
14.4.3	Side-effects on designated objects	503
14.5	Self-reference	509
14.6	Mutually dependent types using access types	512
14.7	Dereferencing	512
14.7.1	Implicit Dereferencing	514
14.8	Ragged arrays	519
14.8.1	Uniform multidimensional arrays	519
14.8.2	Non-uniform multidimensional array	521
14.9	Aliasing	523
14.9.1	Aliased objects	525
14.9.2	Aliased components	530
14.9.3	Aliased parameters	532
14.10	Accessibility Levels and Rules: An Introduction	534
14.10.1	Lifetime of objects	534
14.10.2	Accessibility Levels	535
14.10.3	Accessibility Rules	536
14.10.4	Accessibility rules on parameters	540
14.10.5	Dangling References	541
14.11	Unchecked Access	544
14.12	Unchecked Deallocation	546
14.12.1	Unchecked Deallocation and Dangling References	549
14.12.2	Dereferencing dangling references	551
14.12.3	Restrictions for Ada.Unchecked_Deallocation	552
14.13	Null & Not Null Access	554
14.14	Design strategies for access types	558
14.14.1	Abstract data type for access types	558
14.14.2	Controlled type for access types	561
14.15	Access to subprograms	566
14.15.1	Static vs. dynamic calls	566
14.15.2	Access to subprogram declaration	567
14.15.3	Objects of access-to-subprogram type	569
14.15.4	Components of access-to-subprogram type	570
14.15.5	Access-to-subprogram as discriminant types	573
14.15.6	Access-to-subprograms as formal parameters	575
14.15.7	Selecting subprograms	578
14.15.8	Null exclusion	580
14.15.9	Access to protected subprograms	585
14.16	Accessibility Rules and Access-To-Subprograms	591
14.16.1	Unchecked Access	593
14.17	Access and Address	595
14.17.1	Address and access conversion	596
15	Anonymous Access Types	601
15.1	Named and Anonymous Access Types	601
15.1.1	Relation to named types	602
15.1.2	Benefits of anonymous access types	602
15.2	Anonymous Access-To-Object Types	605

15.2.1	Not Null Anonymous Access-To-Object Types	607
15.2.2	Drawbacks of Anonymous Access-To-Object Types	608
15.3	Access discriminants	615
15.3.1	Default Value of Access Discriminants	617
15.3.2	Benefits of Access Discriminants	619
15.3.3	Preventing dangling pointers	621
15.4	Self-reference	622
15.5	Mutually dependent types using anonymous access types	624
15.6	Access parameters	625
15.6.1	Interfacing To Other Languages	628
15.6.2	Inherited Primitive Operations For Tagged Types	631
15.7	User-Defined References	634
15.7.1	Dereferencing of tagged types	636
15.7.2	Simple container	637
15.8	Anonymous Access Types and Accessibility Rules	643
15.8.1	Conversions between Anonymous and Named Access Types	645
15.8.2	Accessibility rules on access parameters	647
15.9	Anonymous Access-To-Subprograms	648
15.9.1	Examples of anonymous access-to-subprogram usage	650
15.9.2	Application of anonymous access-to-subprogram types	656
15.9.3	Readability	656
15.10	Accessibility Rules and Anonymous Access-To-Subprograms	658
15.10.1	Named vs. anonymous access-to-subprograms	658
15.10.2	Named vs. anonymous access-to-subprograms as parameters	659
15.10.3	Iterator	663
16	Limited Types	669
16.1	Assignment and equality	669
16.1.1	Assignments	672
16.1.2	Equality	674
16.2	Limited private types	677
16.2.1	Non-Record Limited Types	678
16.2.2	Partial and full view of limited types	679
16.2.3	Limited and nonlimited in full view	682
16.2.4	Limited private component	683
16.2.5	Tagged limited private types	685
16.3	Explicitly limited types	685
16.4	Subtypes of Limited Types	687
16.5	Deriving from limited types	688
16.5.1	Deriving from limited private types	689
16.5.2	Deriving from non-explicitly limited private types	690
16.6	Immutable Limited Types	695
16.6.1	Non immutably limited types	698
16.7	Record components of limited type	698
16.8	Limited types and aggregates	699
16.8.1	Full coverage rules for limited types	700
16.9	Constructor functions for limited types	703
16.10	Return objects	707
16.10.1	Extended return statements for limited types	707
16.10.2	Initialization and function return	709
16.11	Building objects from constructors	713
16.12	Limited types as parameter	716

Warning: This version of the website contains UNPUBLISHED contents. Please do not share it externally!

Copyright © 2019 - 2023, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](#)¹



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

¹ <http://creativecommons.org/licenses/by-sa/4.0>

² https://learn.adacore.com/zip/learning-ada_code.zip

you will find it in this directory:

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

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

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

This course will teach you advanced topics of the Ada programming language. The [Introduction to Ada³](#) 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 Arnaud Charlet, Emmanuel Briot, Franco Gasperoni, Gary Dismukes, Javier Miranda, Patrick Rogers, Quentin Ochem, Robert Dewar, and Yannick Moy.

These contributions are clearly indicated in the document, together with the original publication source.

Special thanks to Patrick Rogers for all comments and suggestions. In particular, thanks for sharing the training slides on access types: many ideas from those slides were integrated into this course.

This document was reviewed by Patrick Rogers and Tucker Taft.

CHANGELOG

Changes are being tracked on the [CHANGELOG](#) page.

³ <https://learn.adacore.com/courses/intro-to-ada/index.html#intro-ada-course-index>

Part I

Data types

1.1 Scalar Types

In general terms, scalar types are the most basic types that we can get. As we know, we can classify them as follows:

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

We already discussed some of these attributes in the Introduction to Ada course (in the sections about [range and related attributes](#)⁴ and [image attribute](#)⁵). In this section, we'll discuss some aspects that have been left out of the previous course.

In the Ada Reference Manual

- [3.5 Scalar types](#)⁶

⁴ <https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-range-attribute>

⁵ https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-image-attribute

⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-5.html>

1.1.1 Ranges

We've seen that the `First` and `Last` attributes can be used with discrete types. Those attributes are also available for real types. Here's an example using the `Float` type and a subtype of it:

Listing 1: `show_first_last_real.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_First_Last_Real is
4     subtype Norm is Float range 0.0 .. 1.0;
5 begin
6     Put_Line ("Float'First: " & Float'First'Image);
7     Put_Line ("Float'Last:  " & Float'Last'Image);
8     Put_Line ("Norm'First:  " & Norm'First'Image);
9     Put_Line ("Norm'Last:   " & Norm'Last'Image);
10 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).

1.1.2 Predecessor and Successor

We can use the `Pred` and `Succ` attributes to get the predecessor and successor of a specific value. For discrete types, this is simply the next discrete value. For example, `Pred (2)` is 1 and `Succ (2)` is 3. Let's look at a complete source-code example:

Listing 2: `show_succ_pred_discrete.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Succ_Pred_Discrete is
4     type State is (Idle, Started,
5                   Processing, Stopped);
6
7     Machine_State : constant State := Started;
8
9     I : constant Integer := 2;
10 begin
11     Put_Line ("State           : "
12             & Machine_State'Image);
13     Put_Line ("State'Pred (Machine_State): "
14             & State'Pred (Machine_State)'Image);
15     Put_Line ("State'Succ (Machine_State): "
16             & State'Succ (Machine_State)'Image);
17     Put_Line ("-----");
```

(continues on next page)

(continued from previous page)

```

18
19   Put_Line ("I           : "
20           & I'Image);
21   Put_Line ("Integer'Pred (I): "
22           & Integer'Pred (I)'Image);
23   Put_Line ("Integer'Succ (I): "
24           & Integer'Succ (I)'Image);
25 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 3: show_succ_pred_real.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Succ_Pred_Real is
4      subtype My_Float is
5          Float range 0.0 .. 0.5;
6
7      type Decimal is
8          delta 0.1 digits 2
9          range 0.0 .. 0.5;
10
11     D : Decimal;
12     N : My_Float;
13 begin
14     Put_Line ("---- DECIMAL -----");
15     Put_Line ("Small: " & Decimal'Small'Image);
16     Put_Line ("----- Succ -----");
17     D := Decimal'First;
18     loop
19         Put_Line (D'Image);
20         D := Decimal'Succ (D);
21

```

(continues on next page)

(continued from previous page)

```

22     exit when D = Decimal'Last;
23 end loop;
24 Put_Line ("----- Pred -----");
25
26 D := Decimal'Last;
27 loop
28     Put_Line (D'Image);
29     D := Decimal'Pred (D);
30
31     exit when D = Decimal'First;
32 end loop;
33 Put_Line ("=====");
34
35 Put_Line ("---- MY_FLOAT ----");
36 Put_Line ("----- Succ -----");
37 N := My_Float'First;
38 for I in 1 .. 5 loop
39     Put_Line (N'Image);
40     N := My_Float'Succ (N);
41 end loop;
42 Put_Line ("----- Pred -----");
43
44 for I in 1 .. 5 loop
45     Put_Line (N'Image);
46     N := My_Float'Pred (N);
47 end loop;
48 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: 1.0000000000000000E-01
----- Succ -----
0.0
0.1
0.2
0.3
0.4
----- Pred -----
0.5
0.4
0.3
0.2
0.1
=====
---- MY_FLOAT ----
----- Succ -----
0.00000E+00
1.40130E-45
2.80260E-45
4.20390E-45
5.60519E-45
----- Pred -----
7.00649E-45
5.60519E-45
4.20390E-45

```

(continues on next page)

(continued from previous page)

```
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.

1.1.3 Scalar To String Conversion

We've seen that we can use the `Image` and `Value` attributes to perform conversions between values of a given subtype and a string:

Listing 4: `show_image_value_attr.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Image_Value_Attr is
4   I : constant Integer := Integer'Value ("42");
5 begin
6   Put_Line (I'Image);
7 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	<code>Image</code>	<code>String</code>
	<code>Wide_Image</code>	<code>Wide_String</code>
	<code>Wide_Wide_Image</code>	<code>Wide_Wide_String</code>
Conversion to subtype	<code>Value</code>	<code>String</code>
	<code>Wide_Value</code>	<code>Wide_String</code>
	<code>Wide_Wide_Value</code>	<code>Wide_Wide_String</code>

We discuss more about `Wide_String` and `Wide_Wide_String` in *another section* (page 229).

1.1.4 Width attribute

When converting a value to a string by using the `Image` attribute, we get a string with variable width. We can assess the maximum width of that string for a specific subtype by using the `Width` attribute. For example, `Integer'Width` gives us the maximum width returned by the `Image` attribute when converting a value of `Integer` type to a string of `String` type.

This attribute is useful when we're using bounded strings in our code to store the string returned by the `Image` attribute. For example:

Listing 5: show_width_attr.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Strings;          use Ada.Strings;
3 with Ada.Strings.Bounded;
4
5 procedure Show_Width_Attr is
6   package B_Str is new
7     Ada.Strings.Bounded.Generic_Bounded_Length
8     (Max => Integer'Width);
9   use B_Str;
10
11   Str_I : Bounded_String;
12
13   I : constant Integer := 42;
14   J : constant Integer := 103;
15 begin
16   Str_I := To_Bounded_String (I'Image);
17   Put_Line ("Value: "
18     & To_String (Str_I));
19   Put_Line ("String Length: "
20     & Length (Str_I)'Image);
21   Put_Line ("----");
22
23   Str_I := To_Bounded_String (J'Image);
24   Put_Line ("Value: "
25     & To_String (Str_I));
26   Put_Line ("String Length: "
27     & Length (Str_I)'Image);
28 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`.

1.1.5 Base

The Base attribute gives us the unconstrained underlying hardware representation selected for a given numeric type. As an example, let's say we declared a subtype of the **Integer** type named `One_To_Ten`:

Listing 6: my_integers.ads

```

1 package My_Integers is
2
3     subtype One_To_Ten is Integer
4         range 1 .. 10;
5
6 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 7: int_def.ads

```
1 package Int_Def is
2
3     type Too_Big_To_Fail is
4         range -2 ** 255 .. 2 ** 255 - 1;
5
6 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 8: show_base.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with My_Integers; use My_Integers;
3
4 procedure Show_Base is
5     C : constant One_To_Ten := One_To_Ten'Last;
6 begin
7     Using_Constrained_Subtype : declare
8         V : One_To_Ten := C;
9     begin
10        Put_Line
11            ("Increasing value for One_To_Ten...");
12
13        V := One_To_Ten'Succ (V);
14    exception
15        when others =>
16            Put_Line ("Exception raised!");
17    end Using_Constrained_Subtype;
18
19    Using_Base : declare
20        V : One_To_Ten'Base := C;
21    begin
22        Put_Line
23            ("Increasing value for One_To_Ten'Base...");
24
25        V := One_To_Ten'Succ (V);
26    exception
27        when others =>
28            Put_Line ("Exception raised!");
29    end Using_Base;
30
31    Put_Line ("One_To_Ten'Last: "
```

(continues on next page)

(continued from previous page)

```

32         & One_To_Ten'Last'Image);
33     Put_Line ("One_To_Ten'Base'Last: "
34             & One_To_Ten'Base'Last'Image);
35 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 9: my_integers.ads

```

1 package My_Integers is
2
3     subtype One_To_Ten is Integer range 1 .. 10;
4
5     function Sat_Add (V1, V2 : One_To_Ten'Base)
6         return One_To_Ten;
7
8     function Sat_Sub (V1, V2 : One_To_Ten'Base)
9         return One_To_Ten;
10
11 end My_Integers;
```

Listing 10: my_integers.adb

```

1 -- with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Integers is
4
5     function Saturate (V : One_To_Ten'Base)
6         return One_To_Ten is
7     begin
```

(continues on next page)

(continued from previous page)

```

8      -- Put_Line ("SATURATE " & V'Image);
9
10     if V < One_To_Ten'First then
11         return One_To_Ten'First;
12     elsif V > One_To_Ten'Last then
13         return One_To_Ten'Last;
14     else
15         return V;
16     end if;
17 end Saturate;
18
19 function Sat_Add (V1, V2 : One_To_Ten'Base)
20     return One_To_Ten is
21 begin
22     return Saturate (V1 + V2);
23 end Sat_Add;
24
25 function Sat_Sub (V1, V2 : One_To_Ten'Base)
26     return One_To_Ten is
27 begin
28     return Saturate (V1 - V2);
29 end Sat_Sub;
30
31 end My_Integers;

```

Listing 11: show_base.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with My_Integers; use My_Integers;
3
4 procedure Show_Base is
5
6     type Display_Saturate_Op is (Add, Sub);
7
8     procedure Display_Saturate
9         (V1, V2 : One_To_Ten;
10          Op    : Display_Saturate_Op)
11     is
12         Res : One_To_Ten;
13     begin
14         case Op is
15         when Add =>
16             Res := Sat_Add (V1, V2);
17         when Sub =>
18             Res := Sat_Sub (V1, V2);
19         end case;
20         Put_Line ("SATURATE " & Op'Image
21                 & " (" & V1'Image
22                 & ", " & V2'Image
23                 & ") = " & Res'Image);
24     end Display_Saturate;
25
26 begin
27     Display_Saturate (1, 1, Add);
28     Display_Saturate (10, 8, Add);
29     Display_Saturate (1, 8, Sub);
30 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.

1.2 Enumerations

We've introduced enumerations back in the [Introduction to Ada course](#)⁷. 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](#)⁸
-

1.2.1 Enumerations as functions

If you have used programming language such as C in the past, you're familiar with the concept of enumerations being constants with integer values. In Ada, however, enumerations are not integers. In fact, they're actually parameterless functions! Let's consider this example:

Listing 12: days.ads

```
1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     -- Essentially, we're declaring
8     -- these functions:
9     --
```

(continues on next page)

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

⁸ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-1.html>

(continued from previous page)

```
10  -- function Mon return Day;
11  -- function Tue return Day;
12  -- function Wed return Day;
13  -- function Thu return Day;
14  -- function Fri return Day;
15  -- function Sat return Day;
16  -- function Sun return Day;
17
18  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 13: enumeration_example.ads

```
1  package Enumeration_Example is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     function Monday    return Day renames Mon;
8     function Tuesday   return Day renames Tue;
9     function Wednesday return Day renames Wed;
10    function Thursday  return Day renames Thu;
11    function Friday     return Day renames Fri;
12    function Saturday  return Day renames Sat;
13    function Sunday    return Day renames Sun;
14
15  end Enumeration_Example;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: e2e12bb3bfc0b6e94769ced9a4b80f9
```

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

Listing 14: show_renaming.adb

```
1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Enumeration_Example; use Enumeration_Example;
```

(continues on next page)

(continued from previous page)

```

3
4 procedure Show_Renaming is
5     D1 : constant Day := Mon;
6     D2 : constant Day := Monday;
7 begin
8     if D1 = D2 then
9         Put_Line ("D1 = D2");
10        Put_Line (Day'Image (D1)
11                & " = "
12                & Day'Image (D2));
13    end if;
14 end Show_Renaming;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: 2d7177def2c9e9fb11c7dc5e036c3be3

Runtime output

```

D1 = D2
MON = MON

```

When running this application, we can confirm that D1 is equal to D2. Also, even though we've assigned Monday to D2 (instead of Mon), the application displays Mon = Mon, since Monday is just another name to refer to the actual enumeration (Mon).

Hint

If you just want to have a single (renamed) enumeration visible in your application — and make the original enumeration invisible —, you can use a separate package. For example:

Listing 15: enumeration_example.ads

```

1 package Enumeration_Example is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7 end Enumeration_Example;

```

Listing 16: enumeration_renaming.ads

```

1 with Enumeration_Example;
2
3 package Enumeration_Renaming is
4
5     subtype Day is Enumeration_Example.Day;
6
7     function Monday return Day renames
8         Enumeration_Example.Mon;
9     function Tuesday return Day renames
10        Enumeration_Example.Tue;
11    function Wednesday return Day renames
12        Enumeration_Example.Wed;
13    function Thursday return Day renames
14        Enumeration_Example.Thu;
15    function Friday return Day renames
16        Enumeration_Example.Fri;

```

(continues on next page)

(continued from previous page)

```
17  function Saturday return Day renames
18     Enumeration_Example.Sat;
19  function Sunday   return Day renames
20     Enumeration_Example.Sun;
21
22  end Enumeration_Renaming;
```

Listing 17: show_renaming.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Enumeration_Renaming;
4  use Enumeration_Renaming;
5
6  procedure Show_Renaming is
7     D1 : constant Day := Monday;
8  begin
9     Put_Line (Day'Image (D1));
10  end Show_Renaming;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: 87fe75026f0fc118921eae45fe55a8a

Runtime output

```
MON
```

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

1.2.2 Enumeration overloading

Enumerations can be overloaded. In simple terms, this means that the same name can be used to declare an enumeration of different types. A typical example is the declaration of colors:

Listing 18: colors.ads

```
1  package Colors is
2
3     type Color is
4         (Salmon,
5          Firebrick,
6          Red,
7          Darkred,
8          Lime,
9          Forestgreen,
10         Green,
11         Darkgreen,
12         Blue,
13         Mediumblue,
14         Darkblue);
15
16     type Primary_Color is
17         (Red,
18         Green,
19         Blue);
```

(continues on next page)

(continued from previous page)

```
20
21 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 19: red_colors.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Red_Colors is
5   C1 : constant Color      := Red;
6   -- Using Red from Color
7
8   C2 : constant Primary_Color := Red;
9   -- Using Red from Primary_Color
10 begin
11   if C1 = Red then
12     Put_Line ("C1 = Red");
13   end if;
14   if C2 = Red then
15     Put_Line ("C2 = Red");
16   end if;
17 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 20: colors-shades.ads

```
1 package Colors.Shades is
2
3   subtype Blue_Shades is
4     Colors range Blue .. Darkblue;
```

(continues on next page)

(continued from previous page)

```
5
6 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 21: colors.ads

```
1 package Colors is
2
3     type Color is
4         (Salmon,
5          Firebrick,
6          Red,
7          Darkred,
8          Lime,
9          Forestgreen,
10         Green,
11         Darkgreen,
12         Blue,
13         Mediumblue,
14         Darkblue);
15
16     type Primary_Color is
17         (Red,
18          Green,
19          Blue);
20
21 end Colors;
```

Listing 22: color_loop.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Color_Loop is
5 begin
6     for C in Red .. Blue loop
7         --           ^^^^^^^^^^^
8         -- ERROR: range is ambiguous!
9         Put_Line (Color'Image (C));
10    end loop;
11 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 23: color_loop.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Color_Loop is
5 begin
6     for C in Color'(Red) .. Color'(Blue) loop
7         Put_Line (Color'Image (C));
8     end loop;
9 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 24: color_loop.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Color_Loop is
5 begin
6     for C in Color range Red .. Blue loop
7         Put_Line (Color'Image (C));
8     end loop;
9 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 25: color_loop.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Color_Loop is
5     subtype Red_To_Blue is Color range Red .. Blue;
6 begin
7     for C in Red_To_Blue loop
8         Put_Line (Color'Image (C));
9     end loop;
10 end Color_Loop;
```

1.2.3 Position and Internal Code

As we've said above, a typical Ada compiler doesn't generate function calls for code that deals with enumerations. On the contrary, each enumeration has values associated with it, and the compiler uses those values instead.

Each enumeration has:

- a position value, which is a natural value indicating the position of the enumeration in the enumeration type; and
- an internal code, which, by default, in most cases, is the same as the position value.

Also, by default, the value of the first position is zero, the value of the second position is one, and so on. We can see this by listing each enumeration of the `Day` type and displaying the value of the corresponding position:

Listing 26: days.ads

```
1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7 end Days;
```

Listing 27: show_days.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Days;        use Days;
3
4 procedure Show_Days is
5 begin
6     for D in Day loop
7         Put_Line (Day'Image (D))
```

(continues on next page)

(continued from previous page)

```

8         & " position      = "
9         & Integer'Image (Day'Pos (D)));
10    Put_Line (Day'Image (D)
11             & " internal code = "
12             & Integer'Image
13             (Day'Enum_Rep (D)));
14    end loop;
15    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 77).

1.3 Definite and Indefinite Subtypes

Indefinite types were mentioned back in the [Introduction to Ada course](#)⁹. In this section, we'll recapitulate and extend on both definite and indefinite types.

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

Indefinite types include:

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

Let's see some examples of indefinite types:

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

Listing 28: unconstrained_types.ads

```
1 package Unconstrained_Types is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     type Simple_Record (Extended : Boolean) is
7     record
8         V : Integer;
9         case Extended is
10            when False =>
11                null;
12            when True =>
13                V_Float : Float;
14        end case;
15    end record;
16
17 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.

Important

Note that we cannot use indefinite subtypes as discriminants. For example, the following code won't compile:

Listing 29: unconstrained_types.ads

```
1 package Unconstrained_Types is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     type Simple_Record (Arr : Integer_Array) is
7     record
8         L : Natural := Arr'Length;
9     end record;
10
11 end Unconstrained_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↳Indefinite_Types_Error
MD5: cf73d308ddb4a8c2503146ecd550a791
```

Build output

```
unconstrained_types.ads:6:30: error: discriminants must have a discrete or access_
↳type
gprbuild: *** compilation phase failed
```

`Integer_Array` is a correct type declaration — although the type itself is indefinite after the declaration. However, we cannot use it as the discriminant in the declaration of

Simple_Record. We could, however, have a correct declaration by using discriminants as access values:

Listing 30: unconstrained_types.ads

```
1 package Unconstrained_Types is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     type Integer_Array_Access is
7         access Integer_Array;
8
9     type Simple_Record
10        (Arr : Integer_Array_Access) is
11        record
12            L : Natural := Arr'Length;
13        end record;
14
15 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 492).

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

Listing 31: using_unconstrained_type.adb

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Using_Unconstrained_Type is
4
5     A : Integer_Array;
6
7     R : Simple_Record;
8
9 begin
10    null;
11 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)
```

(continues on next page)

(continued from previous page)

```
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 32: using_unconstrained_type.adb

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Using_Unconstrained_Type is
4
5     subtype Integer_Array_5 is
6         Integer_Array (1 .. 5);
7
8     A1 : Integer_Array_5;
9     A2 : Integer_Array (1 .. 5);
10
11     subtype Simple_Record_Ext is
12         Simple_Record (Extended => True);
13
14     R1 : Simple_Record_Ext;
15     R2 : Simple_Record (Extended => True);
16
17 begin
18     null;
19 end Using_Unconstrained_Type;
```

Code block metadata

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

Build output

```
using_unconstrained_type.adb:8:04: warning: variable "A1" is never read and never_
↳assigned [-gnatwv]
using_unconstrained_type.adb:9:04: warning: variable "A2" is never read and never_
↳assigned [-gnatwv]
using_unconstrained_type.adb:14:04: warning: variable "R1" is never read and never_
↳assigned [-gnatwv]
using_unconstrained_type.adb:15:04: warning: variable "R2" is never read and never_
↳assigned [-gnatwv]
```

In this example, we declare the `Integer_Array_5` subtype, which is definite because we're constraining it to a range from 1 to 5, thereby defining the information that was missing in the indefinite type `Integer_Array`. Because we now have a definite type, we can use it to declare the `A1` variable. Similarly, we can use the indefinite type `Integer_Array` directly in the declaration of `A2` by specifying the previously unknown range.

Similarly, in this example, we declare the `Simple_Record_Ext` subtype, which is definite because we're initializing the record discriminant `Extended`. We can therefore use it in

the declaration of the R1 variable. Alternatively, we can simply use the indefinite type `Simple_Record` and specify the information required for the discriminants. This is what we do in the declaration of the R2 variable.

Although we cannot use indefinite types directly in variable declarations, they're very useful to generalize algorithms. For example, we can use them as parameters of a subprogram:

Listing 33: `show_integer_array.ads`

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Show_Integer_Array (A : Integer_Array);
```

Listing 34: `show_integer_array.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Integer_Array (A : Integer_Array)
4 is
5 begin
6   for I in A'Range loop
7     Put_Line (Positive'Image (I)
8               & ": "
9               & Integer'Image (A (I)));
10  end loop;
11  Put_Line ("-----");
12 end Show_Integer_Array;
```

Listing 35: `using_unconstrained_type.adb`

```
1 with Unconstrained_Types; use Unconstrained_Types;
2 with Show_Integer_Array;
3
4 procedure Using_Unconstrained_Type is
5   A_5 : constant Integer_Array (1 .. 5) :=
6     (1, 2, 3, 4, 5);
7   A_10 : constant Integer_Array (1 .. 10) :=
8     (1, 2, 3, 4, 5, others => 99);
9 begin
10  Show_Integer_Array (A_5);
11  Show_Integer_Array (A_10);
12 end Using_Unconstrained_Type;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↳ Indefinite_Types
MD5: 3f744fa5921a55865bc5361ec4c6eb88
```

Runtime output

```
1: 1
2: 2
3: 3
4: 4
5: 5
-----
1: 1
2: 2
3: 3
4: 4
5: 5
```

(continues on next page)

(continued from previous page)

```
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 36: show_integer_array_header.ads

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Show_Integer_Array_Header
4   (AA : Integer_Array;
5    HH : String);
```

Listing 37: show_integer_array_header.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Show_Integer_Array;
3
4 procedure Show_Integer_Array_Header
5   (AA : Integer_Array;
6    HH : String)
7 is
8 begin
9   Put_Line (HH);
10  Show_Integer_Array (AA);
11 end Show_Integer_Array_Header;
```

Listing 38: using_unconstrained_type.adb

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 with Show_Integer_Array_Header;
4
5 procedure Using_Unconstrained_Type is
6   A_5 : constant Integer_Array (1 .. 5) :=
7     (1, 2, 3, 4, 5);
8   A_10 : constant Integer_Array (1 .. 10) :=
9     (1, 2, 3, 4, 5, others => 99);
10 begin
11   Show_Integer_Array_Header (A_5,
12     "First example");
13   Show_Integer_Array_Header (A_10,
14     "Second example");
15 end Using_Unconstrained_Type;
```

Code block metadata

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

(continues on next page)

(continued from previous page)

```
↳Indefinite_Types
MD5: dd09f8c4089c6ad4c18410879f80f731
```

Runtime output

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

In this case, we're calling the `Show_Integer_Array` procedure with another unconstrained parameter (the `AA` parameter). However, although we could have a long *chain* of procedure calls using indefinite types in their parameters, we still use a (definite) object at the beginning of this chain. For example, for the `A_5` object, we have this chain:

```
A_5

==> Show_Integer_Array_Header (AA => A_5,
                               ...);

==> Show_Integer_Array (A => AA);
```

Therefore, at this specific call to `Show_Integer_Array`, even though `A` is declared as a parameter of indefinite type, the actual argument is of definite type because `A_5` is constrained — and, thus, of definite type.

Note that we can declare variables based on parameters of indefinite type. For example:

Listing 39: `show_integer_array_plus.ads`

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Show_Integer_Array_Plus
4   (A : Integer_Array;
5    V : Integer);
```

Listing 40: `show_integer_array_plus.adb`

```
1 with Show_Integer_Array;
2
3 procedure Show_Integer_Array_Plus
4   (A : Integer_Array;
5    V : Integer)
6 is
7   A_Plus : Integer_Array (A'Range);
8 begin
```

(continues on next page)

(continued from previous page)

```
9   for I in A_Plus'Range loop
10     A_Plus (I) := A (I) + V;
11   end loop;
12   Show_Integer_Array (A_Plus);
13 end Show_Integer_Array_Plus;
```

Listing 41: using_unconstrained_type.adb

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 with Show_Integer_Array_Plus;
4
5 procedure Using_Unconstrained_Type is
6   A_5 : constant Integer_Array (1 .. 5) :=
7     (1, 2, 3, 4, 5);
8 begin
9   Show_Integer_Array_Plus (A_5, 5);
10 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](#)¹⁰

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

1.3.1 Constrained Attribute

We can use the Constrained attribute to verify whether an object of discriminated type is constrained or not. Let's start our discussion by reusing the Simple_Record type from previous examples. In this version of the Unconstrained_Types package, we're adding a Reset procedure for the discriminated record type:

Listing 42: unconstrained_types.ads

```

1 package Unconstrained_Types is
2
3   type Simple_Record
4     (Extended : Boolean := False) is
5     record
6       V : Integer;
7       case Extended is
8         when False =>
9           null;
10        when True =>
11          V_Float : Float;
12        end case;
13     end record;
14
15     procedure Reset (R : in out Simple_Record);
16
17 end Unconstrained_Types;
```

Listing 43: unconstrained_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Unconstrained_Types is
4
5   procedure Reset (R : in out Simple_Record) is
6     Zero_Not_Extended : constant
7       Simple_Record := (Extended => False,
8                          V      => 0);
9
10    Zero_Extended : constant
11      Simple_Record := (Extended => True,
12                       V      => 0,
13                       V_Float => 0.0);
14  begin
15    Put_Line ("---- Reset: R'Constrained => "
16             & R'Constrained'Image);
17
18    if not R'Constrained then
19      R := Zero_Extended;
20    else
21      if R.Extended then
22        R := Zero_Extended;
23      else
24        R := Zero_Not_Extended;
25      end if;
26    end if;
27  end Reset;
28
29 end Unconstrained_Types;
```

Code block metadata

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

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 44: `using_constrained_attribute.adb`

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2
3 with Unconstrained_Types; use Unconstrained_Types;
4
5 procedure Using_Constrained_Attribute is
6   R1 : Simple_Record;
7   R2 : Simple_Record (Extended => False);
8
9   procedure Show_Rs is
10    begin
11      Put_Line ("R1'Constrained => "
12              & R1'Constrained'Image);
13      Put_Line ("R1.Extended => "
14              & R1.Extended'Image);
15      Put_Line ("--");
16      Put_Line ("R2'Constrained => "
17              & R2'Constrained'Image);
18      Put_Line ("R2.Extended => "
19              & R2.Extended'Image);
20      Put_Line ("-----");
21    end Show_Rs;
22  begin
23    Show_Rs;
24
25    Reset (R1);
26    Reset (R2);
27    Put_Line ("-----");
28
29    Show_Rs;
30  end Using_Constrained_Attribute;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.  
↳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 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 Types¹¹](#)

¹¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-7-2.html>

1.4 Incomplete types

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

```
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 45: incomplete_type_example.ads

```
1 package Incomplete_Type_Example is
2
3     type R;
4     -- Incomplete type declaration!
5
6     type R is record
7         I : Integer;
8     end record;
9     -- type R is now complete!
10
11 end Incomplete_Type_Example;
```

Code block metadata

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

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 46: incomplete_type_example.ads

```
1 package Incomplete_Type_Example is
2
3     private
4
5         type R;
6         -- Incomplete type declaration!
7
8     end Incomplete_Type_Example;
```

Listing 47: incomplete_type_example.adb

```
1 package body Incomplete_Type_Example is
2
3     type R is record
4         I : Integer;
5     end record;
6     -- type R is now complete!
7
8 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 based on those incomplete types. This kind of type is called a recursive type. For example:

Listing 48: linked_list_example.ads

```
1 package Linked_List_Example is
2
3   type Integer_List;
4
5   type Next is access Integer_List;
6
7   type Integer_List is record
8     I : Integer;
9     N : Next;
10  end record;
11
12 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 149), 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](#)¹²
-

1.5 Type view

Ada distinguishes between the partial and the full view of a type. The full view is a type declaration that contains all the information needed by the compiler. For example, the following declaration of type R represents the full view of this type:

Listing 49: full_view.ads

```
1 package Full_View is
2
3   -- Full view of the R type:
4   type R is record
5     I : Integer;
6   end record;
7
8 end Full_View;
```

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

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 50: partial_full_views.ads

```
1 package Partial_Full_Views is
2
3   -- Partial view of the R type:
4   type R is private;
5
6 private
7
8   -- Full view of the R type:
9   type R is record
10    I : Integer;
11  end record;
12
13 end Partial_Full_Views;
```

Code block metadata

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

As indicated in the example, the **type R is private** declaration is the partial view of the R type, while the **type R is record** [...] declaration in the private part of the package is the full view.

Although the partial view doesn't contain the full type declaration, it contains very important information for the users of the package where it's declared. In fact, the partial view of a private type is all that users actually need to know to effectively use this type, while the full view is only needed by the compiler.

In the previous example, the partial view indicates that R is a private type, which means that, even though users cannot directly access any information stored in this type — for example, read the value of the I component of R —, they can use the R type to declare objects. For example:

Listing 51: main.adb

```
1 with Partial_Full_Views; use Partial_Full_Views;
2
3 procedure Main is
4   -- Partial view of R indicates that
5   -- R exists as a private type, so we
6   -- can declare objects of this type:
7   C : R;
8 begin
9   -- But we cannot directly access any
10  -- information declared in the full
11  -- view of R:
12  --
13  -- C.I := 42;
14  --
15  null;
16 end Main;
```

Code block metadata

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

Build output

```
main.adb:7:04: warning: variable "C" is never read and never assigned [-gnatwv]
```

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 52: limited_private_example.ads

```
1 package Limited_Private_Example is
2
3   -- Partial view must be limited,
4   -- since the full view is limited.
5   type R is limited private;
6
7 private
8
9   type R is limited record
10    I : Integer;
11  end record;
12
13 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 53: tagged_full_view_example.ads

```
1 package Tagged_Full_View_Example is
2
3   -- Partial view using non-tagged type:
4   type R is private;
5
6 private
7
8   -- Full view using tagged type:
9   type R is tagged record
10    I : Integer;
11  end record;
12
13 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](#)¹³
-

1.5.1 Non-Record Private Types

Although it's very common to declare private types as record types, this is not the only option. In fact, we could declare any type in the full view — scalars, for example —, so we could declare a "private integer" type:

Listing 54: private_integers.ads

```
1 package Private_Integers is
2
3   -- Partial view of private Integer type:
4   type Private_Integer is private;
5
6 private
7
8   -- Full view of private Integer type:
9   type Private_Integer is new Integer;
10
11 end Private_Integers;
```

Code block metadata

```
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 55: private_integers.ads

```
1 package Private_Integers is
2
3   -- Partial view of private Integer type:
4   type Private_Integer is private;
5
6   function "+" (Left, Right : Private_Integer)
7     return Private_Integer;
8
9 private
10
11   -- Full view of private Integer type:
12   type Private_Integer is new Integer;
13
14 end Private_Integers;
```

Listing 56: private_integers.adb

```
1 package body Private_Integers is
2
3   function "+" (Left, Right : Private_Integer)
```

(continues on next page)

¹³ <http://www.ada-auth.org/standards/22rm/html/RM-7-3.html>

(continued from previous page)

```

4         return Private_Integer
5     is
6         Res : constant Integer :=
7             Integer (Left) + Integer (Right);
8         -- Note that we're converting Left
9         -- and Right to Integer, which calls
10        -- the "+" operator of the Integer
11        -- type. Writing "Left + Right" would
12        -- have called the "+" operator of
13        -- Private_Integer, which leads to
14        -- recursive calls, as this is the
15        -- operator we're currently in.
16    begin
17        return Private_Integer (Res);
18    end "+";
19
20 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 57: show_private_integers.adb

```

1 with Private_Integers; use Private_Integers;
2
3 procedure Show_Private_Integers is
4     A, B : Private_Integer;
5     begin
6         A := A + B;
7     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 58: private_arrays.ads

```
1 package Private_Arrays is
2
3     type Private_Unconstrained_Array is private;
4
5     type Private_Constrained_Array
6       (L : Positive) is private;
7
8 private
9
10    type Integer_Array is
11      array (Positive range <>) of Integer;
12
13    type Private_Unconstrained_Array is
14      array (Positive range <>) of Integer;
15
16    type Private_Constrained_Array
17      (L : Positive) is
18      array (1 .. 2) of Integer;
19
20    -- NOTE: using an array type fails as well:
21    --
22    -- type Private_Constrained_Array
23    --   (L : Positive) is
24    --     Integer_Array (1 .. L);
25
26 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 59: private_arrays.ads

```
1 package Private_Arrays is
2
3     type Private_Constrained_Array
4       (L : Positive) is private;
```

(continues on next page)

(continued from previous page)

```

5
6 private
7
8   type Integer_Array is
9     array (Positive range <>) of Integer;
10
11  type Private_Constrained_Array
12    (L : Positive) is
13    record
14      Arr : Integer_Array (1 .. 2);
15    end record;
16
17 end Private_Arrays;

```

Listing 60: declare_private_array.adb

```

1 with Private_Arrays; use Private_Arrays;
2
3 procedure Declare_Private_Array is
4   Arr : Private_Constrained_Array (5);
5 begin
6   null;
7 end Declare_Private_Array;

```

Code block metadata

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

Build output

```

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

```

Now, the code compiles fine — but we had to use a record type in the full view to make it work.

Another solution is to make the private type indefinite. In this case, the client's partial view would be consistent with a completion as an indefinite type in the private part:

Listing 61: private_arrays.ads

```

1 package Private_Arrays is
2
3   type Private_Constrained_Array (<>) is
4     private;
5
6   function Init
7     (L : Positive)
8     return Private_Constrained_Array;
9
10 private
11
12   type Private_Constrained_Array is
13     array (Positive range <>) of Integer;
14
15 end Private_Arrays;

```

Listing 62: private_arrays.adb

```
1 package body Private_Arrays is
2
3     function Init
4         (L : Positive)
5         return Private_Constrained_Array
6     is
7         PCA : Private_Constrained_Array (1 .. L);
8     begin
9         return PCA;
10    end Init;
11
12 end Private_Arrays;
```

Listing 63: declare_private_array.adb

```
1 with Private_Arrays; use Private_Arrays;
2
3 procedure Declare_Private_Array is
4     Arr : Private_Constrained_Array := Init (5);
5 begin
6     null;
7 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.

1.6 Type conversion

An important operation when dealing with objects of different types is type conversion, which we already discussed in the [Introduction to Ada course](#)¹⁴. In fact, we can convert an object `Obj_X` of an *operand* type `X` to a similar, closely related *target* type `Y` by simply indicating the target type: `Y (Obj_X)`. In this section, we discuss type conversions for different kinds of types.

Ada distinguishes between two kinds of conversion: value conversion and view conversion. The main difference is the way how the operand (argument) of the conversion is evaluated:

- in a value conversion, the operand is evaluated as an *expression* (page 319);
- in a view conversion, the operand is evaluated as a name.

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

In the Ada Reference Manual

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

- 4.6 Type Conversions¹⁵

1.6.1 Value conversion

Value conversions are possible for various types. In this section, we see some examples, starting with types derived from scalar types up to array conversions.

Root and derived types

Let's start with the conversion between a scalar type and its derived types. For example, we can convert back-and-forth between the **Integer** type and the derived **Int** type:

Listing 64: custom_integers.ads

```

1 package Custom_Integers is
2
3   type Int is new Integer
4     with Dynamic_Predicate => Int /= 0;
5
6   function Double (I : Integer)
7     return Integer is
8     (I * 2);
9
10 end Custom_Integers;
```

Listing 65: show_conversion.adb

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Custom_Integers; use Custom_Integers;
3
4 procedure Show_Conversion is
5   Int_Var      : Int      := 1;
6   Integer_Var  : Integer := 2;
7 begin
8   -- Int to Integer conversion
9   Integer_Var := Integer (Int_Var);
10
11   Put_Line ("Integer_Var : "
12           & Integer_Var'Image);
13
14   -- Int to Integer conversion
15   -- as an actual parameter
16   Integer_Var := Double (Integer (Int_Var));
17
18   Put_Line ("Integer_Var : "
19           & Integer_Var'Image);
20
21   -- Integer to Int conversion
22   -- using an expression
23   Int_Var := Int (Integer_Var * 2);
24
25   Put_Line ("Int_Var :      "
26           & Int_Var'Image);
27 end Show_Conversion;
```

Code block metadata

¹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-6.html>

```
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 66: `show_conversion.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Custom_Integers; use Custom_Integers;
3
4 procedure Show_Conversion is
5     Int_Var      : Int;
6     Integer_Var  : Integer;
7 begin
8     Integer_Var := 0;
9     Int_Var     := Int (Integer_Var);
10
11     Put_Line ("Int_Var : "
12             & Int_Var'Image);
13 end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Root_Derived_Type_
↳Conversion
MD5: 4150cdfd4c1fed39fa1728a77fa599f
```

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 67: show_conversion.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Conversion is
4   F : Float := 1.0;
5   I : Integer := 2;
6 begin
7   I := Integer (F);
8
9   Put_Line ("I : "
10            & I'Image);
11
12   I := 4;
13   F := Float (I);
14
15   Put_Line ("F : "
16            & F'Image);
17 end Show_Conversion;

```

Code block metadata

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

Runtime output

```

I : 1
F : 4.00000E+00

```

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

Listing 68: fixed_point_defs.ads

```

1 package Fixed_Point_Defs is
2   S : constant := 32;
3   Exp : constant := 15;
4   D : constant := 2.0 ** (-S + Exp + 1);
5
6   type TQ15_31 is delta D
7     range -1.0 * 2.0 ** Exp ..
8           1.0 * 2.0 ** Exp - D;
9
10  pragma Assert (TQ15_31'Size = S);
11 end Fixed_Point_Defs;

```

Listing 69: show_conversion.adb

```

1 with Fixed_Point_Defs; use Fixed_Point_Defs;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 procedure Show_Conversion is
5   F : Float;
6   FP : TQ15_31;
7   I : Integer;
8 begin
9   FP := TQ15_31 (10.25);

```

(continues on next page)

(continued from previous page)

```
10  I := Integer (FP);
11
12  Put_Line ("FP : "
13           & FP'Image);
14  Put_Line ("I : "
15           & I'Image);
16
17  I := 128;
18  FP := TQ15_31 (I);
19  F := Float (FP);
20
21  Put_Line ("FP : "
22           & FP'Image);
23  Put_Line ("F : "
24           & F'Image);
25  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 70: custom_enumerations.ads

```
1 package Custom_Enumerations is
2
3   type Priority is (Low, Mid, High);
4
5   type Important_Priority is new
6     Priority range Mid .. High;
7
8 end Custom_Enumerations;
```

Listing 71: show_conversion.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Custom_Enumerations; use Custom_Enumerations;
3
4 procedure Show_Conversion is
5   P : Priority := Low;
6   IP : Important_Priority := High;
7 begin
8   P := Priority (IP);
```

(continues on next page)

(continued from previous page)

```

9
10   Put_Line ("P: "
11             & P'Image);
12
13   P := Mid;
14   IP := Important_Priority (P);
15
16   Put_Line ("IP: "
17             & IP'Image);
18 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 72: show_conversion.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Custom_Enumerations; use Custom_Enumerations;
3
4 procedure Show_Conversion is
5   P : Priority;
6   IP : Important_Priority;
7 begin
8   P := Low;
9   IP := Important_Priority (P);
10
11   Put_Line ("IP: "
12             & IP'Image);
13 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 73: `custom_arrays.ads`

```
1 package Custom_Arrays is
2
3     type Integer_Array is
4       array (Positive range <>) of Integer;
5
6     type Derived_Integer_Array is new
7       Integer_Array;
8
9 end Custom_Arrays;
```

Listing 74: `show_conversion.adb`

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4 with Custom_Arrays; use Custom_Arrays;
5
6 procedure Show_Conversion is
7     subtype Common_Range is Positive range 1 .. 3;
8
9     AI : Integer_Array (Common_Range);
10    AI_D : Derived_Integer_Array (Common_Range);
11 begin
12    AI_D := [1, 2, 3];
13    AI := Integer_Array (AI_D);
14
15    Put_Line ("AI: "
16             & AI'Image);
17
18    AI := [4, 5, 6];
19    AI_D := Derived_Integer_Array (AI);
20
21    Put_Line ("AI_D: "
22             & AI_D'Image);
23 end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Array_Type_
↳ Conversion
MD5: e0a9fd519685b418a06dc7a3d0dab1c0
```

Runtime output

```
AI:
[ 1,  2,  3]
AI_D:
[ 4,  5,  6]
```

Note that both arrays must have the same number of components in order for the conversion to be successful. (Sliding is fine, though.) In this example, both arrays have the same range: `Common_Range`.

We can also convert between array types that aren't derived one from the other. As long

as the components and the index subtypes are of the same type, the conversion between those types is possible. To be more precise, these are the requirements for the array conversion to be accepted:

- The component types must be the same type.
- The index types (or subtypes) must be the same or, at least, convertible.
- The dimensionality of the arrays must be the same.
- The bounds must be compatible (but not necessarily equal).

Converting between different array types can be very handy, especially when we're dealing with array types that were not declared in the same package. For example:

Listing 75: custom_arrays_1.ads

```
1 package Custom_Arrays_1 is
2
3   type Integer_Array_1 is
4     array (Positive range <>) of Integer;
5
6   type Float_Array_1 is
7     array (Positive range <>) of Float;
8
9 end Custom_Arrays_1;
```

Listing 76: custom_arrays_2.ads

```
1 package Custom_Arrays_2 is
2
3   type Integer_Array_2 is
4     array (Positive range <>) of Integer;
5
6   type Float_Array_2 is
7     array (Positive range <>) of Float;
8
9 end Custom_Arrays_2;
```

Listing 77: show_conversion.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO;      use Ada.Text_IO;
4 with Custom_Arrays_1; use Custom_Arrays_1;
5 with Custom_Arrays_2; use Custom_Arrays_2;
6
7 procedure Show_Conversion is
8   subtype Common_Range is Positive range 1 .. 3;
9
10  AI_1 : Integer_Array_1 (Common_Range);
11  AI_2 : Integer_Array_2 (Common_Range);
12  AF_1 : Float_Array_1 (Common_Range);
13  AF_2 : Float_Array_2 (Common_Range);
14 begin
15  AI_2 := [1, 2, 3];
16  AI_1 := Integer_Array_1 (AI_2);
17
18  Put_Line ("AI_1: "
19           & AI_1'Image);
20
21  AI_1 := [4, 5, 6];
22  AI_2 := Integer_Array_2 (AI_1);
```

(continues on next page)

(continued from previous page)

```
23
24 Put_Line ("AI_2: "
25           & AI_2'Image);
26
27 -- ERROR: Cannot convert arrays whose
28 --       components have different types:
29 --
30 -- AF_1 := Float_Array_1 (AI_1);
31 --
32 -- Instead, use array aggregate where each
33 -- component is converted from integer to
34 -- float:
35 --
36 AF_1 := [for I in AF_1'Range =>
37         Float (AI_1 (I))];
38
39 Put_Line ("AF_1: "
40           & AF_1'Image);
41
42 AF_2 := Float_Array_2 (AF_1);
43
44 Put_Line ("AF_2: "
45           & AF_2'Image);
46 end Show_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Array_Type_
↳Conversion
MD5: 5c62f7cf94eedf8b0b223c24a83cc8d3
```

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 182).)

We may also encounter array types originating from the instantiation of generic packages. In this case as well, we can use array conversions. Consider the following generic package:

Listing 78: custom_arrays.ads

```

1 generic
2   type T is private;
3 package Custom_Arrays is
4   type T_Array is
5     array (Positive range <>) of T;
6 end Custom_Arrays;
```

Code block metadata

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

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

Listing 79: show_conversion.adb

```

1 pragma Ada_2022;
2
3 with Ada.Text_IO;   use Ada.Text_IO;
4 with Custom_Arrays;
5
6 procedure Show_Conversion is
7   package CA_Int_1 is
8     new Custom_Arrays (T => Integer);
9   package CA_Int_2 is
10    new Custom_Arrays (T => Integer);
11
12    subtype Common_Range is Positive range 1 .. 3;
13
14    AI_1 : CA_Int_1.T_Array (Common_Range);
15    AI_2 : CA_Int_2.T_Array (Common_Range);
16 begin
17   AI_2 := [1, 2, 3];
18   AI_1 := CA_Int_1.T_Array (AI_2);
19
20   Put_Line ("AI_1: "
21            & AI_1'Image);
22
23   AI_1 := [4, 5, 6];
24   AI_2 := CA_Int_2.T_Array (AI_1);
25
26   Put_Line ("AI_2: "
27            & AI_2'Image);
28 end Show_Conversion;
```

Code block metadata

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

Runtime output

```

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

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

has a `T_Array` type. Even though these `T_Array` types have the same name, they're actually completely unrelated types. However, we can still convert between them in the same way as we did in the previous code examples.

1.6.2 View conversion

As mentioned before, view conversions just allow names to be converted. Thus, we cannot use expressions in this case.

Note that a view conversion never changes the value during the conversion. We could say that a view conversion is simply making us *view* an object from a different angle. The object itself is still the same for both the original and the target types.

For example, consider this package:

Listing 80: `some_tagged_types.ads`

```
1 package Some_Tagged_Types is
2
3   type T is tagged record
4     A : Integer;
5   end record;
6
7   type T_Derived is new T with record
8     B : Float;
9   end record;
10
11   Obj : T_Derived;
12
13 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 81: points.ads

```

1 package Points is
2
3   type Point_1D is tagged record
4     X : Float;
5   end record;
6
7   procedure Display (P : Point_1D);
8
9   type Point_2D is new Point_1D with record
10    Y : Float;
11  end record;
12
13  procedure Display (P : Point_2D);
14
15  type Point_3D is new Point_2D with record
16    Z : Float;
17  end record;
18
19  procedure Display (P : Point_3D);
20
21 end Points;
```

Listing 82: points.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_1D) is
6   begin
7     Put_Line ("(X => " & P.X'Image & ")");
8   end Display;
9
10  procedure Display (P : Point_2D) is
11  begin
12    Put_Line ("(X => " & P.X'Image
13              & ", Y => " & P.Y'Image & ")");
14  end Display;
15
16  procedure Display (P : Point_3D) is
17  begin
18    Put_Line ("(X => " & P.X'Image
19              & ", Y => " & P.Y'Image
20              & ", Z => " & P.Z'Image & ")");
21  end Display;
22
23 end Points;
```

Code block metadata

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

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

Listing 83: show_conversion.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Points;      use Points;
3
4 procedure Show_Conversion is
5   P_1D : Point_1D;
6   P_3D : Point_3D;
7 begin
8   P_3D := (X => 0.1, Y => 0.5, Z => 0.3);
9   P_1D := Point_1D (P_3D);
10
11  Put ("P_3D : ");
12  Display (P_3D);
13
14  Put ("P_1D : ");
15  Display (P_1D);
16 end Show_Conversion;
```

Code block metadata

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

Runtime output

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

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

View conversion of untagged types

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

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

Listing 84: double.ads

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

Listing 85: double.adb

```
1 procedure Double (X : in out Float) is
2 begin
3   X := X * 2.0;
4 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 86: show_conversion.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Double;
3
4 procedure Show_Conversion is
5     I : Integer;
6 begin
7     I := 2;
8     Put_Line ("I : "
9               & I'Image);
10
11     -- Calling Double with
12     -- Integer parameter:
13     Double (Float (I));
14     Put_Line ("I : "
15             & I'Image);
16 end Show_Conversion;

```

Code block metadata

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

Runtime output

```

I : 2
I : 4

```

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

Listing 87: show_conversion.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Double;
3
4 procedure Show_Conversion is
5     I : Integer;
6 begin
7     I := 2;
8     Put_Line ("I : "
9               & I'Image);
10
11     declare
12         F : Float := Float (I);
13     begin
14         Double (F);
15         I := Integer (F);
16     end;
17     Put_Line ("I : "
18             & I'Image);
19 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.

1.6.3 Implicit conversions

Implicit conversions are only possible when we have a type `T` and a subtype `S` related to the `T` type. For example:

Listing 88: `custom_integers.ads`

```
1 package Custom_Integers is
2
3   type Int is new Integer
4     with Dynamic_Predicate => Int /= 0;
5
6   subtype Sub_Int_1 is Integer
7     with Dynamic_Predicate => Sub_Int_1 /= 0;
8
9   subtype Sub_Int_2 is Sub_Int_1
10    with Dynamic_Predicate => Sub_Int_2 /= 1;
11
12 end Custom_Integers;
```

Listing 89: `show_conversion.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Custom_Integers; use Custom_Integers;
3
4 procedure Show_Conversion is
5   Int_Var      : Int;
6   Sub_Int_1_Var : Sub_Int_1;
7   Sub_Int_2_Var : Sub_Int_2;
8   Integer_Var  : Integer;
9 begin
10  Integer_Var := 5;
11  Int_Var     := Int (Integer_Var);
12
13  Put_Line ("Int_Var : "
14           & Int_Var'Image);
15
16  -- Implicit conversions:
17  -- no explicit conversion required!
18  Sub_Int_1_Var := Integer_Var;
19  Sub_Int_2_Var := Integer_Var;
20
21  Put_Line ("Sub_Int_1_Var : "
22           & Sub_Int_1_Var'Image);
23  Put_Line ("Sub_Int_2_Var : "
24           & Sub_Int_2_Var'Image);
25 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.

1.6.4 Conversion of other types

For other kinds of types, such as records, a direct conversion as we've seen so far isn't possible. In this case, we have to write a conversion function ourselves. A common convention in Ada is to name this function `To_Typename`. For example, if we want to convert from any type to **Integer** or **Float**, we implement the `To_Integer` and `To_Float` functions, respectively. (Obviously, because Ada supports subprogram overloading, we can have multiple `To_Typename` functions for different operand types.)

Let's see a code example:

Listing 90: custom_rec.ads

```
1 package Custom_Rec is
2
3     type Rec is record
4         X : Integer;
5     end record;
6
7     function To_Integer (R : Rec)
8         return Integer is
9         (R.X);
10
11 end Custom_Rec;
```

Listing 91: show_conversion.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Custom_Rec; use Custom_Rec;
3
4 procedure Show_Conversion is
5     R : Rec;
6     I : Integer;
7 begin
8     R := (X => 2);
9     I := To_Integer (R);
10
11     Put_Line ("I : " & I'Image);
12 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;
};
```

(continues on next page)

(continued from previous page)

```

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 92: custom_defs.ads

```

1 package Custom_Defs is
2
3   type T1 is private;
4
5   function Init (X : Float)

```

(continues on next page)

(continued from previous page)

```

6         return T1;
7
8     procedure Display (Obj : T1);
9
10    type T3 is private;
11
12    function Init (X, Y, Z : Float)
13        return T3;
14
15    function To_Float (Obj : T3)
16        return Float;
17
18    function To_T1 (Obj : T3)
19        return T1;
20
21    procedure Display (Obj : T3);
22
23 private
24     type T1 is record
25         X : Float;
26     end record;
27
28     function Init (X : Float)
29         return T1 is
30         (X => X);
31
32     type T3 is record
33         X, Y, Z : Float;
34     end record;
35
36     function Init (X, Y, Z : Float)
37         return T3 is
38         (X => X, Y => Y, Z => Z);
39
40 end Custom_Defs;

```

Listing 93: custom_defs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Defs is
4
5     procedure Display (Obj : T1) is
6     begin
7         Put_Line ("(X => "
8             & Obj.X'Image & ")");
9     end Display;
10
11    function To_Float (Obj : T3)
12        return Float is
13        ((Obj.X + Obj.Y + Obj.Z) / 3.0);
14
15    function To_T1 (Obj : T3)
16        return T1 is
17        (Init (To_Float (Obj)));
18
19    procedure Display (Obj : T3) is
20    begin
21        Put_Line ("(X => " & Obj.X'Image
22            & ", Y => " & Obj.Y'Image
23            & ", Z => " & Obj.Z'Image

```

(continues on next page)

(continued from previous page)

```

24         & "));
25     end Display;
26
27 end Custom_Defs;
```

Listing 94: show_conversion.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Custom_Defs; use Custom_Defs;
3
4  procedure Show_Conversion is
5      T_3 : constant T3 := Init (0.5, 0.4, 0.6);
6      T_1 :          T1 := Init (0.0);
7      F   : Float;
8  begin
9      -- Explicit conversion from
10     -- T3 to Float type
11     F := To_Float (T_3);
12
13     Put_Line ("F : " & F'Image);
14
15     -- Explicit conversion from
16     -- T3 to T1 type
17     T_1 := To_T1 (T_3);
18
19     Put ("T_1 : ");
20     Display (T_1);
21 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.)

1.7 Qualified Expressions

We already saw qualified expressions in the [Introduction to Ada¹⁶](#) course. As mentioned there, a qualified expression specifies the exact type or subtype that the target expression will be resolved to, and it can be either any expression in parentheses, or an aggregate:

Listing 95: simple_integers.ads

```

1  package Simple_Integers is
2
```

(continues on next page)

¹⁶ https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_types.html#intro-ada-qualified-expressions

(continued from previous page)

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

Listing 96: show_qualified_expressions.adb

```
1  with Simple_Integers; use Simple_Integers;
2
3  procedure Show_Qualified_Expressions is
4     I : Int;
5  begin
6     -- Using qualified expression Int'(N)
7     I := Int'(0);
8  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 Expressions¹⁷](#)

1.7.1 Verifying subtypes

Note: This feature was introduced in Ada 2022.

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

Listing 97: show_qualified_expressions.adb

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

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example
MD5: 3c4ab8ad7bf75ae029047f673aa15d70

Build output

¹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-7.html>

```
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.)

1.8 Default initial values

In the [Introduction to Ada course](#)¹⁸, we've seen that record components can have default values. For example:

Listing 98: defaults.ads

```
1 package Defaults is  
2  
3   type R is record  
4     X : Positive := 1;  
5     Y : Positive := 10;  
6   end record;  
7  
8 end Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_1  
MD5: e230be602cbb24a854e71c8176c7148c
```

In this section, we'll extend the concept of default values to other kinds of type declarations, such as scalar types and arrays.

To assign a default value for a scalar type declaration — such as an enumeration and a new integer —, we use the `Default_Value` aspect:

Listing 99: defaults.ads

```
1 package Defaults is  
2  
3   type E is (E1, E2, E3)  
4     with Default_Value => E1;  
5  
6   type T is new Integer  
7     with Default_Value => -1;  
8  
9 end Defaults;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_2  
MD5: e6cd8261b099278ceeb5fda91d318f6e
```

¹⁸ <https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-default-values>

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

Listing 100: defaults.ads

```
1 package Defaults is
2
3     subtype T is Integer
4         with Default_Value => -1;
5     -- ERROR!!
6
7 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 101: defaults.ads

```
1 package Defaults is
2
3     type Arr is
4         array (Positive range <>) of Integer
5         with Default_Component_Value => -1;
6
7 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 102: defaults.ads

```
1 package Defaults is
2
3     type E is (E1, E2, E3)
4         with Default_Value => E1;
5
6     type T is new Integer
7         with Default_Value => -1;
8
9     -- We cannot specify default
10    -- values for subtypes:
11    --
12    -- subtype T is Integer
13    --     with Default_Value => -1;
14
15    type R is record
16        X : Positive := 1;
17        Y : Positive := 10;
18    end record;
19
```

(continues on next page)

(continued from previous page)

```

20  type Arr is
21     array (Positive range <>) of Integer
22     with Default_Component_Value => -1;
23
24  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 103: use_defaults.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Defaults; use Defaults;
3
4  procedure Use_Defaults is
5     E1 : E;
6     T1 : T;
7     R1 : R;
8     A1 : Arr (1 .. 5);
9  begin
10     Put_Line ("Enumeration: "
11              & E'Image (E1));
12     Put_Line ("Integer type: "
13              & T'Image (T1));
14     Put_Line ("Record type: "
15              & Positive'Image (R1.X)
16              & ", "
17              & Positive'Image (R1.Y));
18
19     Put ("Array type: ");
20     for V of A1 loop
21         Put (Integer'Image (V) & " ");
22     end loop;
23     New_Line;
24  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](#)¹⁹

¹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-5.html>

- 3.6 Array Types²⁰
-

1.9 Deferred Constants

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

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

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

Listing 104: deferred_constants.ads

```
1 package Deferred_Constants is
2
3   type Speed is new Long_Float;
4
5   Light : constant Speed;
6   --   ^ deferred constant declaration
7
8 private
9
10  Light : constant Speed := 299_792_458.0;
11  --   ^ full constant declaration
12
13 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:

Listing 105: constants.c

```
1 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 106: deferred_constants.ads

```
1 package Deferred_Constants is
2
```

(continues on next page)

²⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

(continued from previous page)

```

3  type Speed is new Long_Float;
4
5  Light : constant Speed with
6      Import, Convention => C;
7  -- ^^^^ deferred constant
8  --      declaration; imported
9  --      from C file
10
11 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 107: `deferred_constants.ads`

```

1  package Deferred_Constants is
2
3      type Speed is new Long_Float;
4
5      subtype Positive_Speed is
6          Speed range 0.0 .. Speed'Last;
7
8      Light : constant Speed;
9      -- ^ deferred constant declaration
10
11 private
12
13     Light : constant Positive_Speed :=
14         299_792_458.0;
15     -- ^ full constant declaration
16     --     using a subtype
17
18 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 108: deferred_constants.ads

```
1 package Deferred_Constants is
2
3     type Speed is new Long_Float;
4
5     Light : constant Speed;
6     --      ^ deferred constant declaration
7
8 private
9
10    function Calculate_Light return Speed is
11        (299_792_458.0);
12
13    Light : constant Speed := Calculate_Light;
14    --      ^ full constant declaration
15    --      calling a private function
16
17 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](#)²¹

1.10 User-defined literals

Note: This feature was introduced in Ada 2022.

Any type definition has a kind of literal associated with it. For example, integer types are associated with integer literals. Therefore, we can initialize an object of integer type with an integer literal:

Listing 109: simple_integer_literal.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Integer_Literal is
4     V : Integer;
5 begin
6     V := 10;
7
8     Put_Line (Integer'Image (V));
9 end Simple_Integer_Literal;
```

Code block metadata

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

Project: Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.Simple_Enumeration_Literal
 MD5: 9f65e7c319be2b292dc1fdf02dd7c4b4

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 110: simple_enumeration.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Enumeration is
4     type Activation_State is (Unknown, Off, On);
5
6     S : Activation_State;
7 begin
8     S := On;
9     Put_Line (Activation_State'Image (S));
10 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 111: activation_states.ads

```
1 package Activation_States is
2
3     type Activation_State is (Unknown, Off, On)
4       with Integer_Literal =>
5         Integer_To_Activation_State;
6
7     function Integer_To_Activation_State
8       (S : String)
9       return Activation_State;
10
11 end Activation_States;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.User_Defined_↵Literals
MD5: 67b6d96f049ab6cde962aefda96bffca

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

```
S : Activation_State := 1;
```

Note that we have a string parameter in the declaration of the Integer_To_Activation_State function, even though the function itself is only used to convert integer literals (but not string literals) to the Activation_State type. It's our job to process that string parameter in the implementation of the Integer_To_Activation_State function and convert it to an integer value — using `Integer'Value`, for example:

Listing 112: activation_states.adb

```
1 package body Activation_States is
2
3     function Integer_To_Activation_State
4       (S : String)
5       return Activation_State is
6     begin
7       case Integer'Value (S) is
8         when 0    => return Off;
9         when 1    => return On;
10        when others => return Unknown;
11      end case;
12    end Integer_To_Activation_State;
13
14 end Activation_States;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.User_Defined_↵Literals
MD5: 104a835915b93ea3b860bce03fd709a3

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

Listing 113: activation_states.ads

```
1 package Activation_States is
2
3     type Activation_State is (Unknown, Off, On)
```

(continues on next page)

(continued from previous page)

```

4     with String_Literal =>
5         To_Activation_State,
6         Integer_Literal =>
7             Integer_To_Activation_State,
8         Real_Literal =>
9             Real_To_Activation_State;
10
11    function To_Activation_State
12        (S : Wide_Wide_String)
13        return Activation_State;
14
15    function Integer_To_Activation_State
16        (S : String)
17        return Activation_State;
18
19    function Real_To_Activation_State
20        (S : String)
21        return Activation_State;
22
23    end Activation_States;

```

Listing 114: activation_states.adb

```

1    package body Activation_States is
2
3        function To_Activation_State
4            (S : Wide_Wide_String)
5            return Activation_State
6        is
7        begin
8            if S = "Off" then
9                return Off;
10           elsif S = "On" then
11               return On;
12           else
13               return Unknown;
14           end if;
15        end To_Activation_State;
16
17        function Integer_To_Activation_State
18            (S : String)
19            return Activation_State
20        is
21        begin
22            case Integer'Value (S) is
23                when 0      => return Off;
24                when 1      => return On;
25                when others => return Unknown;
26            end case;
27        end Integer_To_Activation_State;
28
29        function Real_To_Activation_State
30            (S : String)
31            return Activation_State
32        is
33            V : constant Float := Float'Value (S);
34        begin
35            if V < 0.0 then
36                return Unknown;
37            elsif V < 1.0 then
38                return Off;

```

(continues on next page)

(continued from previous page)

```
39     else
40         return On;
41     end if;
42 end Real_To_Activation_State;
43
44 end Activation_States;
```

Listing 115: activation_examples.adb

```
1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Activation_States; use Activation_States;
3
4  procedure Activation_Examples is
5      S : Activation_State;
6  begin
7      S := "Off";
8      Put_Line ("String: Off => "
9                & Activation_State'Image (S));
10
11     S := 1;
12     Put_Line ("Integer: 1  => "
13              & Activation_State'Image (S));
14
15     S := 1.5;
16     Put_Line ("Real:    1.5 => "
17              & Activation_State'Image (S));
18 end Activation_Examples;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.Activation_States
MD5: 186b7b898e4c16bfd8dcd683e8f0379d

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 116: `using_string_literal.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Activation_States; use Activation_States;
3
4 procedure Using_String_Literal is
5     S1 : constant Activation_State := On;
6     S2 : constant Activation_State := "On";
7 begin
8     Put_Line (Activation_State'Image (S1));
9     Put_Line (Activation_State'Image (S2));
10 end Using_String_Literal;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.Activation_States`
MD5: `6ca6aa79b88058801688fc2dfb186091`

Runtime output

```
ON
ON
```

Note we need to be very careful when designing conversion functions. For example, the use of string literals may limit the kind of checks that we can do. Consider the following misspelling of the `Off` literal:

Listing 117: `misspelling_example.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Activation_States; use Activation_States;
3
4 procedure Misspelling_Example is
5     S : constant Activation_State :=
6         Offf;
7     -- ^ Error: Off is misspelled.
8 begin
9     Put_Line (Activation_State'Image (S));
10 end Misspelling_Example;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.Activation_States`
MD5: `ebc1036a58e460a9212106606461b014`

Build output

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

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

Listing 118: misspelling_example.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Activation_States; use Activation_States;
3
4 procedure Misspelling_Example is
5     S : constant Activation_State :=
6         "Offf";
7     --     ^ Error: Off is misspelled.
8 begin
9     Put_Line (Activation_State'Image (S));
10 end Misspelling_Example;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.Activation_States
MD5: 99f74c67712a9b55c146b9d57405e47f

Runtime output

UNKNOWN

Here, our implementation of `To_Activation_State` simply returns `Unknown`. In some cases, this might be exactly the behavior that we want. However, let's assume that we'd prefer better error handling instead. In this case, we could change the implementation of `To_Activation_State` to check all literals that we want to allow, and indicate an error otherwise — by raising an exception, for example. Alternatively, we could specify this in the preconditions of the conversion function:

```
function To_Activation_State
(S : Wide_Wide_String)
return Activation_State
with Pre => S = "Off" or
           S = "On" or
           S = "Unknown";
```

In this case, the precondition explicitly indicates which string literals are allowed for the `To_Activation_State` type.

User-defined literals can also be used for more complex types, such as records. For example:

Listing 119: silly_records.ads

```
1 package Silly_Records is
2
3     type Silly is record
4         X : Integer;
5         Y : Float;
6     end record
7     with String_Literal => To_Silly;
8
9     function To_Silly (S : Wide_Wide_String)
10        return Silly;
11 end Silly_Records;
```

Listing 120: silly_records.adb

```
1 package body Silly_Records is
2
```

(continues on next page)

(continued from previous page)

```

3  function To_Silly (S : Wide_Wide_String)
4      return Silly
5  is
6  begin
7      if S = "Magic" then
8          return (X => 42, Y => 42.0);
9      else
10         return (X => 0, Y => 0.0);
11     end if;
12 end To_Silly;
13
14 end Silly_Records;
```

Listing 121: silly_magic.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Silly_Records; use Silly_Records;
3
4  procedure Silly_Magic is
5      R1 : Silly;
6  begin
7      R1 := "Magic";
8      Put_Line (R1.X'Image & ", " & R1.Y'Image);
9  end Silly_Magic;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User-Defined_Literals.Record_
↳ Literals

MD5: 2a077045f058a8d5c09c43f66fc128be

Runtime output

```
42, 4.20000E+01
```

In this example, when we initialize an object of Silly type with a string, its components are:

- set to 42 when using the "Magic" string; or
- simply set to zero when using any other string.

Obviously, this example isn't particularly useful. However, the goal is to show that this approach is useful for more complex types where a string literal (or a numeric literal) might simplify handling those types. User-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 Literals²²](#)

²² <http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html>

TYPES AND REPRESENTATION

2.1 Enumeration Representation Clauses

We have talked about the internal code of an enumeration *in another section* (page 22). We may change this internal code by using a representation clause, which has the following format:

```
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

```
1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     for Day use (Mon => 2#00000001#,
8                 Tue => 2#00000010#,
9                 Wed => 2#00000100#,
10                Thu => 2#00001000#,
11                Fri => 2#00010000#,
12                Sat => 2#00100000#,
13                Sun => 2#01000000#);
14
15 end Days;
```

Listing 2: show_days.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Days;        use Days;
3
4 procedure Show_Days is
5 begin
6     for D in Day loop
7         Put_Line (Day'Image (D)
8                 & " position = "
9                 & Integer'Image (Day'Pos (D)));
10        Put_Line (Day'Image (D)
11                & " internal code = "
```

(continues on next page)

(continued from previous page)

```
12         & Integer'Image
13           (Day'Enum_Rep (D)));
14     end loop;
15 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

```
1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     for Day use (Mon => 5,
8                 Tue => 9,
9                 Wed => 42,
10                Thu => 49,
11                Fri => 50,
12                Sat => 66,
13                Sun => 99);
14
15 end Days;
```

2.2 Data Representation

This section provides a glimpse on attributes and aspects used for data representation. They are usually used for embedded applications because of strict requirements that are often found there. Therefore, unless you have very specific requirements for your application, in most cases, you won't need them. However, you should at least have a rudimentary understanding of them. To read a thorough overview on this topic, please refer to the [Introduction to Embedded Systems Programming](#)²³ course.

In the Ada Reference Manual

- [13.2 Packed Types](#)²⁴
- [13.3 Operational and Representation Attributes](#)²⁵
- [13.5.3 Bit Ordering](#)²⁶

2.2.1 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.

²³ https://learn.adacore.com/courses/intro-to-embedded-sys-prog/chapters/low_level_programming.html#intro-embedded-sys-prog-low-level-programming

²⁴ <http://www.ada-auth.org/standards/22rm/html/RM-13-2.html>

²⁵ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

²⁶ <http://www.ada-auth.org/standards/22rm/html/RM-13-5-3.html>

Size attribute and aspect

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

Listing 4: custom_types.ads

```
1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_S32 is range 0 .. 127
6         with Size => 32;
7
8 end Custom_Types;
```

Listing 5: show_sizes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Sizes is
6     V1 : UInt_7;
7     V2 : UInt_7_S32;
8 begin
9     Put_Line ("UInt_7'Size:      "
10             & UInt_7'Size'Image);
11     Put_Line ("UInt_7'Object_Size:  "
12             & UInt_7'Object_Size'Image);
13     Put_Line ("V1'Size:          "
14             & V1'Size'Image);
15     New_Line;
16
17     Put_Line ("UInt_7_S32'Size:      "
18             & UInt_7_S32'Size'Image);
19     Put_Line ("UInt_7_S32'Object_Size:  "
20             & UInt_7_S32'Object_Size'Image);
21     Put_Line ("V2'Size:          "
22             & V2'Size'Image);
23 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
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!"

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`

```

1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_Array is
6         array (Positive range <>) of UInt_7;
7
8     type UInt_7_Array_Comp_32 is
9         array (Positive range <>) of UInt_7
10            with Component_Size => 32;
11
12 end Custom_Types;
    
```

Listing 7: show_sizes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Sizes is
6   Arr_1 : UInt_7_Array (1 .. 20);
7   Arr_2 : UInt_7_Array_Comp_32 (1 .. 20);
8 begin
9   Put_Line
10    ("UInt_7_Array'Size:           "
11     & UInt_7_Array'Size'Image);
12   Put_Line
13    ("UInt_7_Array'Object_Size:    "
14     & UInt_7_Array'Object_Size'Image);
15   Put_Line
16    ("UInt_7_Array'Component_Size: "
17     & UInt_7_Array'Component_Size'Image);
18   Put_Line
19    ("Arr_1'Component_Size:        "
20     & Arr_1'Component_Size'Image);
21   Put_Line
22    ("Arr_1'Size:                  "
23     & Arr_1'Size'Image);
24   New_Line;
25
26   Put_Line
27    ("UInt_7_Array_Comp_32'Object_Size:  "
28     & UInt_7_Array_Comp_32'Object_Size'Image);
29   Put_Line
30    ("UInt_7_Array_Comp_32'Object_Size:  "
31     & UInt_7_Array_Comp_32'Object_Size'Image);
32   Put_Line
33    ("UInt_7_Array_Comp_32'Component_Size: "
34     &
35     UInt_7_Array_Comp_32'Component_Size'Image);
36   Put_Line
37    ("Arr_2'Component_Size:          "
38     & Arr_2'Component_Size'Image);
39   Put_Line
40    ("Arr_2'Size:                   "
41     & Arr_2'Size'Image);
42   New_Line;
43 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
```

(continues on next page)

(continued from previous page)

```

UInt_7_Array'Component_Size:      8
Arr_1'Component_Size:             8
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).

Storage size

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

Listing 8: custom_types.ads

```
1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_Access is access UInt_7;
6
7 end Custom_Types;
```

Listing 9: show_sizes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 with Custom_Types; use Custom_Types;
5
6 procedure Show_Sizes is
7     AV1, AV2 : UInt_7_Access;
8 begin
9     Put_Line
10      ("UInt_7_Access'Storage_Size:      "
11       & UInt_7_Access'Storage_Size'Image);
12     Put_Line
13      ("UInt_7_Access'Storage_Size (bits):  "
14       & Integer'Image (UInt_7_Access'Storage_Size
15                        * System.Storage_Unit));
16
17     Put_Line
18      ("UInt_7'Size:                      "
19       & UInt_7'Size'Image);
20     Put_Line
21      ("UInt_7_Access'Size:                "
22       & UInt_7_Access'Size'Image);
23     Put_Line
24      ("UInt_7_Access'Object_Size:         "
25       & UInt_7_Access'Object_Size'Image);
26     Put_Line
27      ("AV1'Size:                          "
28       & AV1'Size'Image);
29     New_Line;
30
31     Put_Line ("Allocating AV1...");
32     AV1 := new UInt_7;
33     Put_Line ("Allocating AV2...");
34     AV2 := new UInt_7;
35     New_Line;
36
37     Put_Line
38      ("AV1.all'Size:                      "
39       & AV1.all'Size'Image);
40     New_Line;
41 end Show_Sizes;
```

Code block metadata

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

(continues on next page)

(continued from previous page)

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`

```

1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_Reserved_Access is access UInt_7
6         with Storage_Size => 8;
7
8 end Custom_Types;
    
```


Listing 11: show_sizes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 with Custom_Types; use Custom_Types;
5
6 procedure Show_Sizes is
7   RAV1, RAV2 : UInt_7_Reserved_Access;
8 begin
9   Put_Line
10    ("UInt_7_Reserved_Access'Storage_Size:      "
11     & UInt_7_Reserved_Access'Storage_Size'Image);
12
13   Put_Line
14    ("UInt_7_Reserved_Access'Storage_Size (bits): "
15     & Integer'Image
16     (UInt_7_Reserved_Access'Storage_Size
17      * System.Storage_Unit));
18
19   Put_Line
20    ("UInt_7_Reserved_Access'Size:              "
21     & UInt_7_Reserved_Access'Size'Image);
22   Put_Line
23    ("UInt_7_Reserved_Access'Object_Size:      "
24     & UInt_7_Reserved_Access'Object_Size'Image);
25   Put_Line
26    ("RAV1'Size:                                "
27     & RAV1'Size'Image);
28   New_Line;
29
30   Put_Line ("Allocating RAV1...");
31   RAV1 := new UInt_7;
32   Put_Line ("Allocating RAV2...");
33   RAV2 := new UInt_7;
34   New_Line;
35 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-pooisiz.adb:108 explicit raise
```

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

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

(continues on next page)

(continued from previous page)

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

Allocating RAV1...
Allocating RAV2...

raised STORAGE_ERROR : s-pooisiz.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.

2.2.2 Alignment

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

Listing 12: custom_types.ads

```
1 package Custom_Types is
2
3   type UInt_7 is range 0 .. 127;
4
5   type Aligned_UInt_7 is new UInt_7
6     with Alignment => 4;
7
8 end Custom_Types;
```

Listing 13: show_alignment.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Alignment is
6   V      : constant UInt_7      := 0;
7   Aligned_V : constant Aligned_UInt_7 := 0;
8 begin
9   Put_Line
10    ("UInt_7'Alignment:      "
```

(continues on next page)

(continued from previous page)

```
11     & UInt_7'Alignment'Image);
12 Put_Line
13     ("UInt_7'Size:           "
14     & UInt_7'Size'Image);
15 Put_Line
16     ("UInt_7'Object_Size:   "
17     & UInt_7'Object_Size'Image);
18 Put_Line
19     ("V'Alignment:         "
20     & V'Alignment'Image);
21 Put_Line
22     ("V'Size:              "
23     & V'Size'Image);
24 New_Line;
25
26 Put_Line
27     ("Aligned_UInt_7'Alignment:  "
28     & Aligned_UInt_7'Alignment'Image);
29 Put_Line
30     ("Aligned_UInt_7'Size:       "
31     & Aligned_UInt_7'Size'Image);
32 Put_Line
33     ("Aligned_UInt_7'Object_Size: "
34     & Aligned_UInt_7'Object_Size'Image);
35 Put_Line
36     ("Aligned_V'Alignment:      "
37     & Aligned_V'Alignment'Image);
38 Put_Line
39     ("Aligned_V'Size:          "
40     & Aligned_V'Size'Image);
41 New_Line;
42 end Show_Alignment;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↳Alignment
MD5: a2fea340559193c293ccaee226de2558

Runtime output

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

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

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

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

(continues on next page)

(continued from previous page)

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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Class_Alignment is
4
5   type Point_1D is tagged record
6     X : Integer;
7   end record;
8
9   type Point_2D is new Point_1D with record
10    Y : Integer;
11  end record
12    with Alignment => 16;
13
14  type Point_3D is new Point_2D with record
15    Z : Integer;
16  end record;
17
18 begin
19   Put_Line ("1D_Point'Alignment:      "
20     & Point_1D'Alignment'Image);
21   Put_Line ("1D_Point'Class'Alignment: "
22     & Point_1D'Class'Alignment'Image);
23   Put_Line ("2D_Point'Alignment:      "
24     & Point_2D'Alignment'Image);
25   Put_Line ("2D_Point'Class'Alignment: "
26     & Point_2D'Class'Alignment'Image);
27   Put_Line ("3D_Point'Alignment:      "
28     & Point_3D'Alignment'Image);
29   Put_Line ("3D_Point'Class'Alignment: "
30     & Point_3D'Class'Alignment'Image);
31 end Show_Class_Alignment;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.  
↳Class_Alignment  
MD5: 4eb28d59439d1eb86cd23fb08acd3493
```

Runtime output

```
1D_Point'Alignment:      8  
1D_Point'Class'Alignment: 8  
2D_Point'Alignment:     16  
2D_Point'Class'Alignment: 16  
3D_Point'Alignment:     16  
3D_Point'Class'Alignment: 16
```

2.2.3 Overlapping Storage

Algorithms can be designed to perform in-place or out-of-place processing. In other words, they can take advantage of the fact that input and output arrays share the same storage space or not.

We can use the `Has_Same_Storage` and the `Overlaps_Storage` attributes to retrieve more information about how the storage space of two objects related to each other:

- the `Has_Same_Storage` attribute indicates whether two objects have the exact same storage.
 - A typical example is when both objects are exactly the same, so they obviously share the same storage. For example, for array A, `A'Has_Same_Storage (A)` is always **True**.
- the `Overlaps_Storage` attribute indicates whether two objects have at least one bit in common.
 - Note that, if two objects have the same storage, this implies that their storage also overlaps. In other words, `A'Has_Same_Storage (B) = True` implies that `A'Overlaps_Storage (B) = True`.

Let's look at this example:

Listing 15: `int_array_processing.ads`

```
1 package Int_Array_Processing is  
2  
3   type Int_Array is  
4     array (Positive range <>) of Integer;  
5  
6   procedure Show_Storage (X : Int_Array;  
7                         Y : Int_Array);  
8  
9   procedure Process (X : Int_Array;  
10                  Y : out Int_Array);  
11  
12 end Int_Array_Processing;
```

Listing 16: `int_array_processing.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 package body Int_Array_Processing is  
4
```

(continues on next page)

(continued from previous page)

```

5  procedure Show_Storage (X : Int_Array;
6                          Y : Int_Array) is
7  begin
8      if X'Has_Same_Storage (Y) then
9          Put_Line
10         ("Info: X and Y have the same storage.");
11     else
12         Put_Line
13         ("Info: X and Y don't have"
14          & "the same storage.");
15     end if;
16     if X'Overlaps_Storage (Y) then
17         Put_Line
18         ("Info: X and Y overlap.");
19     else
20         Put_Line
21         ("Info: X and Y don't overlap.");
22     end if;
23 end Show_Storage;
24
25 procedure Process (X : Int_Array;
26                  Y : out Int_Array) is
27 begin
28     Put_Line ("==== PROCESS ====");
29     Show_Storage (X, Y);
30
31     if X'Has_Same_Storage (Y) then
32         Put_Line ("In-place processing...");
33     else
34         if not X'Overlaps_Storage (Y) then
35             Put_Line
36             ("Out-of-place processing...");
37         else
38             Put_Line
39             ("Cannot process "
40              & "overlapping arrays...");
41         end if;
42     end if;
43     New_Line;
44 end Process;
45
46 end Int_Array_Processing;

```

Listing 17: main.adb

```

1  with Int_Array_Processing;
2  use Int_Array_Processing;
3
4  procedure Main is
5      A : Int_Array (1 .. 20) := (others => 3);
6      B : Int_Array (1 .. 20) := (others => 4);
7  begin
8      Process (A, A);
9      -- In-place processing:
10     -- sharing the exact same storage
11
12     Process (A (1 .. 10), A (10 .. 20));
13     -- Overlapping one component: A (10)
14
15     Process (A (1 .. 10), A (11 .. 20));
16     -- Out-of-place processing:

```

(continues on next page)

(continued from previous page)

```
17  -- same array, but not sharing any storage
18
19  Process (A, B);
20  -- Out-of-place processing:
21  -- two different arrays
22 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.

2.2.4 Packed Representation

As we've seen previously, the minimum number of bits required to represent a data type might be less than the actual number of bits used to store an object of that same type. We've seen an example where `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

```

1 package Flag_Definitions is
2
3     type Flags is
4         array (Positive range <>) of Boolean;
5
6 end Flag_Definitions;
```

Listing 19: show_flags.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Flag_Definitions; use Flag_Definitions;
3
4 procedure Show_Flags is
5     Flags_1 : Flags (1 .. 8);
6 begin
7     Put_Line ("Boolean'Size: "
8             & Boolean'Size'Image);
9     Put_Line ("Boolean'Object_Size: "
10            & Boolean'Object_Size'Image);
11    Put_Line ("Flags_1'Size: "
12            & Flags_1'Size'Image);
13    Put_Line ("Flags_1'Component_Size: "
14            & Flags_1'Component_Size'Image);
15 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

```
1 package Flag_Definitions is
2
3     type Flags is
4         array (Positive range <>) of Boolean;
5
6     type Packed_Flags is
7         array (Positive range <>) of Boolean
8         with Pack;
9
10 end Flag_Definitions;
```

Listing 21: show_packed_flags.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Flag_Definitions; use Flag_Definitions;
3
4 procedure Show_Packed_Flags is
5     Flags_1 : Flags (1 .. 8);
6     Flags_2 : Packed_Flags (1 .. 8);
7 begin
8     Put_Line ("Boolean'Size:      "
9         & Boolean'Size'Image);
10    Put_Line ("Boolean'Object_Size:  "
11        & Boolean'Object_Size'Image);
12    Put_Line ("Flags_1'Size:         "
13        & Flags_1'Size'Image);
14    Put_Line ("Flags_1'Component_Size: "
15        & Flags_1'Component_Size'Image);
16    Put_Line ("Flags_2'Size:         "
17        & Flags_2'Size'Image);
18    Put_Line ("Flags_2'Component_Size: "
19        & Flags_2'Component_Size'Image);
20 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

```
1 package Flag_Definitions is
2
3     type Flags is
4         array (Positive range <>) of Boolean;
5
6     type Packed_Flags is
7         array (Positive range <>) of Boolean
8         with Pack;
9
10    Default_Flags : constant Flags :=
11        (True, True, False, True,
12         False, False, True, True);
13
14 end Flag_Definitions;
```

Listing 23: show_flag_conversion.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Flag_Definitions; use Flag_Definitions;
3
4 procedure Show_Flag_Conversion is
5   Flags_1 : Flags (1 .. 8);
6   Flags_2 : Packed_Flags (1 .. 8);
7 begin
8   Flags_1 := Default_Flags;
9   Flags_2 := Packed_Flags (Flags_1);
10
11   for I in Flags_2'Range loop
12     Put_Line (I'Image & ": "
13              & Flags_1 (I)'Image & ", "
14              & Flags_2 (I)'Image);
15   end loop;
16 end Show_Flag_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↳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 106).

Trade-offs

As indicated previously, when we're using a packed representation (vs. using a standard *unpacked* representation), we're trading off speed of access for less memory consumption. The following table summarizes this:

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.

2.3 Record Representation and storage clauses

In this section, we discuss how to use record representation clauses to specify how a record is represented in memory. Our goal is to provide a brief introduction into the topic. If you're interested in more details, you can find a thorough discussion about record representation clauses in the [Introduction to Embedded Systems Programming²⁷](#) 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

```

1 package P is
2
3   type R is record
4     A : Integer;
5     B : Integer;
6   end record;
7
8 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:

²⁷ https://learn.adacore.com/courses/intro-to-embedded-sys-prog/chapters/low_level_programming.html#intro-embedded-sys-prog-low-level-programming

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.

Listing 25: p.ads

```

1 package P is
2
3   type R is record
4     A : Integer;
5     B : Integer;
6   end record;
7
8   -- Representation clause for record R:
9   for R use record
10    A at 0 range 0 .. 31;
11    -- ^ starting memory position
12    B at 4 range 0 .. 31;
13    -- ^ first bit .. last bit
14  end record;
15
16 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
```

(continues on next page)

(continued from previous page)

```

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 Clauses²⁸](#)
-

2.3.1 Storage Place Attributes

We can retrieve information about the start position, and the first and last bits of a component by using the storage place attributes:

- `Position`, which retrieves the start position of a component;
- `First_Bit`, which retrieves the first bit of a component;
- `Last_Bit`, which retrieves the last bit of a component.

Note, however, that these attributes can only be used with actual records, and not with record types.

We can revisit the previous example and verify how the compiler represents the R type in memory:

Listing 26: p.ads

```

1 package P is
2
3   type R is record
4     A : Integer;
5     B : Integer;
6   end record;
7
8 end P;

```

²⁸ <http://www.ada-auth.org/standards/22rm/html/RM-13-5-1.html>

Listing 27: show_storage.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 with P;          use P;
5
6 procedure Show_Storage is
7   R1 : R;
8 begin
9   Put_Line ("R'Size:           "
10            & R'Size'Image);
11   Put_Line ("R'Object_Size:    "
12            & R'Object_Size'Image);
13   New_Line;
14
15   Put_Line ("System.Storage_Unit: "
16            & System.Storage_Unit'Image);
17   New_Line;
18
19   Put_Line ("R1.A'Position  : "
20            & R1.A'Position'Image);
21   Put_Line ("R1.A'First_Bit : "
22            & R1.A'First_Bit'Image);
23   Put_Line ("R1.A'Last_Bit  : "
24            & R1.A'Last_Bit'Image);
25   New_Line;
26
27   Put_Line ("R1.B'Position  : "
28            & R1.B'Position'Image);
29   Put_Line ("R1.B'First_Bit : "
30            & R1.B'First_Bit'Image);
31   Put_Line ("R1.B'Last_Bit  : "
32            & R1.B'Last_Bit'Image);
33 end Show_Storage;
```

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 [-gnatvw]
```

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 Attributes²⁹

2.3.2 Using Representation Clauses

We can use representation clauses to change the way the compiler handles memory for a record type. For example, let's say we want to have an empty storage unit between components A and B. We can use a representation clause where we specify that component B starts at position #5 instead of #4, leaving an empty byte after component A and before component B:

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

```

1 package P is
2
3   type R is record
4     A : Integer;
5     B : Integer;
6   end record;
7
8   for R use record
9     A at 0 range 0 .. 31;
10    B at 5 range 0 .. 31;
11  end record;
12
13 end P;
```

Listing 29: show_empty_byte.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;          use P;
4
5 procedure Show_Empty_Byte is
6 begin
7   Put_Line ("R'Size:      "
8     & R'Size'Image);
9   Put_Line ("R'Object_Size: "
10    & R'Object_Size'Image);
11 end Show_Empty_Byte;
```

Code block metadata

²⁹ <http://www.ada-auth.org/standards/22rm/html/RM-13-5-2.html>


```
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

```
1 package P is
2
3     type R is record
4         A : Integer;
5         B : Integer;
6     end record
7     with Object_Size => 72;
8
9     for R use record
10        A at 0 range 0 .. 31;
11        B at 5 range 0 .. 31;
12    end record;
13
14 end P;
```

Listing 31: show_empty_byte.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;          use P;
4
5 procedure Show_Empty_Byte is
6 begin
7     Put_Line ("R'Size:          "
8             & R'Size'Image);
9     Put_Line ("R'Object_Size: "
10            & R'Object_Size'Image);
11 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.

2.3.3 Derived Types And Representation Clauses

In some cases, you might want to modify the memory representation of a record without impacting existing code. For example, you might want to use a record type that was declared in a package that you're not allowed to change. Also, you would like to modify its memory representation in your application. A nice strategy is to derive a type and use a representation clause for the derived type.

We can apply this strategy on our previous example. Let's say we would like to use record type R from package P in our application, but we're not allowed to modify package P — or the record type, for that matter. In this case, we could simply derive R as R_New and use a representation clause for R_New. This is exactly what we do in the specification of the child package P.Rep:

Listing 32: p.ads

```

1 package P is
2
3   type R is record
4     A : Integer;
5     B : Integer;
6   end record;
7
8 end P;
```

Listing 33: p-rep.ads

```

1 package P.Rep is
2
3   type R_New is new R
4     with Object_Size => 72;
5
6   for R_New use record
7     A at 0 range 0 .. 31;
8     B at 5 range 0 .. 31;
9   end record;
10
11 end P.Rep;
```

Listing 34: show_empty_byte.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;           use P;
4 with P.Rep;      use P.Rep;
5
6 procedure Show_Empty_Byte is
7 begin
8   Put_Line ("R'Size:          "
9             & R'Size'Image);
10  Put_Line ("R'Object_Size: "
11           & R'Object_Size'Image);
12
13  Put_Line ("R_New'Size:      "
14           & R_New'Size'Image);
15  Put_Line ("R_New'Object_Size: "
16           & R_New'Object_Size'Image);
17 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.

2.3.4 Representation on Bit Level

A very common application of representation clauses is to specify individual bits of a record. This is particularly useful, for example, when mapping registers or implementing protocols.

Let's consider the following fictitious register as an example:

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

```
1 package P is
2
3   type Status is (Ready, Waiting,
4                   Processing, Done);
5   type UInt_3 is range 0 .. 2 ** 3 - 1;
6
7   type Simple_Reg is record
8     S      : Status;
9     Error  : Boolean;
10    V1     : UInt_3;
11  end record;
12
13  for Simple_Reg use record
14    S      at 0 range 0 .. 1;
15    -- Bit #2 and 3: reserved!
16    Error  at 0 range 4 .. 4;
17    V1     at 0 range 5 .. 7;
18  end record;
19
20 end P;
```

Listing 36: show_simple_reg.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;          use P;
4
5 procedure Show_Simple_Reg is
6 begin
7   Put_Line ("Simple_Reg'Size:      "
8             & Simple_Reg'Size'Image);
9   Put_Line ("Simple_Reg'Object_Size: "
10            & Simple_Reg'Object_Size'Image);
11 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

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;          use P;
4
5 procedure Show_Simple_Reg is
6   Default : constant Simple_Reg :=
7     (S      => Ready,
8      Error => False,
9      V1     => 0);
10
11   R : Simple_Reg := Default;
12 begin
13   Put_Line ("R.S: " & R.S'Image);
14
15   R.V1 := 4;
16
17   Put_Line ("R.V1: " & R.V1'Image);
18 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 `VI` field of the register with the value 4, we just have to write `R.VI := 4`. No extra code — such as bit-masking or bit-shifting — is needed here.

In other languages

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

2.4 Changing Data Representation

Note: This section was originally written by Robert Dewar and published as [Gem #27: Changing Data Representation](#)³⁰ and [Gem #28](#)³¹.

A powerful feature of Ada is the ability to specify the exact data layout. This is particularly important when you have an external device or program that requires a very specific format. Some examples are:

Listing 38: communication.ads

```
1 package Communication is
2
3     type Com_Packet is record
4         Key : Boolean;
5         Id  : Character;
6         Val : Integer range 100 .. 227;
7     end record;
8
9     for Com_Packet use record
10        Key at 0 range 0 .. 0;
11        Id  at 0 range 1 .. 8;
12        Val at 0 range 9 .. 15;
13    end record;
14
15 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:

³⁰ <https://www.adacore.com/gems/gem-27>

³¹ <https://www.adacore.com/gems/gem-28>

Listing 39: array_representation.ads

```

1 package Array_Representation is
2
3     type Val is (A, B, C, D, E, F, G, H);
4
5     type Arr is array (1 .. 16) of Val
6         with Component_Size => 3;
7
8 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

```

1 package Enumeration_Representation is
2
3     type Status is (Off, On, Unknown);
4     for Status use (Off      => 2#001#,
5                    On       => 2#010#,
6                    Unknown => 2#100#);
7
8 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"](#)³².

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

Listing 41: array_representation.ads

```

1 package Array_Representation is
2
```

(continues on next page)

³² <http://www.ada-auth.org/standards/22rm/html/RM-13-6.html>

(continued from previous page)

```
3  type Val is (A, B, C, D, E, F, G, H);
4  type Arr is array (1 .. 16) of Val;
5
6  type External_Arr is new Arr
7    with Component_Size => 3;
8
9  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

```
1  with Array_Representation;
2  use Array_Representation;
3
4  procedure Using_Array_For_IO is
5    Input_Data  : External_Arr;
6    Work_Data   : Arr;
7    Output_Data : External_Arr;
8  begin
9    -- (read data into Input_Data)
10
11    -- Now convert to internal form
12    Work_Data := Arr (Input_Data);
13
14    -- (computations using efficient
15    -- Work_Data form)
16
17    -- Convert back to external form
18    Output_Data := External_Arr (Work_Data);
19
20  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

```

1 package Enumeration_Representation is
2
3     type Status_In is (Off, On, Unknown);
4     type Status_Out is new Status_In;
5
6     for Status_In use (Off      => 2#001#,
7                       On       => 2#010#,
8                       Unknown  => 2#100#);
9     for Status_Out use (Off      => 103,
10                      On       => 1045,
11                      Unknown  => 7700);
12
13 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 13.1³³](#) (10) that restricts the placement of interesting representation clauses:

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

All the representation clauses that are interesting from the point of view of change of representation are "type related", so for example, the following sequence would be illegal:

Listing 44: array_representation.ads

```

1 package Array_Representation is
2
3     type Val is (A, B, C, D, E, F, G, H);
4     type Arr is array (1 .. 16) of Val;
5
6     procedure Rearrange (Arg : in out Arr);
7
8     type External_Arr is new Arr
9         with Component_Size => 3;
10
11 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
```

³³ <http://www.ada-auth.org/standards/22rm/html/RM-13-1.html>

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

2.4.1 Restrictions

In the previous subsection, we discussed the use of derived types and representation clauses to achieve automatic change of representation. More accurately, this feature is not completely automatic, since it requires you to write an explicit conversion. In fact there is a principle behind the design here which says that a change of representation should never occur implicitly behind the back of the programmer without such an explicit request by means of a type conversion.

The reason for that is that the change of representation operation can be very expensive, since in general it can require component by component copying, changing the representation on each component.

Let's have a look at the -gnatG expanded code to see what is hidden under the covers here. For example, the conversion `Arr (Input_Data)` from the previous example generates the following expanded code:

```
B26b : declare
  [subtype p__TarrD1 is integer range 1 .. 16]
  R25b : p__TarrD1 := 1;
begin
  for L24b in 1 .. 16 loop
    [subtype p__arr__XP3 is
      system_unsigned_types__long_long_unsigned range 0 ..
      16#FFFF_FFFF_FFFF#]
    work_data := p__arr__XP3!((work_data and not shift_left!(
      16#7#, 3 * (integer(L24b - 1)))) or shift_left!(p__arr__XP3!
      (input_data (R25b)), 3 * (integer(L24b - 1))));
    R25b := p__TarrD1'succ(R25b);
  end loop;
end B26b;
```

That's pretty horrible! In fact, we could have simplified it for this section, but we have left it in its original form, so that you can see why it is nice to let the compiler generate all this stuff so you don't have to worry about it yourself.

Given that the conversion can be pretty inefficient, you don't want to convert backwards and forwards more than you have to, and the whole approach is only worthwhile if we'll be doing extensive computations involving the value.

The expense of the conversion explains two aspects of this feature that are not obvious. First, why do we require derived types instead of just allowing subtypes to have different representations, avoiding the need for an explicit conversion?

The answer is precisely that the conversions are expensive, and you don't want them happening behind your back. So if you write the explicit conversion, you get all the gobbledygook listed above, but you can be sure that this never happens unless you explicitly ask for it.

This also explains the restriction we mentioned in previous subsection from [RM 13.1³⁴](#) (10):

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

It turns out this restriction is all about avoiding implicit changes of representation. Let's have a look at how type derivation works when there are primitive subprograms defined at the point of derivation. Consider this example:

³⁴ <http://www.ada-auth.org/standards/22rm/html/RM-13-1.html>

Listing 45: my_ints.ads

```

1 package My_Ints is
2
3     type My_Int_1 is range 1 .. 10;
4
5     function Odd (Arg : My_Int_1)
6                 return Boolean;
7
8     type My_Int_2 is new My_Int_1;
9
10 end My_Ints;

```

Listing 46: my_ints.adb

```

1 package body My_Ints is
2
3     function Odd (Arg : My_Int_1)
4                 return Boolean is
5         (True);
6     -- Dummy implementation!
7
8 end My_Ints;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.My_Int
 MD5: a29401698307998288f02b349d04d1d2

Now when we do the type derivation, we inherit the function `Odd` for `My_Int_2`. But where does this function come from? We haven't written it explicitly, so the compiler somehow materializes this new implicit function. How does it do that?

We might think that a complete new function is created including a body in which `My_Int_2` replaces `My_Int_1`, but that would be impractical and expensive. The actual mechanism avoids the need to do this by use of implicit type conversions. Suppose after the above declarations, we write:

Listing 47: using_my_int.adb

```

1 with My_Ints; use My_Ints;
2
3 procedure Using_My_Int is
4     Var : My_Int_2;
5 begin
6
7     if Odd (Var) then
8         -- ^ Calling Odd function
9         --   for My_Int_2 type.
10        null;
11    end if;
12
13 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

```
1 with My_Ints; use My_Ints;
2
3 procedure Using_My_Int is
4   Var : My_Int_2;
5 begin
6
7   if Odd (My_Int_1 (Var)) then
8     --   ^ Converting My_Int_2 to
9     --   My_Int_1 type before
10    --   calling Odd function.
11    null;
12  end if;
13
14 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.

2.5 Valid Attribute

When receiving data from external sources, we're subjected to problems such as transmission errors. If not handled properly, erroneous data can lead to major issues in an application.

One of those issues originates from the fact that transmission errors might lead to invalid information stored in memory. When proper checks are active, using invalid information is detected at runtime and an exception is raised at this point, which might then be handled by the application.

Instead of relying on exception handling, however, we could instead ensure that the information we're about to use is valid. We can do this by using the `Valid` attribute. For example, if we have a variable `Var`, we can verify that the value stored in `Var` is valid by writing `Var'Valid`, which returns a `Boolean` value. Therefore, if the value of `Var` isn't valid, `Var'Valid` returns `False`, so we can have code that handles this situation before we actually make use of `Var`. In other words, instead of handling a potential exception in other parts of the application, we can proactively verify that input information is correct and avoid that an exception is raised.

In the next example, we show an application that

- generates a file containing mock-up data, and then
- reads information from this file as state values.

The mock-up data includes valid and invalid states.

Listing 49: create_test_file.ads

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

Listing 50: create_test_file.adb

```
1 with Ada.Sequential_IO;
2
3 procedure Create_Test_File (File_Name : String)
4 is
5     package Integer_Sequential_IO is new
6         Ada.Sequential_IO (Integer);
7     use Integer_Sequential_IO;
8
9     F : File_Type;
10 begin
11     Create (F, Out_File, File_Name);
12     Write (F, 1);
13     Write (F, 2);
14     Write (F, 4);
15     Write (F, 3);
16     Write (F, 2);
17     Write (F, 10);
18     Close (F);
19 end Create_Test_File;
```

Listing 51: states.ads

```
1 with Ada.Sequential_IO;
2
3 package States is
4
5     type State is (Off, On, Waiting)
6         with Size => Integer'Size;
```

(continues on next page)

(continued from previous page)

```
7
8   for State use (Off      => 1,
9                 On       => 2,
10                Waiting => 4);
11
12   package State_Sequential_IO is new
13     Ada.Sequential_IO (State);
14
15   procedure Read_Display_States
16     (File_Name : String);
17
18 end States;
```

Listing 52: states.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body States is
4
5   procedure Read_Display_States
6     (File_Name : String)
7   is
8     use State_Sequential_IO;
9
10    F : State_Sequential_IO.File_Type;
11    S : State;
12
13    procedure Display_State (S : State) is
14    begin
15      -- Before displaying the value,
16      -- check whether it's valid or not.
17      if S'Valid then
18        Put_Line (S'Image);
19      else
20        Put_Line ("Invalid value detected!");
21      end if;
22    end Display_State;
23
24  begin
25    Open (F, In_File, File_Name);
26
27    while not End_Of_File (F) loop
28      Read (F, S);
29      Display_State (S);
30    end loop;
31
32    Close (F);
33  end Read_Display_States;
34
35 end States;
```

Listing 53: show_states_from_file.adb

```
1 with States; use States;
2 with Create_Test_File;
3
4 procedure Show_States_From_File is
5   File_Name : constant String := "data.bin";
6 begin
7   Create_Test_File (File_Name);
```

(continues on next page)

(continued from previous page)

```
8   Read_Display_States (File_Name);  
9   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](#)³⁵

³⁵ <http://www.ada-auth.org/standards/22rm/html/RM-13-9-2.html>

2.6 Unchecked Union

We've introduced variant records back in the [Introduction to Ada course](#)³⁶. In simple terms, a variant record is a record with discriminants that allows for changing its structure. Basically, it's a record containing a **case**.

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

Listing 54: `states.ads`

```
1 package States is
2
3   type State is (Off, On, Waiting)
4     with Size => Integer'Size;
5
6   for State use (Off      => 1,
7                 On       => 2,
8                 Waiting => 4);
9
10  type State_Or_Integer (Use_Enum : Boolean) is
11    record
12      case Use_Enum is
13        when False => I : Integer;
14        when True  => S : State;
15      end case;
16    end record;
17
18  procedure Display_State_Value
19    (V : State_Or_Integer);
20
21 end States;
```

Listing 55: `states.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body States is
4
5   procedure Display_State_Value
6     (V : State_Or_Integer)
7   is
8     begin
9       Put_Line ("State: " & V.S'Image);
10      Put_Line ("Value: " & V.I'Image);
11    end Display_State_Value;
12
13 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` ex-

³⁶ https://learn.adacore.com/courses/intro-to-ada/chapters/more_about_records.html#intro-ada-variant-records

ception is raised as expected because `Use_Enum` is set to `True`, so that the I component is invalid — only the S component is valid in this case.

Listing 56: `show_variant_rec_error.adb`

```
1 with States; use States;
2
3 procedure Show_Variant_Rec_Error is
4   V : State_Or_Integer (Use_Enum => True);
5 begin
6   V.S := 0n;
7   Display_State_Value (V);
8 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: 0N
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`

```
1 with States; use States;
2
3 procedure Show_Variant_Rec_Error is
4   V : State_Or_Integer (Use_Enum => True);
5 begin
6   V.I := 4;
7   -- Error: V.I cannot be accessed because
8   --       Use_Enum is set to True.
9 end Show_Variant_Rec_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_Or_Integer
MD5: 985a84facc3d590ac767e914bea0c1d

Build output

```
show_variant_rec_error.adb:4:04: warning: variable "V" is never read and never assigned [-gnatw]
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 runtime [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

```
1 package States is
2
3   type State is (Off, On, Waiting)
4     with Size => Integer'Size;
5
6   for State use (Off      => 1,
7                 On       => 2,
8                 Waiting => 4);
9
10  type State_Or_Integer (Use_Enum : Boolean) is
11    record
12      case Use_Enum is
13        when False => I : Integer;
14        when True  => S : State;
15      end case;
16    end record;
17
18  type Unchecked_State_Or_Integer
19    (Use_Enum : Boolean) is new
20    State_Or_Integer (Use_Enum)
21    with Unchecked_Union;
22
23  procedure Display_State_Value
24    (V : Unchecked_State_Or_Integer);
25
26 end States;
```

Listing 59: states.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body States is
4
5   procedure Display_State_Value
6     (V : Unchecked_State_Or_Integer)
7   is
8     begin
9       Put_Line ("State: " & V.S'Image);
10      Put_Line ("Value: " & V.I'Image);
11    end Display_State_Value;
12
13 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

```

1 with States; use States;
2
3 procedure Show_Unchecked_Union is
4   V : State_Or_Integer (Use_Enum => True);
5 begin
6   V.S := 0n;
7   Display_State_Value
8     (Unchecked_State_Or_Integer (V));
9 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: 0N
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

```

1 with States; use States;
2
3 procedure Show_Unchecked_Union is
4   V : Unchecked_State_Or_Integer
5     (Use_Enum => True);
6 begin
7   V := (Use_Enum => True, S => 0n);
8   Display_State_Value (V);
9
10  V := (Use_Enum => False, I => 4);
11  Display_State_Value (V);
12 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: 0N
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

```
1 with States; use States;
2
3 procedure Show_Unchecked_Union is
4   V : Unchecked_State_Or_Integer
5     (Use_Enum => True);
6 begin
7   V.S := 0n;
8   Display_State_Value (V);
9
10  V.I := 4;
11  -- Error: cannot directly assign to V.I,
12  --      as Use_Enum is set to True.
13
14  Display_State_Value (V);
15 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: 0N
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

```
1 with States; use States;
2
3 procedure Show_Unchecked_Union is
4   V : Unchecked_State_Or_Integer
5     (Use_Enum => True);
6 begin
7   V := (Use_Enum => False, I => 3);
8   Display_State_Value (V);
9 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`

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 package body States is  
4  
5   procedure Display_State_Value  
6     (V : Unchecked_State_Or_Integer)  
7   is  
8     begin  
9       if V.S'Valid then  
10        Put_Line ("State: " & V.S'Image);  
11      else  
12        Put_Line ("State: <invalid>");  
13      end if;  
14      Put_Line ("Value: " & V.I'Image);  
15    end Display_State_Value;  
16  
17 end States;
```

Listing 65: `show_unchecked_union.adb`

```
1 with States; use States;  
2  
3 procedure Show_Unchecked_Union is  
4   V : Unchecked_State_Or_Integer  
5     (Use_Enum => True);  
6 begin  
7   V := (Use_Enum => False, I => 3);  
8   Display_State_Value (V);  
9 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](#)³⁷

³⁷ <http://www.ada-auth.org/standards/22rm/html/RM-B-3-3.html>

2.7 Shared variable control

Ada has built-in support for handling both volatile and atomic data. Let's start by discussing volatile objects.

In the Ada Reference Manual

- [C.6 Shared Variable Control](#)³⁸
-

2.7.1 Volatile

A [volatile](#)³⁹ object can be described as an object in memory whose value may change between two consecutive memory accesses of a process A — even if process A itself hasn't changed the value. This situation may arise when an object in memory is being shared by multiple threads. For example, a thread *B* may modify the value of that object between two read accesses of a thread *A*. Another typical example is the one of [memory-mapped I/O](#)⁴⁰, where the hardware might be constantly changing the value of an object in memory.

Because the value of a volatile object may be constantly changing, a compiler cannot generate code to store the value of that object in a register and then use the value from the register in subsequent operations. Storing into a register is avoided because, if the value is stored there, it would be outdated if another process had changed the volatile object in the meantime. Instead, the compiler generates code in such a way that the process must read the value of the volatile object from memory for each access.

Let's look at a simple example:

Listing 66: show_volatile_object.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Volatile_Object is
4   Val : Long_Float with Volatile;
5 begin
6   Val := 0.0;
7   for I in 0 .. 999 loop
8     Val := Val + 2.0 * Long_Float (I);
9   end loop;
10
11   Put_Line ("Val: " & Long_Float'Image (Val));
12 end Show_Volatile_Object;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Volatile_Object_Ada
MD5: aa1e276e64e69813bfc3e3ef39f3dd47
```

Runtime output

```
Val: 9.990000000000000E+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:

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

³⁹ [https://en.wikipedia.org/wiki/Volatile_\(computer_programming\)](https://en.wikipedia.org/wiki/Volatile_(computer_programming))

⁴⁰ https://en.wikipedia.org/wiki/Memory-mapped_I/O

Listing 67: shared_var_types.ads

```

1 package Shared_Var_Types is
2
3     type Volatile_Long_Float is new
4       Long_Float with Volatile;
5
6 end Shared_Var_Types;
```

Listing 68: show_volatile_type.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Shared_Var_Types; use Shared_Var_Types;
3
4 procedure Show_Volatile_Type is
5     Val : Volatile_Long_Float;
6 begin
7     Val := 0.0;
8     for I in 0 .. 999 loop
9         Val := Val + 2.0 * Volatile_Long_Float (I);
10    end loop;
11
12    Put_Line ("Val: "
13              & Volatile_Long_Float'Image (Val));
14 end Show_Volatile_Type;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Volatile_Type
 MD5: 0d31156d47b2edcfb94debd016c8bb87

Runtime output

```
Val: 9.990000000000000E+05
```

Here, we're declaring a new type `Volatile_Long_Float` in the `Shared_Var_Types` package. This type is based on the `Long_Float` type and uses the `Volatile` aspect. Any object of this type is automatically volatile.

In addition to that, we can declare components of an array to be volatile. In this case, we can use the `Volatile_Components` aspect in the array declaration. For example:

Listing 69: show_volatile_array_components.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Volatile_Array_Components is
4     Arr : array (1 .. 2) of Long_Float
5         with Volatile_Components;
6 begin
7     Arr := (others => 0.0);
8
9     for I in 0 .. 999 loop
10        Arr (1) := Arr (1) + 2.0 * Long_Float (I);
11        Arr (2) := Arr (2) + 10.0 * Long_Float (I);
12    end loop;
13
14    Put_Line ("Arr (1): "
15              & Long_Float'Image (Arr (1)));
16    Put_Line ("Arr (2): "
```

(continues on next page)

(continued from previous page)

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

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_  
↳Control.Volatile_Array_Components  
MD5: 05b3ee20f08c5a85f5872727a61c148d
```

Runtime output

```
Arr (1): 9.990000000000000E+05  
Arr (2): 4.995000000000000E+06
```

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

Listing 70: `shared_var_types.ads`

```
1 package Shared_Var_Types is  
2  
3 private  
4     Arr : array (1 .. 2) of Long_Float  
5         with Volatile;  
6  
7 end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_  
↳Control.Volatile_Array  
MD5: c9b7b9f94f1fac295753c7e7b9426fb2
```

Note that, if the `Volatile` aspect is specified for an object, then the `Volatile_Components` aspect is also specified automatically — if it makes sense in the context, of course. In the example above, even though `Volatile_Components` isn't specified in the declaration of the `Arr` array, it's automatically set as well.

2.7.2 Independent

When you write code to access a single object in memory, you might actually be accessing multiple objects at once. For example, when you declare types that make use of representation clauses — as we've seen in previous sections —, you might be accessing multiple objects that are grouped together in a single storage unit. For example, if you have components `A` and `B` stored in the same storage unit, you cannot update `A` without actually writing (the same value) to `B`. Those objects aren't independently addressable because, in order to access one of them, we have to actually address multiple objects at once.

When an object is independently addressable, we call it an independent object. In this case, we make sure that, when accessing that object, we won't be simultaneously accessing another object. As a consequence, this feature limits the way objects can be represented in memory, as we'll see next.

To indicate that an object is independent, we use the `Independent` aspect:

Listing 71: `shared_var_types.ads`

```
1 package Shared_Var_Types is  
2  
3     I : Integer with Independent;
```

(continues on next page)

(continued from previous page)

```
4
5 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Independent_Object
MD5: d90fef37584ca8802b8a3e3858c0095b

Similarly, we can use this aspect when declaring types:

Listing 72: shared_var_types.ads

```
1 package Shared_Var_Types is
2
3     type Independent_Boolean is new Boolean
4       with Independent;
5
6     type Flags is record
7       F1 : Independent_Boolean;
8       F2 : Independent_Boolean;
9     end record;
10
11 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.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 73: shared_var_types-representation.ads

```
1 package Shared_Var_Types.Representation is
2
3     type Rep_Flags is new Flags;
4
5     for Rep_Flags use record
6       F1 at 0 range 0 .. 0;
7       F2 at 0 range 1 .. 1;
8       --      ^ ERROR: start position of
9       --      F2 is wrong!
10      --      ^ ERROR: F1 and F2 share the
11      --      same storage unit!
12     end record;
13
14 end Shared_Var_Types.Representation;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.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 74: `shared_var_types-representation.ads`

```
1 with System;
2
3 package Shared_Var_Types.Representation is
4
5     type Rep_Flags is new Flags;
6
7     for Rep_Flags use record
8         F1 at 0 range 0 .. System.Storage_Unit - 1;
9         F2 at 1 range 0 .. System.Storage_Unit - 1;
10    end record;
11
12 end Shared_Var_Types.Representation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.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 75: `shared_var_types-representation.ads`

```
1 with System;
2
3 package Shared_Var_Types.Representation is
4
5     type Rep_Flags is new Flags;
6
7     for Rep_Flags use record
8         F1 at 0 range 0 .. System.Storage_Unit - 1;
9         F2 at 2 range 0 .. System.Storage_Unit - 1;
```

(continues on next page)

(continued from previous page)

```

10   end record;
11
12 end Shared_Var_Types.Representation;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Independent_Type
MD5: 71fedf8aac7c19bca1ba3b487efa9b17
```

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

For arrays, we can use the `Independent_Components` aspect:

Listing 76: `shared_var_types.ads`

```

1 package Shared_Var_Types is
2
3   Flags : array (1 .. 8) of Boolean
4         with Independent_Components;
5
6 end Shared_Var_Types;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Independent_Components
MD5: b331d0a13adf45624b664839fe4ba42c
```

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

Listing 77: `shared_var_types.ads`

```

1 package Shared_Var_Types is
2
3   type Flags is
4     array (Positive range <>) of Boolean
5     with Independent_Components, Pack;
6
7   F : Flags (1 .. 8) with Size => 8;
8
9 end Shared_Var_Types;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Packed_Independent_Components
MD5: dbaff4f2559ef8a449dad251f42cddc0
```

Build output

```

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

As expected, this code doesn't compile. Here, we can have either independent components, or packed components. We cannot have both at the same time because packed

components aren't independently addressable. The compiler warns us that the Pack aspect won't have any effect on independent components. When we use the Size aspect in the declaration of F, we confirm this limitation. If we remove the Size aspect, however, the code is compiled successfully because the compiler ignores the Pack aspect and allocates a larger size for F:

Listing 78: shared_var_types.ads

```
1 package Shared_Var_Types is
2
3     type Flags is
4         array (Positive range <>) of Boolean
5         with Independent_Components, Pack;
6
7 end Shared_Var_Types;
```

Listing 79: show_flags_size.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 with Shared_Var_Types; use Shared_Var_Types;
5
6 procedure Show_Flags_Size is
7     F : Flags (1 .. 8);
8 begin
9     Put_Line ("Flags'Size:      "
10              & F'Size'Image & " bits");
11     Put_Line ("Flags (1)'Size:  "
12              & F (1)'Size'Image & " bits");
13     Put_Line ("# storage units:  "
14              & Integer'Image
15              (F'Size /
16              System.Storage_Unit));
17 end Show_Flags_Size;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Packed_Independent_Components
MD5: b96f921b08b1d8207749517f833fc121
```

Build output

```
show_flags_size.adb:7:04: warning: variable "F" is read but never assigned [-
↳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.

2.7.3 Atomic

An atomic object is an object that only accepts atomic reads and updates. The Ada standard specifies that "for an atomic object (including an atomic component), all reads and updates of the object as a whole are indivisible." In this case, the compiler must generate Assembly code in such a way that reads and updates of an atomic object must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes.

In other contexts

Generally, we can say that operations are said to be atomic when they can be completed without interruptions. This is an important requirement when we're performing operations on objects in memory that are shared between multiple processes.

This definition of atomicity above is used, for example, when implementing databases. However, for this section, we're using the term "atomic" differently. Here, it really means that reads and updates must be performed with a single Assembly instruction.

For example, if we have a 32-bit object composed of four 8-bit bytes, the compiler cannot generate code to read or update the object using four 8-bit store / load instructions, or even two 16-bit store / load instructions. In this case, in order to maintain atomicity, the compiler must generate code using one 32-bit store / load instruction.

Because of this strict definition, we might have objects for which the `Atomic` aspect cannot be specified. Lots of machines support integer types that are larger than the native word-sized integer. For example, a 16-bit machine probably supports both 16-bit and 32-bit integers, but only 16-bit integer objects can be marked as atomic — or, more generally, only objects that fit into at most 16 bits.

Atomicity may be important, for example, when dealing with shared hardware registers. In fact, for certain architectures, the hardware may require that memory-mapped registers are handled atomically. In Ada, we can use the `Atomic` aspect to indicate that an object is atomic. This is how we can use the aspect to declare a shared hardware register:

Listing 80: `shared_var_types.ads`

```

1  with System;
2
3  package Shared_Var_Types is
4
5  private
6      R : Integer
7          with Atomic,
8              Address =>
9                  System'To_Address (16#FFFF00A0#);
10
11 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Atomic_Object
MD5: 5c2d8e0a9615084c2a15f896c61adaa6

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

Later on, we talk again about the [Address aspect](#) (page 133) and the GNAT-specific [System'To_Address attribute](#) (page 134).

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

Listing 81: shared_var_types.ads

```
1 with System;
2
3 package Shared_Var_Types is
4
5     type Atomic_Integer is new Integer
6       with Atomic;
7
8 private
9     R : Atomic_Integer
10        with Address =>
11           System'To_Address (16#FFFF00A0#);
12
13 end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Atomic_Types
MD5: 009632ba0155d70def8281ba590f3d12
```

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

We can also declare atomic array components:

Listing 82: shared_var_types.ads

```
1 package Shared_Var_Types is
2
3 private
4     Arr : array (1 .. 2) of Integer
5         with Atomic_Components;
6
7 end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Atomic_Array_Components
MD5: 7501bdf618621a822d451da8d731ef75
```

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

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

Listing 83: shared_var_types.ads

```
1 package Shared_Var_Types is
2
3     type Atomic_Integer_1 is new Integer
4       with Atomic;
5
6     type Atomic_Integer_2 is new Integer
7       with Atomic,
8         Volatile,
9         Independent;
```

(continues on next page)

(continued from previous page)

```

10
11 end Shared_Var_Types;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
↳Control.Atomic_Volatile_Independent
MD5: 3034c7a07698491f961d9b4fb74f03d8
```

A similar rule applies to components of an array. When we use the `Atomic_Components`, the following aspects are implied: `Volatile`, `Volatile_Components` and `Independent_Components`. For example, these array declarations are equivalent:

Listing 84: `shared_var_types.ads`

```

1 package Shared_Var_Types is
2
3   Arr_1 : array (1 .. 2) of Integer
4         with Atomic_Components;
5
6   Arr_2 : array (1 .. 2) of Integer
7         with Atomic_Components,
8              Volatile,
9              Volatile_Components,
10             Independent_Components;
11
12 end Shared_Var_Types;
```

2.8 Addresses

In other languages, such as C, the concept of pointers and addresses plays a prominent role. (In fact, in C, many optimizations rely on the usage of pointer arithmetic.) The concept of addresses does exist in Ada, but it's mainly reserved for very specific applications, mostly related to low-level programming. In general, other approaches — such as using access types — are more than sufficient. (We discuss *access types* (page 481) in another chapter. Also, later on in that chapter, we discuss the *relation between access types and addresses* (page 595).) In this section, we discuss some details about using addresses in Ada.

We make use of the `Address` type, which is defined in the `System` package, to handle addresses. In contrast to other programming languages (such as C or C++), an address in Ada isn't an integer value: its definition depends on the compiler implementation, and it's actually driven directly by the hardware. For now, let's consider it to usually be a private type — this can be seen as an attempt to achieve application code portability, given the variations in hardware that result in different definitions of what an address actually is.

The `Address` type has support for *address comparison* (page 135) and *address arithmetic* (page 137) (also known as *pointer arithmetic* in C). We discuss these topics later in this section. First, let's talk about the `Address` attribute and the `Address` aspect.

In the Ada Reference Manual

- [13.7 The Package System](#)⁴¹

⁴¹ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

2.8.1 Address attribute

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

Listing 85: use_address.adb

```
1 with System; use System;
2
3 procedure Use_Address is
4   I : aliased Integer := 5;
5   A : Address;
6 begin
7   A := I'Address;
8 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.

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 a user message, for example:

Listing 86: show_address_attribute.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System.Address_Image;
3
4 procedure Show_Address_Attribute is
5   I : aliased Integer := 5;
6 begin
7   Put_Line ("Address : "
8     & System.Address_Image (I'Address));
9 end Show_Address_Attribute;
```

Code block metadata

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

Runtime output

```
Address : 00007FFC1194DB04
```

In the Ada Reference Manual

- [13.3 Operational and Representation Attributes](#)⁴²

⁴² <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

- 13.7 The Package System⁴³

2.8.2 Address aspect

Usually, we let the compiler select the address of an object in memory, or let it use a register to store that object. However, we can specify the address of an object with the **Address** aspect. In this case, the compiler won't select an address automatically, but use the address that we're specifying. For example:

Listing 87: show_address.adb

```

1 with System; use System;
2 with System.Address_Image;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Address is
7
8     I_Main   : aliased Integer;
9     I_Mapped : Integer
10              with Address => I_Main'Address;
11 begin
12     Put_Line ("I_Main'Address   : "
13             & System.Address_Image
14             (I_Main'Address));
15     Put_Line ("I_Mapped'Address : "
16             & System.Address_Image
17             (I_Mapped'Address));
18 end Show_Address;
```

Code block metadata

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

Runtime output

```

I_Main'Address   : 00007FFD95171854
I_Mapped'Address : 00007FFD95171854
```

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

Listing 88: simple_overlay.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Overlay is
4     type State is (Off, State_1, State_2)
5                 with Size => Integer'Size;
6
7     for State use (Off      => 0,
8                  State_1 => 32,
9                  State_2 => 64);
10
11     S : State;
12     I : Integer
13       with Address => S'Address, Import, Volatile;
```

(continues on next page)

⁴³ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

(continued from previous page)

```
14 begin
15     S := State_2;
16     Put_Line ("I = " & Integer'Image (I));
17 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, as we did while talking about the Atomic (page 129) aspect:

Listing 89: shared_var_types.ads

```
1 with System;
2
3 package Shared_Var_Types is
4
5 private
6     R : Integer
7         with Atomic,
8         Address =>
9             System'To_Address (16#FFFF00A0#);
10
11 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 Manual⁴⁴, the System'To_Address attribute denotes a function identical to To_Address (from the System.Storage_Elements package) except that it is a static attribute. (We talk about the To_Address function (page 136) function later on.)

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes⁴⁵
- 13.7 The Package System⁴⁶

⁴⁴ https://gcc.gnu.org/onlinedocs/gnat_rm/Attribute-To_005fAddress.html

⁴⁵ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

⁴⁶ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

- 13.7.1 The Package System.Storage_Elements⁴⁷

2.8.3 Address comparison

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

Listing 90: show_address.adb

```

1 with System; use System;
2 with System.Address_Image;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Address is
7
8   I, J : Integer;
9 begin
10  Put_Line ("I'Address  : "
11           & System.Address_Image
12           (I'Address));
13  Put_Line ("J'Address  : "
14           & System.Address_Image
15           (J'Address));
16
17  if I'Address = J'Address then
18    Put_Line ("I'Address = J'Address");
19  elsif I'Address < J'Address then
20    Put_Line ("I'Address < J'Address");
21  else
22    Put_Line ("I'Address > J'Address");
23  end if;
24 end Show_Address;

```

Code block metadata

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

Runtime output

```

I'Address   : 00007FFC9FF008AC
J'Address   : 00007FFC9FF008A8
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 System⁴⁸

⁴⁷ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html>

⁴⁸ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

2.8.4 Address to integer conversion

The `System.Storage_Elements` package offers an integer representation of an address via the `Integer_Address` type, which is an integer type unrelated to common integer types such as `Integer` and `Long_Integer`. (The actual definition of `Integer_Address` is compiler-dependent, and it can be a signed or modular integer subtype.)

We can convert between the `Address` and `Integer_Address` types by using the `To_Address` and `To_Integer` functions. Let's see an example:

Listing 91: `show_address.adb`

```
1 with System;      use System;
2
3 with System.Storage_Elements;
4 use System.Storage_Elements;
5
6 with System.Address_Image;
7
8 with Ada.Text_IO; use Ada.Text_IO;
9
10 procedure Show_Address is
11     I      : Integer;
12     A1, A2 : Address;
13     IA     : Integer_Address;
14 begin
15     A1 := I'Address;
16     IA := To_Integer (A1);
17     A2 := To_Address (IA);
18
19     Put_Line ("A1 : "
20             & System.Address_Image (A1));
21     Put_Line ("IA : "
22             & Integer_Address'Image (IA));
23     Put_Line ("A2 : "
24             & System.Address_Image (A2));
25 end Show_Address;
```

Code block metadata

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

Runtime output

```
A1 : 00007FFE6C199BFC
IA : 140730712038396
A2 : 00007FFE6C199BFC
```

Here, we retrieve the address of the `I` object and store it in the `A1` address. Then, we convert `A1` to an integer address by calling `To_Integer` (and store it in `IA`). Finally, we convert this integer address back to an actual address by calling `To_Address`.

In the Ada Reference Manual

- [13.7.1 The Package `System.Storage_Elements`](#)⁴⁹

⁴⁹ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html>


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

Runtime output

```
Curr'Address : 00007FFC306CD8F0
Curr'Address : 00007FFC306CD8F4
Curr'Address : 00007FFC306CD8F8
Curr'Address : 00007FFC306CD8FC
Curr'Address : 00007FFC306CD900
Curr'Address : 00007FFC306CD904
Curr'Address : 00007FFC306CD908
Curr'Address : 00007FFC306CD90C
Curr'Address : 00007FFC306CD910
Curr'Address : 00007FFC306CD914
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
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 84).)

In other languages

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

Listing 93: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int i;
6     int arr[10];
7
8     int *a = arr;
9     /* int *a = &arr[0]; */
10
11     for (i = 0; i < 10; i++)
12     {
13         *a++ = i;
14         printf("curr address: %p\n", a);
15     }
16
17     for (i = 0; i < 10; i++)
```

(continues on next page)

(continued from previous page)

```
18     {
19         printf("arr[%d]: %d\n", i, arr[i]);
20     }
21
22     return 0;
23 }
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_
↪Arith_C
MD5: 7aa709a4d7ed6ce2346dbabc853e28c0
```

Runtime output

```
curr address: 0x7fff85be73f4
curr address: 0x7fff85be73f8
curr address: 0x7fff85be73fc
curr address: 0x7fff85be7400
curr address: 0x7fff85be7404
curr address: 0x7fff85be7408
curr address: 0x7fff85be740c
curr address: 0x7fff85be7410
curr address: 0x7fff85be7414
curr address: 0x7fff85be7418
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](#)⁵⁰
 - [13.7.1 The Package System.Storage_Elements](#)⁵¹
-

⁵⁰ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

⁵¹ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html>

2.9 Discarding names

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

Listing 94: show_enumeration_image.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Enumeration_Image is
4
5     type Months is
6         (January, February, March, April,
7          May, June, July, August, September,
8          October, November, December);
9
10    M : constant Months := January;
11 begin
12     Put_Line ("Month: "
13              & Months'Image (M));
14 end Show_Enumeration_Image;
```

Code block metadata

```

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 95: show_enumeration_image.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Enumeration_Image is
4
5     type Months is
6         (January, February, March, April,
7          May, June, July, August, September,
8          October, November, December);
9
10    M : constant Months := January;
11
12    function Months_Image (M : Months)
13                          return String is
14 begin
15     case M is
16     when January => return "JANUARY";
17     when February => return "FEBRUARY";
18     when March => return "MARCH";
19     when April => return "APRIL";
20     when May => return "MAY";
21     when June => return "JUNE";
22     when July => return "JULY";
23     when August => return "AUGUST";
24     when September => return "SEPTEMBER";
```

(continues on next page)

(continued from previous page)

```

25     when October   => return "OCTOBER";
26     when November => return "NOVEMBER";
27     when December => return "DECEMBER";
28     end case;
29     end Months_Image;
30
31 begin
32     Put_Line ("Month: "
33              & Months_Image (M));
34 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 96: `show_discard_names.adb`

```

1  procedure Show_Discard_Names is
2     pragma Warnings (Off, "is not referenced");
3
4     type Months is
5         (January, February, March, April,
6          May, June, July, August, September,
7          October, November, December)
8         with Discard_Names;
9
10    M : constant Months := January;
11 begin
12    null;
13 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 during compilation.

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

In the GNAT toolchain

If we add this statement to the `Show_Discard_Names` procedure above:

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

we see that the application displays "0" instead of "JANUARY". This is because GNAT doesn't store the strings associated with the `Months` type when we use the `Discard_Names` aspect for the `Months` type. (Therefore, the `Months'Image` attribute doesn't have that information.) Instead, the compiler uses the integer value of the enumeration, so that `Months'Image` returns the corresponding string for this integer value.

In the Ada Reference Manual

- [Aspect Discard_Names](#)⁵²
-

⁵² <http://www.ada-auth.org/standards/22rm/html/RM-C-5.html>

3.1 Default Initialization

As mentioned in the [Introduction to Ada](#)⁵³ course, record components can have default initial values. Also, we've seen that other kinds of types can have *default values* (page 63).

In the Ada Reference Manual, we refer to these default initial values as "default expressions of record components." The term *default expression* indicates that we can use any kind of expression for the default initialization of record components — which includes subprogram calls for example:

Listing 1: show_default_initialization.ads

```
1 package Show_Default_Initialization is
2
3     function Init return Integer is
4         (42);
5
6     type Rec is record
7         A : Integer := Init;
8     end record;
9
10 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](#)⁵⁴

⁵³ <https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-default-values>

⁵⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-8.html>

3.1.1 Dependencies

Default expressions cannot depend on other components. For example, if we have two components A and B, we cannot initialize B based on the value that A has:

Listing 2: show_default_initialization_dependency.ads

```
1 package Show_Default_Initialization_Dependency is
2
3     function Init return Integer is
4         (42);
5
6     type Rec is record
7         A : Integer := Init;
8         B : Integer := Rec.A; -- Illegal!
9     end record;
10
11 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.)

3.1.2 Initialization Order

The default initialization of record components is performed in arbitrary order. In fact, the order is decided by the compiler, so we don't have control over it.

Let's see an example:

Listing 3: simple_recs.ads

```
1 package Simple_Recs is
2
3     function Init (S : String;
4                   I : Integer)
5         return Integer;
6
7     type Rec is record
8         A : Integer := Init ("A", 1);
9         B : Integer := Init ("B", 2);
10    end record;
11
12 end Simple_Recs;
```

Listing 4: simple_recs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Simple_Recs is
4
5     function Init (S : String;
6                   I : Integer)
7                   return Integer is
8
9     begin
10        Put_Line (S & ": " & I'Image);
11        return I;
12    end Init;
13 end Simple_Recs;
```

Listing 5: show_initialization_order.adb

```

1 with Simple_Recs; use Simple_Recs;
2
3 procedure Show_Initialization_Order is
4     R : Rec;
5 begin
6     null;
7 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.

3.1.3 Evaluation

According to the Annotated Ada Reference Manual, the "default expression of a record component is only evaluated upon the creation of a default-initialized object of the record type." This means that the default expression is by itself not evaluated when we declare the record type, but when we create an object of this type. It follows from this rule that the default is only evaluated when necessary, i.e., when an explicit initial value is not specified in the object declaration.

Let's see an example:

Listing 6: show_initialization_order.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Simple_Recs; use Simple_Recs;
3
4 procedure Show_Initialization_Order is
5 begin
6   Put_Line ("Some processing first...");
7   Put_Line
8     ("Now, let's declare an object "
9     & "of the record type Rec...");
10
11  declare
12    R : Rec;
13  begin
14    Put_Line
15      ("An object of Rec type has "
16      & "just been created.");
17  end;
18
19 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](#)⁵⁵

⁵⁵ <http://www.ada-auth.org/standards/22aarm/html/AA-3-8.html>

3.1.4 Defaults and object declaration

Note: This subsection was originally written by Robert A. Duff and published as [Gem #12: Limited Types in Ada 2005](#)⁵⁶.

Consider the following type declaration:

Listing 7: type_defaults.ads

```

1 package Type_Defaults is
2   type Color_Enum is (Red, Blue, Green);
3
4   type T is private;
5 private
6   type T is
7     record
8       Color      : Color_Enum := Red;
9       Is_Gnarly  : Boolean := False;
10      Count       : Natural;
11    end record;
12
13   procedure Do_Something;
14 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

```

1 package body Type_Defaults is
2
3   Object_100 : constant T :=
4     (Color      => <>,
5      Is_Gnarly => <>,
6      Count     => 100);
7
8   procedure Do_Something is null;
9
10 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:

⁵⁶ <https://www.adacore.com/gems/ada-gem-12>

Listing 9: type_defaults.adb

```
1 package body Type_Defaults is
2
3   Object_100 : constant T :=
4       (Color    => Red,
5        Is_Gnarly => False,
6        Count    => 100);
7
8   procedure Do_Something is null;
9
10 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

```
1 package body Type_Defaults is
2
3   Object_100 : constant T := (Count => 100,
4                               others => <>);
5
6   procedure Do_Something is null;
7
8 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

```
1 package body Type_Defaults is
2
3   procedure Do_Something is
4       Object_100 : T;
5   begin
6       Object_100.Count := 100;
7   end Do_Something;
8
9 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**.

3.1.5 Advanced Usages

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

For example:

Listing 12: rec_per_object_expressions.ads

```

1 package Rec_Per_Object_Expressions is
2
3     type T (D : Positive) is private;
4
5 private
6
7     type T (D : Positive) is record
8         V : Natural := D - 1;
9         --      ^^^^^
10        --      Per-object expression
11    end record;
12
13 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.

3.2 Mutually dependent types

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

Listing 13: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T1 is record
4         B : T2;
5     end record;
6
7     type T2 is record
8         A : T1;
9     end record;
10
11 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

```
1 package Mutually_Dependent is
2
3     type T2;
4     type T2_Access is access T2;
5
6     type T1 is record
7         B : T2_Access;
8     end record;
9
10    type T2 is record
11        A : T1;
12    end record;
13
14 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

```
1 package Mutually_Dependent is
2
3     type T1;
4     type T1_Access is access T1;
5
6     type T2;
```

(continues on next page)

(continued from previous page)

```

7  type T2_Access is access T2;
8
9  type T1 is record
10     B : T2_Access;
11 end record;
12
13 type T2 is record
14     A : T1_Access;
15 end record;
16
17 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 624).

In the Ada Reference Manual

- [3.10.1 Incomplete Type Declarations](#)⁵⁷

3.3 Null records

A null record is a record that doesn't have any components. Consequently, it cannot store any information. When declaring a null record, we simply write `null` instead of declaring actual components, as we usually do for records. For example:

Listing 16: null_recs.ads

```

1  package Null_Recs is
2
3     type Null_Record is record
4         null;
5     end record;
6
7  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

```

1  package Null_Recs is
2
3     type Null_Record is null record;
```

(continues on next page)

⁵⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html>

(continued from previous page)

```
4
5 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

```
1 package Null_Recs is
2
3     type Null_Record is null record;
4
5     function "+" (A, B : Null_Record)
6                 return Null_Record;
7
8 end Null_Recs;
```

Listing 19: null_recs.adb

```
1 package body Null_Recs is
2
3     function "+" (A, B : Null_Record)
4                 return Null_Record
5
6     is
7         pragma Unreferenced (A, B);
8     begin
9         return (null record);
10    end "+";
11 end Null_Recs;
```

Listing 20: show_null_rec.adb

```
1 with Null_Recs; use Null_Recs;
2
3 procedure Show_Null_Rec is
4     A, B : Null_Record;
5 begin
6     B := A + A;
7     A := A + B;
8 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 Aggregates⁵⁸

⁵⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html>

3.3.1 Simple Prototyping

A null record doesn't provide much functionality on itself, as we're not storing any information in it. However, it's far from being useless. For example, we can make use of null records to design an API, which we can then use in an application without having to implement the actual functionality of the API. This allows us to design a prototype without having to think about all the implementation details of the API in the first stage.

Consider this example:

Listing 21: devices.ads

```

1 package Devices is
2
3   type Device is private;
4
5   function Create
6     (Active : Boolean)
7     return Device;
8
9   procedure Reset
10    (D : out Device) is null;
11
12  procedure Process
13    (D : in out Device) is null;
14
15  procedure Activate
16    (D : in out Device) is null;
17
18  procedure Deactivate
19    (D : in out Device) is null;
20
21 private
22
23   type Device is null record;
24
25   function Create (Active : Boolean)
26     return Device is
27     (null record);
28
29 end Devices;
```

Listing 22: show_device.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Devices;     use Devices;
3
4 procedure Show_Device is
5   A : Device;
6 begin
7   Put_Line ("Creating device...");
8   A := Create (Active => True);
9
10  Put_Line ("Processing on device...");
11  Process (A);
12
13  Put_Line ("Deactivating device...");
14  Deactivate (A);
15
16  Put_Line ("Activating device...");
17  Activate (A);
18
```

(continues on next page)

(continued from previous page)

```
19   Put_Line ("Resetting device...");
20   Reset (A);
21 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 392).

In the `Show_Device` procedure — which is an application that implements our prototype —, we declare an object of `Device` type and call all subprograms associated with that type.

3.3.2 Extending the prototype

Because we're either using expression functions or null procedures in the specification of the `Devices` package, we don't have a package body for it (as there's nothing to be implemented). We could, however, move those user messages from the `Show_Devices` procedure to a dummy implementation of the `Devices` package. This is the adapted code:

Listing 23: `devices.ads`

```
1 package Devices is
2
3   type Device is null record;
4
5   function Create (Active : Boolean)
6     return Device;
7
8   procedure Reset (D : out Device);
9
10  procedure Process (D : in out Device);
11
12  procedure Activate (D : in out Device);
13
14  procedure Deactivate (D : in out Device);
15
16 end Devices;
```

Listing 24: devices.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Devices is
4
5     function Create (Active : Boolean)
6         return Device
7     is
8         pragma Unreferenced (Active);
9     begin
10        Put_Line ("Creating device...");
11        return (null record);
12    end Create;
13
14    procedure Reset (D : out Device)
15    is
16        pragma Unreferenced (D);
17    begin
18        Put_Line ("Processing on device...");
19    end Reset;
20
21    procedure Process (D : in out Device)
22    is
23        pragma Unreferenced (D);
24    begin
25        Put_Line ("Deactivating device...");
26    end Process;
27
28    procedure Activate (D : in out Device)
29    is
30        pragma Unreferenced (D);
31    begin
32        Put_Line ("Activating device...");
33    end Activate;
34
35    procedure Deactivate (D : in out Device)
36    is
37        pragma Unreferenced (D);
38    begin
39        Put_Line ("Resetting device...");
40    end Deactivate;
41
42 end Devices;
```

Listing 25: show_device.adb

```
1 with Devices; use Devices;
2
3 procedure Show_Device is
4     A : Device;
5 begin
6     A := Create (Active => True);
7     Process (A);
8     Deactivate (A);
9     Activate (A);
10    Reset (A);
11 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.

3.3.3 More complex applications

As we've just seen, we can use null records like any other type and create complex prototypes with them. We could, for instance, design an application that makes use of many null records, or even have types that depend on or derive from null records. Let's see a simple example:

Listing 26: many_devices.ads

```
1 package Many_Devices is  
2  
3   type Device is null record;  
4  
5   type Device_Config is null record;  
6  
7   function Create (Config : Device_Config)  
8                   return Device is  
9       (null record);  
10  
11  type Derived_Device is new Device;  
12  
13  procedure Process (D : Derived_Device) is null;  
14  
15 end Many_Devices;
```

Listing 27: show_derived_device.adb

```
1 with Many_Devices; use Many_Devices;  
2  
3 procedure Show_Derived_Device is  
4     A : Device;  
5     B : Derived_Device;  
6     C : Device_Config;  
7 begin  
8     A := Create (Config => C);  
9     B := Create (Config => C);
```

(continues on next page)

(continued from previous page)

```

10
11   Process (B);
12 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.

3.3.4 Implementing the API

Let's focus again on the previous example. After we have an initial prototype, we can start implementing some of the functionality needed for the Device type. For example, we can store information about the current activation state in the record:

Listing 28: devices.ads

```

1 package Devices is
2
3   type Device is private;
4
5   function Create (Active : Boolean)
6     return Device;
7
8   procedure Reset (D : out Device);
9
10  procedure Process (D : in out Device);
11
12  procedure Activate (D : in out Device);
13
14  procedure Deactivate (D : in out Device);
15
16 private
17
18  type Device is record
19    Active : Boolean;
20  end record;
21
22 end Devices;
```

Listing 29: devices.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Devices is
4
5   function Create (Active : Boolean)
6     return Device
7   is
8     pragma Unreferenced (Active);
```

(continues on next page)

(continued from previous page)

```
9   begin
10      Put_Line ("Creating device...");
11      return (Active => Active);
12   end Create;
13
14   procedure Reset (D : out Device)
15   is
16      pragma Unreferenced (D);
17   begin
18      Put_Line ("Processing on device...");
19   end Reset;
20
21   procedure Process (D : in out Device)
22   is
23      pragma Unreferenced (D);
24   begin
25      Put_Line ("Deactivating device...");
26   end Process;
27
28   procedure Activate (D : in out Device)
29   is
30   begin
31      Put_Line ("Activating device...");
32      D.Active := True;
33   end Activate;
34
35   procedure Deactivate (D : in out Device)
36   is
37   begin
38      Put_Line ("Resetting device...");
39      D.Active := False;
40   end Deactivate;
41
42 end Devices;
```

Listing 30: show_device.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Devices;     use Devices;
3
4  procedure Show_Device is
5     A : Device;
6  begin
7     A := Create (Active => True);
8     Process (A);
9     Deactivate (A);
10    Activate (A);
11    Reset (A);
12 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: pragma Unreferenced given for "Active" [enabled by ↵  
↵default]
```

Runtime output

```
Creating device...
Deactivating device...
Resetting device...
Activating device...
Processing on device...
```

Now, the Device record contains an Active component, which is used in the updated versions of Create, Activate and Deactivate.

Note that we haven't done any change to the implementation of the Show_Device procedure: it's still the same application as before. As we've been hinting in the beginning, using null records makes it easy for us to first create a prototype — as we did in the Show_Device procedure — and postpone the API implementation to a later phase of the project.

3.3.5 Tagged null records

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

Listing 31: null_recs.ads

```
1 package Null_Recs is
2
3     type Tagged_Null_Record is
4       tagged null record;
5
6     type Abstract_Tagged_Null_Record is
7       abstract tagged null record;
8
9 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

```
1 package Devices is
2
3     type Device is private;
4
5     function Create (Active : Boolean)
6       return Device;
7
8     type Derived_Device is private;
9
10 private
11
12     type Device is tagged null record;
13
14     function Create (Active : Boolean)
15       return Device is
16       (null record);
17
```

(continues on next page)

(continued from previous page)

```
18  type Derived_Device is new Device with record
19      Active : Boolean;
20  end record;
21
22  function Create (Active : Boolean)
23      return Derived_Device is
24      (Active => Active);
25
26  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.

3.4 Per-Object Expressions

In record type declarations, we might want to define a component that makes use of a name that refers to a discriminant of the record type, or to the record type itself. An expression where we use such a name is called a per-object expression.

The term "per-object" comes from the fact that, in the component definition, we're referring to a piece of information that will be known just when creating an object of that type. For example, if the per-object expression refers to a discriminant of a type `T`, the actual value of that discriminant will only be specified when we declare an object of type `T`. Therefore, the component definition is specific for that individual object — but not necessarily for other objects of the same type, as we might use different values for the discriminant.

The constraint that contains a per-object expression is called a per-object constraint. The actual constraint of that component isn't completely known when we declare the record type, but only later on when an object of that type is created. (Note that the syntax of a constraint includes the parentheses or the keyword `range`.)

In addition to referring to discriminants, per-object expressions can also refer to the record type itself, as we'll see later.

Let's start with a simple record declaration:

Listing 33: `rec_per_object_expressions.ads`

```
1  package Rec_Per_Object_Expressions is
2
3      type Stack (S : Positive) is private;
4
5  private
6
7      type Integer_Array is
8          array (Positive range <>) of Integer;
9
10     type Stack (S : Positive) is record
11         Arr : Integer_Array (1 .. S);
12         --
```

(continues on next page)

(continued from previous page)

```

13      --
14      --                               S
15      --                               ^
16      --      Per-object expression
17      --
18      --                               (1 .. S)
19      --                               ^^^^^^^
20      --      Per-object constraint
21
22      Top : Natural := 0;
23  end record;
24
25 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 601):

Listing 34: rec_per_object_expressions.ads

```

1 package Rec_Per_Object_Expressions is
2
3     type T is private;
4
5     type T_Processor (Selected_T : access T) is
6         private;
7
8 private
9
10    type T is null record;
11
12    type T_Container (Selected_T : access T) is
13        null record;
14
15    type T_Processor (Selected_T : access T) is
16        record
17        E : T_Container (Selected_T);
18        --
19        --      Selected_T
20        --      ^^^^^^^
21        --      Per-object expression
22        --
23        --      (Selected_T)
24        --      ^^^^^^^
25        --      Per-object constraint
26    end record;
27
28 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 35: `rec_per_object_expressions.ads`

```
1 package Rec_Per_Object_Expressions is
2
3   type T is limited private;
4
5 private
6
7   type T_Processor (Selected_T : access T) is
8     null record;
9
10  type T is limited record
11    E : T_Processor (T'Access);
12    --
13    --           T'Access
14    --           ^^^^^^^^
15    -- Per-object expression
16    --
17    --           (T'Access)
18    --           ^^^^^^^^
19    -- Per-object constraint
20  end record;
21
22 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*.

Relevant topics

- [3.8 Record Types](#)⁵⁹

⁵⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-8.html>

3.4.1 Default value

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

Listing 36: rec_per_object_expressions.ads

```

1 package Rec_Per_Object_Expressions is
2
3   type T (D : Positive) is private;
4
5 private
6
7   type T (D : Positive) is record
8     V : Natural := D - 1;
9     --      ^^^^^
10    --      Per-object expression
11
12    S : Natural := D'Size;
13    --      ^^^^^
14    --      Per-object expression
15  end record;
16
17 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 164) that exist for per-object constraints, as we discuss later on.) Such expressions might include function calls or uses of any defined operator. For this reason, the following code example is accepted by the compiler:

Listing 37: rec_per_object_expressions.ads

```

1 package Rec_Per_Object_Expressions is
2
3   type Stack (S : Positive) is private;
4
5 private
6
7   type Integer_Array is
8     array (Positive range <>) of Integer;
9
10  type Stack (S : Positive) is record
11    Arr : Integer_Array (1 .. S);
12
13    Top : Natural := 0;
14
15    Overflow_Warning : Positive
16      := S * 9 / 10;
17    --      ^^^^^^^^^
18    --      Per-object expression
19    --      using computation for
20    --      the default expression.
21  end record
```

(continues on next page)

(continued from previous page)

```

22   with
23     Dynamic_Predicate =>
24     Overflow_Warning in
25       (S + 1) / 2 .. S - 1;
26   --
27   -- (S + 1) / 2
28   -- ~~~~~
29   -- Per-object expression
30   -- using computation.
31   --
32   --           S - 1
33   --           ~~~~~
34   -- Per-object expression
35   -- using computation.
36
37 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 $S - 1$.

3.4.2 Restrictions

There are some important restrictions on per-object constraints:

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

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

Listing 38: rec_per_object_expressions.ads

```

1  package Rec_Per_Object_Expressions is
2
3     type Bit_Field is
4       array (Positive range <>) of Boolean
5       with Pack;
6
7     type T is record
8       Arr : Bit_Field (1 .. T'Size);
9       --           ~~~~~
10      -- ERROR: per-object range constraint
11      --           using the Size attribute
12      --           is illegal.
13     end record;
14
15 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
```

1. Within a per-object index constraint or discriminant constraint, each per-object expression must be the name of a discriminant directly, without any further computation.
 - Therefore, we're allowed to write $(1 \dots S)$ — as we've seen in a previous example —. However, writing $(1 \dots S - 1)$ would be illegal.
 - For example, the following adaptation to the previous code example doesn't compile:

Listing 39: rec_per_object_expressions.ads

```
1 package Rec_Per_Object_Expressions is
2
3   type Stack (S : Positive) is private;
4
5 private
6
7   type Integer_Array is
8     array (Natural range <>) of Integer;
9
10  type Stack (S : Positive) is record
11    Arr : Integer_Array (0 .. S - 1);
12      --           ^^^^^
13      -- ERROR: computation in per-object
14      --           expression is illegal.
15
16    Top : Integer := -1;
17  end record;
18
19 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 163), this restriction doesn't apply when the computation is used in a per-object expression that calculates the default value of a component.)

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

AGGREGATES

4.1 Container Aggregates

Note: This feature was introduced in Ada 2022.

A container aggregate is a list of elements — such as `[1, 2, 3]` — that we use to initialize or assign to a container. For example:

Listing 1: `show_container_aggregate.adb`

```
1 pragma Ada_2022;
2
3 with Ada.Containers.Vectors;
4
5 procedure Show_Container_Aggregate is
6
7     package Float_Vec is new
8         Ada.Containers.Vectors (Positive, Float);
9
10    V : constant Float_Vec.Vector :=
11        [1.0, 2.0, 3.0];
12
13    pragma Unreferenced (V);
14 begin
15     null;
16 end Show_Container_Aggregate;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Simple_
↳ Container_Aggregate
MD5: ef13386fef0b7be0b3ea999a7752d5f1
```

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

```
1 pragma Ada_2022;
2
3 with Ada.Containers.Vectors;
4
5 procedure Show_Container_Aggregate is
6
7     package Float_Vec is new
8         Ada.Containers.Vectors (Positive, Float);
9
10    -- Null container aggregate
11    Null_V : constant Float_Vec.Vector :=
12            [];
13
14    -- Positional container aggregate
15    Pos_V  : constant Float_Vec.Vector :=
16            [1.0, 2.0, 3.0];
17
18    -- Named container aggregate
19    Named_V : constant Float_Vec.Vector :=
20            [1 => 1.0,
21             2 => 2.0,
22             3 => 3.0];
23
24    pragma Unreferenced (Null_V, Pos_V, Named_V);
25 begin
26     null;
27 end Show_Container_Aggregate;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Simple_
↳ Container_Aggregate
MD5: 15ed6370377423044368a5d56402e940
```

In this example, we see the three forms of container aggregates. The difference between positional and named container aggregates is that:

- for positional container aggregates, the vector index is implied by its position;

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

```
1 pragma Ada_2022;
2
3 with Ada.Containers.Vectors;
4 with Ada.Containers.Indefinite_Hashed_Maps;
5 with Ada.Strings.Hash;
6
7 procedure Show_Named_Container_Aggregate is
8
9     package Float_Vec is new
10        Ada.Containers.Vectors (Positive, Float);
11
```

(continues on next page)

(continued from previous page)

```

12 package Float_Hashed_Maps is new
13   Ada.Containers.Indefinite_Hashed_Maps
14     (Key_Type      => String,
15      Element_Type => Float,
16      Hash         => Ada.Strings.Hash,
17      Equivalent_Keys => "=");
18
19   -- Named container aggregate
20   -- using an index
21   Indexed_Named_V : constant Float_Vec.Vector :=
22     [1 => 1.0,
23      2 => 2.0,
24      3 => 3.0];
25
26   -- Named container aggregate
27   -- using a key
28   Keyed_Named_V : constant
29     Float_Hashed_Maps.Map :=
30     ["Key_1" => 1.0,
31      "Key_2" => 2.0,
32      "Key_3" => 3.0];
33
34   pragma Unreferenced (Indexed_Named_V,
35                       Keyed_Named_V);
36 begin
37   null;
38 end Show_Named_Container_Aggregate;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Named_Container_Aggregate
 MD5: 2eabf312c243856dcb2d6884f71e19e2

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](#)⁶⁰

⁶⁰ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-5.html>

4.2 Record aggregates

We've already seen record aggregates in the [Introduction to Ada⁶¹](#) course, so this is just a brief overview on the topic.

As we already know, record aggregates can have positional and named component associations. For example, consider this package:

Listing 4: points.ads

```
1 package Points is
2
3   type Point_3D is record
4     X, Y, Z : Integer;
5   end record;
6
7   procedure Display (P : Point_3D);
8
9 end Points;
```

Listing 5: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_3D) is
6   begin
7     Put_Line ("X => "
8       & Integer'Image (P.X)
9       & ",");
10    Put_Line (" Y => "
11      & Integer'Image (P.Y)
12      & ",");
13    Put_Line (" Z => "
14      & Integer'Image (P.Z)
15      & ")");
16  end Display;
17
18 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

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D;
5 begin
6   -- Positional component association
7   P := (0, 1, 2);
```

(continues on next page)

⁶¹ <https://learn.adacore.com/courses/intro-to-ada/chapters/records.html#intro-ada-record-aggregates>

(continued from previous page)

```

8
9   Display (P);
10
11  -- Named component association
12  P := (X => 3,
13        Y => 4,
14        Z => 5);
15
16  Display (P);
17  end Show_Record_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
 MD5: f4c4cff950e31a633ab4e2ae3d21ddc7b

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

```

1  with Points; use Points;
2
3  procedure Show_Record_Aggregates is
4     P : Point_3D;
5  begin
6     -- Positional and named component associations
7     P := (3, 4,
8           Z => 5);
9
10    Display (P);
11  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

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D;
5 begin
6   -- Multiple component selection
7   P := (X | Y => 5,
8        Z   => 6);
9
10  Display (P);
11 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](#)⁶²
-

4.2.1 <>

We can use the `<>` syntax to tell the compiler to use the default value for specific components. However, if there's no default value for specific components, that component isn't initialized to a known value. For example:

Listing 9: show_record_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D;
5 begin
6   P := (0, 1, 2);
7   Display (P);
8
9   -- Specifying X component.
10  P := (X => 42,
11       Y => <>,
12       Z => <>);
13  Display (P);
14
```

(continues on next page)

⁶² <http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html>

(continued from previous page)

```

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

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
MD5: 25145e7cba5a566c518ac4218e550899

Runtime output

```

(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

```

1  package Points is
2
3     type Point_3D is record
4         X : Integer := 10;
5         Y : Integer := 20;
6         Z : Integer := 30;
7     end record;
8
9     procedure Display (P : Point_3D);
10
11 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

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D := (0, 0, 0);
5 begin
6   -- Using default value for
7   -- all components
8   P := (X => <>,
9         Y => <>,
10        Z => <>);
11   Display (P);
12 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

```
1 package Points is
2
3   type Point_Value is new Float
4     with Default_Value => 99.9;
5
6   type Point_3D is record
7     X : Point_Value;
8     Y : Point_Value;
9     Z : Point_Value;
10  end record;
11
12  procedure Display (P : Point_3D);
13
14 end Points;
```

Listing 13: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
```

(continues on next page)

(continued from previous page)

```

3 package body Points is
4
5   procedure Display (P : Point_3D) is
6   begin
7     Put_Line ("X => "
8               & Point_Value'Image (P.X)
9               & ",");
10    Put_Line (" Y => "
11             & Point_Value'Image (P.Y)
12             & ",");
13    Put_Line (" Z => "
14             & Point_Value'Image (P.Z)
15             & ")");
16  end Display;
17
18 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

```

1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D := (0.0, 0.0, 0.0);
5 begin
6   -- Using default value of Point_Value
7   -- for all components
8   P := (X => <>,
9         Y => <>,
10        Z => <>);
11   Display (P);
12 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 `<>`.

4.2.2 others

Also, we can use the **others** selector to assign a value to all components that aren't explicitly mentioned in the aggregate. For example:

Listing 15: show_record_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D;
5 begin
6   -- Specifying X component;
7   -- using 42 for all
8   -- other components.
9   P := (X      => 42,
10        others => 100);
11   Display (P);
12
13   -- Specifying all components
14   P := (others => 256);
15   Display (P);
16 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

```
1 package Custom_Records is
2
3   type Integer_Float is record
4     A, B : Integer := 0;
5     Y, Z : Float   := 0.0;
6   end record;
7
8 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

```

1 with Custom_Records; use Custom_Records;
2
3 procedure Show_Record_Aggregates_Others is
4   Dummy : Integer_Float;
5 begin
6   -- ERROR: components selected by
7   --       others must be of same
8   --       type.
9   Dummy := (others => 256);
10 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

```

1 with Custom_Records; use Custom_Records;
2
3 procedure Show_Record_Aggregates_Others is
4   Dummy : Integer_Float;
5 begin
6   -- OK: components selected by
7   --       others have Integer type.
8   Dummy := (Y | Z => 256.0,
9             others => 256);
10 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

```
1 with Custom_Records; use Custom_Records;
2
3 procedure Show_Record_Aggregates_Others is
4   Dummy : Integer_Float;
5 begin
6   Dummy := (others => <>);
7 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.

4.2.3 Record discriminants

When a record type has discriminants, they must appear as components of an aggregate of that type. For example, consider this package:

Listing 20: points.ads

```
1 package Points is
2
3   type Point_Dimension is (Dim_1, Dim_2, Dim_3);
4
5   type Point (D : Point_Dimension) is record
6     case D is
7     when Dim_1 =>
8       X1      : Integer;
9     when Dim_2 =>
10      X2, Y2   : Integer;
11    when Dim_3 =>
12      X3, Y3, Z3 : Integer;
13    end case;
14  end record;
15
16  procedure Display (P : Point);
17
18 end Points;
```

Listing 21: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point) is
6   begin
7     Put_Line (Point_Dimension'Image (P.D));
8
9     case P.D is
10    when Dim_1 =>
11      Put_Line ("  (X => "
12                & Integer'Image (P.X1)
13                & ")");
```

(continues on next page)

(continued from previous page)

```

14     when Dim_2 =>
15         Put_Line (" (X => "
16                 & Integer'Image (P.X2)
17                 & ",");
18         Put_Line ("   Y => "
19                 & Integer'Image (P.Y2)
20                 & ")");
21     when Dim_3 =>
22         Put_Line (" (X => "
23                 & Integer'Image (P.X3)
24                 & ",");
25         Put_Line ("   Y => "
26                 & Integer'Image (P.Y3)
27                 & ",");
28         Put_Line ("   Z => "
29                 & Integer'Image (P.Z3)
30                 & ")");
31     end case;
32 end Display;
33
34 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

```

1  with Points; use Points;
2
3  procedure Show_Rec_Aggregate_Discriminant is
4      -- Positional component association
5      P1 : constant Point := (Dim_1, 0);
6
7      -- Named component association
8      P2 : constant Point := (D => Dim_2,
9                             X2 => 3,
10                            Y2 => 4);
11
12     -- Positional / named component association
13     P3 : constant Point := (Dim_3,
14                            X3 => 3,
15                            Y3 => 4,
16                            Z3 => 5);
17 begin
18     Display (P1);
19     Display (P2);

```

(continues on next page)

(continued from previous page)

```
20   Display (P3);
21 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.

4.3 Full coverage rules for Aggregates

Note: This section was originally written by Robert A. Duff and published as [Gem #1: Limited Types in Ada 2005](#)⁶³.

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

Listing 23: persons.ads

```
1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 package Persons is
5     type Years is new Natural;
6
7     type Person is record
8         Name : Unbounded_String;
9         Age : Years;
10    end record;
11 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:

⁶³ <https://www.adacore.com/gems/gem-1>

Listing 24: show_aggregate_init.adb

```

1 with Ada.Strings.Unbounded;
2 use  Ada.Strings.Unbounded;
3
4 with Persons; use Persons;
5
6 procedure Show_Aggregate_Init is
7
8     X : constant Person :=
9         (Name =>
10            To_Unbounded_String ("John Doe"),
11            Age  => 25);
12 begin
13     null;
14 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:

Listing 25: persons.ads

```

1 with Ada.Strings.Unbounded;
2 use  Ada.Strings.Unbounded;
3
4 package Persons is
5     type Years is new Natural;
6
7     type Person is record
8         Name      : Unbounded_String;
9         Age       : Natural;
10        Shoe_Size : Positive;
11    end record;
12 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 182) and case statements, but occasionally useful for *record aggregates* (page 170)):

Listing 26: show_aggregate_init_others.adb

```

1 with Ada.Strings.Unbounded;
2 use  Ada.Strings.Unbounded;
3
4 with Persons; use Persons;
5
6 procedure Show_Aggregate_Init_Others is
```

(continues on next page)

(continued from previous page)

```
7
8   X : constant Person :=
9       (Name =>
10        To_Unbounded_String ("John Doe"),
11        others => 25);
12 begin
13     null;
14 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 700).

4.4 Array aggregates

We've already discussed array aggregates in the [Introduction to Ada](#)⁶⁴ course. Therefore, this section just presents some details about this topic.

In the Ada Reference Manual

- [4.3.3 Array Aggregates](#)⁶⁵
-

4.4.1 Positional and named array aggregates

Note: The array aggregate syntax using brackets (e.g.: `[1, 2, 3]`), which we mention in this section, was introduced in Ada 2022.

Similar to *record aggregates* (page 170), array aggregates can be positional or named. Consider this package:

Listing 27: points.ads

```
1 package Points is
2
3     type Point_3D is array (1 .. 3) of Integer;
4
5     procedure Display (P : Point_3D);
6
7 end Points;
```

⁶⁴ <https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-array-type-declaration>

⁶⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-3.html>

Listing 28: points.adb

```

1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 package body Points is
6
7     procedure Display (P : Point_3D) is
8     begin
9         Put_Line ("X => "
10                & Integer'Image (P (1))
11                & ",");
12         Put_Line (" Y => "
13                & Integer'Image (P (2))
14                & ",");
15         Put_Line (" Z => "
16                & Integer'Image (P (3))
17                & " ");
18     end Display;
19
20 end Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: 7ed70d1c9685bc36900e1713619f3321

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

Listing 29: show_array_aggregates.adb

```

1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Array_Aggregates is
6     P : Point_3D;
7     begin
8         -- Positional component association
9         P := [0, 1, 2];
10
11         Display (P);
12
13         -- Named component association
14         P := [1 => 3,
15             2 => 4,
16             3 => 5];
17
18         Display (P);
19     end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: 5913ef6f43ea873de4e3f0760265de4b

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

```
1 pragma Ada_2022;  
2  
3 with Points; use Points;  
4  
5 procedure Show_Array_Aggregates is  
6   P : Point_3D := [others => 0];  
7 begin  
8   -- Positional component association  
9   P (2 .. 3) := [1, 2];  
10  
11   Display (P);  
12  
13   -- Named component association  
14   P (2 .. 3) := [1 => 3,  
15                 2 => 4];  
16  
17   Display (P);  
18 end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_  
↳Aggregates  
MD5: 8b36bd7638bd765f45693b78c5c7b872
```

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

```
1 pragma Ada_2012;  
2
```

(continues on next page)

(continued from previous page)

```

3 with Points; use Points;
4
5 procedure Show_Array_Aggregates is
6   P : Point_3D;
7 begin
8   -- Positional component association
9   P := (0, 1, 2);
10
11  Display (P);
12
13  -- Named component association
14  P := (1 => 3,
15        2 => 4,
16        3 => 5);
17
18  Display (P);
19 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.
 ↪Array_Aggregates
 MD5: 3d9f1fda006f1d566ae2743240568879

Runtime output

```

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

```

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

4.4.2 Null array aggregate

Note: This feature was introduced in Ada 2022.

We can also write null array aggregates: []. As the name implies, this kind of array aggregate doesn't have any components.

Consider this package:

Listing 32: integer_arrays.ads

```

1 package Integer_Arrays is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   procedure Display (A : Integer_Array);
7
8 end Integer_Arrays;

```

Listing 33: integer_arrays.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 package body Integer_Arrays is
6
7     procedure Display (A : Integer_Array) is
8     begin
9         Put_Line ("Length = "
10                & A'Length'Image);
11
12         Put_Line ("(");
13         for I in A'Range loop
14             Put (" "
15                & I'Image
16                & " => "
17                & A (I)'Image);
18             if I /= A'Last then
19                 Put_Line (",");
20             else
21                 New_Line;
22             end if;
23         end loop;
24         Put_Line (")");
25     end Display;
26
27 end Integer_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↳Aggregates_2
MD5: 412ebe9de1dfb9157f5379d31162554d
```

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

Listing 34: show_array_aggregates.adb

```
1 pragma Ada_2022;
2
3 with Integer_Arrays; use Integer_Arrays;
4
5 procedure Show_Array_Aggregates is
6     N : constant Integer_Array := [];
7 begin
8     Display (N);
9 end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↳Aggregates_2
MD5: 8cdb9a004ea16f716bf2e2ad5a65358e
```

Runtime output

```
Length = 0
(
)
```

In this example, when we call the Display procedure, we confirm that N doesn't have any components.

4.4.3 |, <>, others

We've seen the following syntactic elements when we were discussing *record aggregates* (page 170): |, <> and **others**. We can apply them to array aggregates as well:

Listing 35: show_array_aggregates.adb

```

1  pragma Ada_2022;
2
3  with Points; use Points;
4
5  procedure Show_Array_Aggregates is
6      P : Point_3D;
7  begin
8      -- All components have a value of zero.
9      P := [others => 0];
10
11     Display (P);
12
13     -- Both first and second components have
14     -- a value of three.
15     P := [1 | 2 => 3,
16           3     => 4];
17
18     Display (P);
19
20     -- The default value is used for the first
21     -- component, and all other components
22     -- have a value of five.
23     P := [1      => <>,
24           others => 5];
25
26     Display (P);
27 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: 053d4f162cc676b61d8e8a720321d40f

Runtime output

```

(X => 0,
 Y => 0,
 Z => 0)
(X => 3,
 Y => 3,
 Z => 4)
(X => 1358105832,
 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 197).

4.4.4 ..

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

Listing 36: show_array_aggregates.adb

```
1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Array_Aggregates is
6   P : Point_3D;
7 begin
8   -- All components have a value of zero.
9   P := [1 .. 3 => 0];
10
11   Display (P);
12
13   -- Both first and second components have
14   -- a value of three.
15   P := [1 .. 2 => 3,
16         3      => 4];
17
18   Display (P);
19
20   -- The default value is used for the first
21   -- component, and all other components
22   -- have a value of five.
23   P := [1      => <>,
24         2 .. 3 => 5];
25
26   Display (P);
27 end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↳Aggregates
MD5: bb36de6dcddf4b0bdcd5aa730f0988b1
```

Runtime output

```
(X => 0,
 Y => 0,
 Z => 0)
(X => 3,
 Y => 3,
 Z => 4)
(X => -1004229096,
 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.

4.4.5 Missing components

All aggregate components must have an associated value. If we don't specify a value for a certain component, an exception is raised:

Listing 37: show_array_aggregates.adb

```

1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Array_Aggregates is
6   P : Point_3D;
7 begin
8   P := [1 => 4];
9   -- ERROR: value of components at indices
10  --       2 and 3 are missing
11
12   Display (P);
13 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: 40d3a65f7fc0602782e548385ae07769

Build output

```

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

Runtime output

```
raised CONSTRAINT_ERROR : show_array_aggregates.adb:8 range check failed
```

We can use **others** to specify a value to all components that haven't been explicitly mentioned in the aggregate:

Listing 38: show_array_aggregates.adb

```

1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Array_Aggregates is
6   P : Point_3D;
7 begin
8   P := [1 => 4, others => 0];
9   -- OK: unspecified components have a
10  --       value of zero
11
12   Display (P);
13 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_↪Aggregates
MD5: 63b60de44e7c08eeae19a6a9117818f5

Runtime output

```
(X => 4,  
Y => 0,  
Z => 0)
```

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

Listing 39: show_array_aggregates.adb

```
1 pragma Ada_2022;  
2  
3 with Integer_Arrays; use Integer_Arrays;  
4  
5 procedure Show_Array_Aggregates is  
6   N1 : Integer_Array := [others => 0];  
7   -- ERROR: range is unknown  
8  
9   N2 : Integer_Array (1 .. 3) := [others => 0];  
10  -- OK: range is known  
11 begin  
12   Display (N1);  
13   Display (N2);  
14 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_↪Aggregates_2
MD5: 65b457e017a4eca6051aac777cc429f4

Build output

```
show_array_aggregates.adb:6:27: error: "others" choice not allowed here  
show_array_aggregates.adb:6:27: 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];`.

4.4.6 Iterated component association

Note: This feature was introduced in Ada 2022.

We can use an iterated component association to specify an aggregate. This is the general syntax:

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

Let's see a complete example:

Listing 40: show_array_aggregates.adb

```

1  pragma Ada_2022;
2
3  with Points; use Points;
4
5  procedure Show_Array_Aggregates is
6      P : Point_3D;
7  begin
8      -- All components have a value of zero
9      P := [for I in 1 .. 3 => 0];
10
11     Display (P);
12
13     -- Both first and second components have
14     -- a value of three
15     P := [for I in 1 .. 3 =>
16           (if I = 1 or I = 2
17            then 3
18             else 4)];
19
20     Display (P);
21
22     -- The first component has a value of 99
23     -- and all other components have a value
24     -- that corresponds to its index
25     P := [1 => 99,
26           for I in 2 .. 3 => I];
27
28     Display (P);
29 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: f11b3119e3fc1ece08f0b01d7e02576d

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

```
1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Array_Aggregates is
6   P : Point_3D := [for I in Point_3D'Range => I];
7 begin
8   -- Each component is doubled
9   P := [for E of P => E * 2];
10
11   Display (P);
12
13   -- Each component is increased
14   -- by one
15   P := [for E of P => E + 1];
16
17   Display (P);
18 end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↳Aggregates
MD5: b8c1878c1fa516005d1861f1a37c4fb0
```

Runtime output

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

In this example, we use iterators in different ways:

1. We write `[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.

4.4.7 Multidimensional array aggregates

So far, we've discussed one-dimensional array aggregates. We can also use the same constructs when dealing with multidimensional arrays. Consider, for example, this package:

Listing 42: matrices.ads

```
1 package Matrices is
2
3   type Matrix is array (Positive range <>,
4                         Positive range <>)
5                         of Integer;
6
7   procedure Display (M : Matrix);
8
9 end Matrices;
```

Listing 43: matrices.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  package body Matrices is
6
7      procedure Display (M : Matrix) is
8
9          procedure Display_Row (M : Matrix;
10                                 I : Integer) is
11              begin
12                  Put_Line ("  (");
13                  for J in M'Range (2) loop
14                      Put ("    "
15                          & J'Image
16                          & " => "
17                          & M (I, J)'Image);
18                      if J /= M'Last (2) then
19                          Put_Line (",");
20                      else
21                          New_Line;
22                      end if;
23                  end loop;
24                  Put ("  )");
25              end Display_Row;
26
27          begin
28              Put_Line ("Length (1) = "
29                          & M'Length (1)'Image);
30              Put_Line ("Length (2) = "
31                          & M'Length (2)'Image);
32
33              Put_Line ("(");
34              for I in M'Range (1) loop
35                  Display_Row (M, I);
36                  if I /= M'Last (1) then
37                      Put_Line (",");
38                  else
39                      New_Line;
40                  end if;
41              end loop;
42              Put_Line (")");
43
44          end Display;
45
46  end Matrices;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Matrix_
 ↳Aggregates
 MD5: 748c7c695dfef43d7d4926edf5ddd3ae

We can assign multidimensional aggregates to a matrix M using positional or named component association:

Listing 44: show_array_aggregates.adb

```
1 pragma Ada_2022;
2
3 with Matrices; use Matrices;
4
5 procedure Show_Array_Aggregates is
6   M : Matrix (1 .. 2, 1 .. 3);
7 begin
8   -- Positional component association
9   M := [[0, 1, 2],
10         [3, 4, 5]];
11
12   Display (M);
13
14   -- Named component association
15   M := [[1 => 3,
16         2 => 4,
17         3 => 5],
18         [1 => 6,
19         2 => 7,
20         3 => 8]];
21
22   Display (M);
23
24 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Matrix_→Aggregates
MD5: 78e1fad3b90c4f4d0f9d45f299e5ae10

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
  )
)
```

(continues on next page)

(continued from previous page)

)

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

```

1 package String_Lists is
2
3     type String_List is array (Positive range <>,
4                               Positive range <>)
5                               of Character;
6
7     procedure Display (SL : String_List);
8
9 end String_Lists;
```

Listing 46: string_lists.adb

```

1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 package body String_Lists is
6
7     procedure Display (SL : String_List) is
8
9         procedure Display_Row (SL : String_List;
10                                I : Integer) is
11
12             begin
13                 Put (" (");
14                 for J in SL'Range (2) loop
15                     Put (SL (I, J));
16                 end loop;
17                 Put ("");
18             end Display_Row;
19
20             begin
21                 Put_Line ("Length (1) = "
22                             & SL'Length (1)'Image);
23                 Put_Line ("Length (2) = "
24                             & SL'Length (2)'Image);
25
26                 Put_Line ("(");
27                 for I in SL'Range (1) loop
28                     Display_Row (SL, I);
29                     if I /= SL'Last (1) then
30                         Put_Line (",");
31                     else
32                         New_Line;
```

(continues on next page)

(continued from previous page)

```
32     end if;
33     end loop;
34     Put_Line ("");
35     end Display;
36
37 end String_Lists;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.String_
↳Aggregates
MD5: 87b2e593cab823218a39c07d85f40c22
```

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

Listing 47: show_array_aggregates.adb

```
1  pragma Ada_2022;
2
3  with String_Lists; use String_Lists;
4
5  procedure Show_Array_Aggregates is
6      SL : String_List (1 .. 2, 1 .. 3);
7  begin
8      -- Positional component association
9      SL := ["ABC",
10           "DEF"];
11
12     Display (SL);
13
14     -- Named component associations
15     SL := [[1 => 'A',
16            2 => 'B',
17            3 => 'C'],
18           [1 => 'D',
19            2 => 'E',
20            3 => 'F']];
21
22     Display (SL);
23
24     SL := [[1 => 'X',
25            2 => 'Y',
26            3 => 'Z'],
27           [others => ' ']];
28
29     Display (SL);
30 end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.String_
↳Aggregates
MD5: 82e376269e3be935d5cbd66202f26ec7
```

Runtime output

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

(continues on next page)

(continued from previous page)

```

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

4.4.8 <> and default values

As we indicated earlier, the <> syntax sets a component to its default value — if such a default value is available. If a default value isn't defined, however, the component will remain uninitialized, so that the behavior is undefined. Let's look at more complex example to illustrate this situation. Consider this package, for example:

Listing 48: points.ads

```

1 package Points is
2
3   subtype Point_Value is Integer;
4
5   type Point_3D is record
6     X, Y, Z : Point_Value;
7   end record;
8
9   procedure Display (P : Point_3D);
10
11  type Point_3D_Array is
12    array (Positive range <>) of Point_3D;
13
14  procedure Display (PA : Point_3D_Array);
15
16 end Points;

```

Listing 49: points.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_3D) is
6   begin
7     Put ("      (X => "
8       & Point_Value'Image (P.X)
9       & ",");
10    New_Line;
11    Put ("      Y => "

```

(continues on next page)

(continued from previous page)

```

12         & Point_Value'Image (P.Y)
13         & ",");
14     New_Line;
15     Put ("          Z => "
16         & Point_Value'Image (P.Z)
17         & ")");
18 end Display;
19
20 procedure Display (PA : Point_3D_Array) is
21 begin
22     Put_Line ("");
23     for I in PA'Range (1) loop
24         Put_Line (" "
25                 & Integer'Image (I)
26                 & " =>");
27         Display (PA (I));
28         if I /= PA'Last (1) then
29             Put_Line (",");
30         else
31             New_Line;
32         end if;
33     end loop;
34     Put_Line ("");
35 end Display;
36
37 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

```

1  pragma Ada_2022;
2
3  with Points; use Points;
4
5  procedure Show_Record_Aggregates is
6      PA : Point_3D_Array (1 .. 2);
7  begin
8      PA := [ (X => 3,
9              Y => 4,
10             Z => 5),
11             (X => 6,
12              Y => 7,
13              Z => 8) ];
14     Display (PA);
15
16     -- Array components are
17     -- uninitialized.
18     PA := [1 => <>,
19           2 => <>];
20     Display (PA);
21 end Show_Record_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Rec_Array_Aggregates
 MD5: 1dee9505222fe9837cd5aa3bf119ee3a

Runtime output

```
(
  1 =>
    (X => 3,
     Y => 4,
     Z => 5),
  2 =>
    (X => 6,
     Y => 7,
     Z => 8)
)
(
  1 =>
    (X => 0,
     Y => 0,
     Z => -278340979),
  2 =>
    (X => 32609,
     Y => -276826432,
     Z => 32609)
)
```

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

```
1 package Integer_Arrays is
2
3   type Value is new Integer
4     with Default_Value => 99;
5
6   type Integer_Array is
7     array (Positive range <>) of Value;
8
9   procedure Display (A : Integer_Array);
10
11 end Integer_Arrays;
```

Listing 52: show_array_aggregates.adb

```
1 pragma Ada_2022;
2
3 with Integer_Arrays; use Integer_Arrays;
4
5 procedure Show_Array_Aggregates is
6   N : Integer_Array (1 .. 4);
7 begin
8   N := [for I in N'Range => Value (I)];
9   Display (N);
10
11   N := [others => <>];
12   Display (N);
```

(continues on next page)

(continued from previous page)

```
13 end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
Aggregates_2
MD5: 17641d696172b052925d5549f53b9712
```

Runtime output

```
Length = 4
(
  1 => 1,
  2 => 2,
  3 => 3,
  4 => 4
)
Length = 4
(
  1 => 99,
  2 => 99,
  3 => 99,
  4 => 99
)
```

When writing an aggregate for the `Point_3D` type, any component that has `<>` gets the default value of the `Point` type (99):

For further reading...

Similarly, we could specify the `Default_Component_Value` aspect (which we discussed *earlier on* (page 64)) in the declaration of the array type:

Listing 53: `integer_arrays.ads`

```
1 package Integer_Arrays is
2
3   type Value is new Integer;
4
5   type Integer_Array is
6     array (Positive range <>) of Value
7     with Default_Component_Value => 9999;
8
9   procedure Display (A : Integer_Array);
10
11 end Integer_Arrays;
```

Listing 54: `show_array_aggregates.adb`

```
1 pragma Ada_2022;
2
3 with Integer_Arrays; use Integer_Arrays;
4
5 procedure Show_Array_Aggregates is
6   N : Integer_Array (1 .. 4);
7 begin
8   N := [for I in N'Range => Value (I)];
9   Display (N);
10
11   N := [others => <>];
```

(continues on next page)

(continued from previous page)

```
12   Display (N);
13 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates_2
 MD5: c6b38711937a1a7bbb92ddb4c207404e

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

Listing 55: integer_arrays.ads

```
1 package Integer_Arrays is
2
3   type Value is new Integer
4     with Default_Value => 99;
5
6   type Integer_Array is
7     array (Positive range <>) of Value
8     with Default_Component_Value => 9999;
9
10  procedure Display (A : Integer_Array);
11
12 end Integer_Arrays;
```

Listing 56: show_array_aggregates.adb

```
1 pragma Ada_2022;
2
3 with Integer_Arrays; use Integer_Arrays;
4
5 procedure Show_Array_Aggregates is
6   N : Integer_Array (1 .. 4);
7 begin
8   N := [for I in N'Range => Value (I)];
9   Display (N);
10
11  N := [others => <>];
```

(continues on next page)

(continued from previous page)

```
12   Display (N);  
13 end Show_Array_Aggregates;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_  
↳Aggregates_2  
MD5: c5b6d45576d59e2d3ba1634953c58b02
```

Runtime output

```
Length = 4  
(  
  1 => 1,  
  2 => 2,  
  3 => 3,  
  4 => 4  
)  
Length = 4  
(  
  1 => 9999,  
  2 => 9999,  
  3 => 9999,  
  4 => 9999  
)
```

Here, 9999 is used when we specify <> for a component.

4.5 Extension Aggregates

Extension aggregates provide a convenient way to express an aggregate for a type that extends — adds components to — some existing type (the "ancestor"). Although mainly a matter of convenience, an extension aggregate is essential when we want to express an aggregate for an extension of a private ancestor type, that is, when we don't have compile-time visibility to the ancestor type's components.

In the Ada Reference Manual

- [4.3.2 Extension Aggregates⁶⁶](#)
-

4.5.1 Assignments to objects of derived types

Before we discuss extension aggregates in more detail, though, let's start with a simple use-case. Let's say we have:

- an object A of tagged type T1, and
- an object B of tagged type T2, which extends T1.

We can initialize object B by:

- copying the T1 specific information from A to B, and
- initializing the T2 specific components of B.

⁶⁶ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-2.html>

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.

4.5.2 Example: Points

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

Listing 57: points.ads

```

1 package Points is
2
3   type Point_1D is tagged record
4     X : Float;
5   end record;
6
7   procedure Display (P : Point_1D);
8
9   type Point_2D is new Point_1D with record
10    Y : Float;
11  end record;
12
13  procedure Display (P : Point_2D);
14
15  type Point_3D is new Point_2D with record
16    Z : Float;
17  end record;
18
19  procedure Display (P : Point_3D);
20
21 end Points;

```

Listing 58: points.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_1D) is
6   begin
7     Put_Line ("X => " & P.X'Image & "");
8   end Display;
9
10  procedure Display (P : Point_2D) is
11  begin
12    Put_Line ("X => " & P.X'Image
13              & ", Y => " & P.Y'Image & "");
14  end Display;
15

```

(continues on next page)

(continued from previous page)

```
16 procedure Display (P : Point_3D) is
17 begin
18     Put_Line ("(X => " & P.X'Image
19             & ", Y => " & P.Y'Image
20             & ", Z => " & P.Z'Image & ")");
21 end Display;
22
23 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

```
1 with Points; use Points;
2
3 procedure Show_Points is
4     P_1D : Point_1D;
5     P_2D : Point_2D;
6 begin
7     P_1D := (X => 0.5);
8     Display (P_1D);
9
10    Point_1D (P_2D) := P_1D;
11    -- Equivalent to: "P_2D.X := P_1D.X;"
12
13    P_2D.Y := 0.7;
14
15    Display (P_2D);
16 end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_
↳Aggregate_Points
MD5: 68ae6fa8e6f779aeb97085bd75e082
```

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.

4.5.3 Using extension aggregates

Note that, in the assignment to `P_1D`, we use a record aggregate. Extension aggregates are similar to record aggregates, but they include the `with` keyword — for example: `(Obj1 with Y => 0.5)`. This allows us to assign to an object with information from another object `Obj1` of a parent type and, in the same expression, set the value of the `Y` component of the type extension.

Let's rewrite the previous `Show_Points` procedure using extension aggregates:

Listing 60: `show_points.adb`

```

1 with Points; use Points;
2
3 procedure Show_Points is
4   P_1D : Point_1D;
5   P_2D : Point_2D;
6 begin
7   P_1D := (X => 0.5);
8   Display (P_1D);
9
10  P_2D := (P_1D with Y => 0.7);
11  Display (P_2D);
12 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);
```


4.5.4 More extension aggregates

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

Listing 61: `show_points.adb`

```
1 with Points; use Points;
2
3 procedure Show_Points is
4   P_1D : Point_1D;
5   P_2D : Point_2D;
6   P_3D : Point_3D;
7 begin
8   P_1D := (X => 0.5);
9   Display (P_1D);
10
11   P_2D := (P_1D with Y => 0.7);
12   Display (P_2D);
13
14   P_3D := (P_2D with Z => 0.3);
15   Display (P_3D);
16
17   P_3D := (P_1D with Y | Z => 0.1);
18   Display (P_3D);
19 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.)

4.5.5 with others

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

Listing 62: `show_points.adb`

```
1 with Points; use Points;
2
3 procedure Show_Points is
4   P_1D : Point_1D;
5   P_2D : Point_2D;
```

(continues on next page)

(continued from previous page)

```

6   P_3D : Point_3D;
7   begin
8     P_1D := (X => 0.5);
9     P_2D := (P_1D with Y => 0.7);
10
11    -- Initialize P_3D with P_1D and set other
12    -- components to 0.6.
13    --
14    P_3D := (P_1D with others => 0.6);
15    Display (P_3D);
16
17    -- Initialize P_3D with P_2D, and other
18    -- components with their default value.
19    --
20    P_3D := (P_2D with others => <>);
21    Display (P_3D);
22 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 => 5.93170E-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 => <>**.

4.5.6 with null record

We can also use extension aggregates with null records. Let's focus on the P_3D_Ext object of Point_3D_Ext type. This object is declared in the Show_Points procedure of the next code example.

Listing 63: points-extensions.ads

```

1 package Points.Extensions is
2
3   type Point_3D_Ext is new
4     Point_3D with null record;
5
6 end Points.Extensions;

```

Listing 64: show_points.adb

```
1 with Points;           use Points;
2 with Points.Extensions; use Points.Extensions;
3
4 procedure Show_Points is
5     P_3D      : Point_3D;
6     P_3D_Ext : Point_3D_Ext;
7 begin
8     P_3D := (X => 0.0, Y => 0.5, Z => 0.4);
9
10    P_3D_Ext := (P_3D with null record);
11    Display (P_3D_Ext);
12 end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: 8ec3ddb3a1f2a6e550ac4d622e97124c

Runtime output

```
(X => 0.000000E+00, Y => 5.000000E-01, Z => 4.000000E-01)
```

The `P_3D_Ext` object is of `Point_3D_Ext` type, which is declared in the `Points.Extensions` package and derived from the `Point_3D` type. Note that we're not extending `Point_3D_Ext` with new components, but using a null record instead in the declaration. Therefore, as the `Point_3D_Ext` type doesn't own any new components, we just write `(P_3D with null record)` to initialize the `P_3D_Ext` object.

4.5.7 Extension aggregates and descendent types

In the examples above, we've been initializing objects of descendent types by using objects of ascending types in extension aggregates. We could, however, do the opposite and initialize objects of ascending types using objects of descendent type in extension aggregates. Consider this code example:

Listing 65: show_points.adb

```
1 with Points; use Points;
2
3 procedure Show_Points is
4     P_2D : Point_2D;
5     P_3D : Point_3D;
6 begin
7     P_3D := (X => 0.5, Y => 0.7, Z => 0.3);
8     Display (P_3D);
9
10    P_2D := (Point_1D (P_3D) with Y => 0.3);
11    Display (P_2D);
12 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.

4.6 Delta Aggregates

Note: This feature was introduced in Ada 2022.

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

4.6.1 Delta Aggregates for Tagged Records

Let's reuse the `Points` package from a previous example:

Listing 66: `points.ads`

```
1 package Points is
2
3     type Point_1D is tagged record
4         X : Float;
5     end record;
6
7     type Point_2D is new Point_1D with record
8         Y : Float;
9     end record;
10
11    type Point_3D is new Point_2D with record
12        Z : Float;
13    end record;
14
15    procedure Display (P : Point_3D);
16
17 end Points;
```

Listing 67: `points.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5     procedure Display (P : Point_3D) is
6     begin
7         Put_Line ("(X => " & P.X'Image
8                 & ", Y => " & P.Y'Image
```

(continues on next page)

(continued from previous page)

```
9         & ", Z => " & P.Z'Image & "));  
10     end Display;  
11  
12 end Points;
```

Listing 68: show_points.adb

```
1  pragma Ada_2022;  
2  
3  with Points; use Points;  
4  
5  procedure Show_Points is  
6      P1, P2, P3 : Point_3D;  
7  begin  
8      P1 := (X => 0.5, Y => 0.7, Z => 0.3);  
9      Display (P1);  
10  
11     P2 := (P1 with delta X => 1.0);  
12     Display (P2);  
13  
14     P3 := (P1 with delta X => 0.2, Y => 0.3);  
15     Display (P3);  
16 end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_Aggregates_Tagged
MD5: affbd4304a683699de48fc44db44f09e

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

```
1  pragma Ada_2022;  
2  
3  with Points; use Points;  
4  
5  procedure Show_Points is  
6      P_3D : Point_3D;  
7  
8      function Reset (P_2D : Point_2D'Class)  
9          return Point_2D'Class is  
10         ((P_2D with delta X | Y => 0.0));  
11  
12  begin  
13     P_3D := [X => 0.1, Y => 0.2, Z => 0.3];  
14     Display (P_3D);  
15  
16     P_3D := Point_3D (Reset (P_3D));  
17     Display (P_3D);  
18
```

(continues on next page)

(continued from previous page)

```
19
20 end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
 ↳Aggregates_Tagged
 MD5: 30e62d564d1b35829a5002223966c101

Runtime output

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

In this example, the Reset function returns an object of Point_2D'Class where all components of Point_2D'Class type are zero. We call the Reset function for the P_3D object of Point_3D type, so that only the Z component remains untouched.

Note that we use the syntax X | Y in the body of the Reset function and assign the same value to both components.

For further reading...

We could have implemented Reset as a procedure — in this case, without using delta aggregates:

Listing 70: show_points.adb

```
1 with Points; use Points;
2
3 procedure Show_Points is
4
5     P_3D : Point_3D;
6
7     procedure Reset
8         (P_2D : in out Point_2D'Class) is
9     begin
10        Point_2D (P_2D) := (others => 0.0);
11    end Reset;
12
13 begin
14     P_3D := (X => 0.1, Y => 0.2, Z => 0.3);
15     Display (P_3D);
16
17     Reset (P_3D);
18     Display (P_3D);
19
20 end Show_Points;
```

4.6.2 Delta Aggregates for Non-Tagged Records

The examples above use tagged types. We can also use delta aggregates with non-tagged types. Let's rewrite the Points package and convert Point_3D to a non-tagged record type.

Listing 71: points.ads

```
1 package Points is
2
3   type Point_3D is record
4     X : Float;
5     Y : Float;
6     Z : Float;
7   end record;
8
9   procedure Display (P : Point_3D);
10
11 end Points;
```

Listing 72: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_3D) is
6   begin
7     Put_Line ("(X => " & P.X'Image
8               & ", Y => " & P.Y'Image
9               & ", Z => " & P.Z'Image & ")");
10  end Display;
11
12 end Points;
```

Listing 73: show_points.adb

```
1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Points is
6   P1, P2, P3 : Point_3D;
7 begin
8   P1 := (X => 0.5, Y => 0.7, Z => 0.3);
9   Display (P1);
10
11   P2 := (P1 with delta X => 1.0);
12   Display (P2);
13
14   P3 := (P1 with delta X => 0.2, Y => 0.3);
15   Display (P3);
16 end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↳Aggregates_Non_Tagged
MD5: 71a3b76ee1988ddea7246d0b8f897865
```

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.

4.6.3 Delta Aggregates for Arrays

We can use delta aggregates for arrays. Let's change the declaration of Point_3D and use an array to represent a 3-dimensional point:

Listing 74: points.ads

```
1 package Points is
2
3   type Float_Array is
4     array (Positive range <>) of Float;
5
6   type Point_3D is new Float_Array (1 .. 3);
7
8   procedure Display (P : Point_3D);
9
10 end Points;
```

Listing 75: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_3D) is
6   begin
7     Put "(";
8     for I in P'Range loop
9       Put (I'Image
10          & " => "
11          & P (I)'Image);
12     end loop;
13     Put_Line ("");
14   end Display;
15
16 end Points;
```

Listing 76: show_points.adb

```
1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Points is
6   P1, P2, P3 : Point_3D;
7 begin
8   P1 := [0.5, 0.7, 0.3];
9   Display (P1);
10
11   P2 := [P1 with delta 1 => 1.0];
12   Display (P2);
13
```

(continues on next page)

(continued from previous page)

```
14 P3 := [P1 with delta 1 => 0.2, 2 => 0.3];
15 -- Alternatively:
16 -- P3 := [P1 with delta 1 .. 2 => 0.2, 0.3];
17
18 Display (P3);
19 end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↳Aggregates_Array
MD5: d32ba51746d7db9cd30f183e64ab0017
```

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 we 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`

```
1 pragma Ada_2022;
2
3 with Points; use Points;
4
5 procedure Show_Points is
6   P1, P3 : Point_3D;
7 begin
8   P1 := [0.5, 0.7, 0.3];
9   Display (P1);
10
11   P3 := [P1 with delta
12         P3'First + 1 .. P3'Last => 0.0];
13   Display (P3);
14 end Show_Points;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↳Aggregates_Array
MD5: 6d1db1634c42a885f7bfce7f7eccc359
```

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

```
1 package Float_Arrays is
2
3     type Float_Array is
4         array (Positive range <>) of Float;
5
6     procedure Display (P : Float_Array);
7
8 end Float_Arrays;
```

Listing 79: float_arrays.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Float_Arrays is
4
5     procedure Display (P : Float_Array) is
6     begin
7
8         Put ("(");
9         for I in P'Range loop
10            Put (I'Image
11                & " => "
12                & P (I)'Image);
13        end loop;
14        Put_Line (")");
15
16    end Display;
17
18 end Float_Arrays;
```

Listing 80: show_multiple_delta_slices.adb

```
1 pragma Ada_2022;
2
3 with Float_Arrays; use Float_Arrays;
4
5 procedure Show_Multiple_Delta_Slices is
6
7     P1, P2 : Float_Array (1 .. 5);
8
9 begin
10    P1 := [1.0, 2.0, 3.0, 4.0, 5.0];
11    Display (P1);
12
13    P2 := [P1 with delta
14          P2'First + 1 .. P2'Last - 2 => 0.0,
15          P2'Last - 1 .. P2'Last => 0.2];
16    Display (P2);
17 end Show_Multiple_Delta_Slices;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
↳Aggregates_Array
MD5: 4c2860616777428618d1100280699ec2
```

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](#)⁶⁷
-

⁶⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-4.html>

ARRAYS

5.1 Unconstrained Arrays

In the [Introduction to Ada course](#)⁶⁸, we've seen that we can declare array types whose bounds are not fixed: in that case, the bounds are provided when creating objects of those types. For example:

Listing 1: measurement_defs.ads

```
1 package Measurement_Defs is
2
3     type Measurements is
4       array (Positive range <>) of Float;
5       --      ^ Bounds are of type Positive,
6       --      but not known at this point.
7
8 end Measurement_Defs;
```

Listing 2: show_measurements.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Measurement_Defs; use Measurement_Defs;
4
5 procedure Show_Measurements is
6     M : Measurements (1 .. 10);
7     --      ^ Providing bounds here!
8 begin
9     Put_Line ("First index: " & M'First'Image);
10    Put_Line ("Last index: " & M'Last'Image);
11 end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Unconstrained_Arrays.Unconstrained_
↳Array_Example
MD5: a5cdc74dd61e36476431cf675452d1d5
```

Build output

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

Runtime output

⁶⁸ <https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-types>

```
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.

The [Introduction to Ada course](#)⁶⁹ also highlights the fact that the bounds are fixed once an object is declared:

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

In the `Show_Measurements` procedure above, once we declare `M`, its bounds are fixed for the whole lifetime of `M`. We cannot *add* another component to this array. In other words, `M` will have 10 components for its whole lifetime.

In the Ada Reference Manual

- [3.6 Array Types](#)⁷⁰
-

5.1.1 Unconstrained Arrays vs. Vectors

If you need, however, the flexibility of increasing the length of an array, you could use vectors instead. This is how we could rewrite the previous example using vectors:

Listing 3: `measurement_defs.ads`

```
1 with Ada.Containers; use Ada.Containers;
2 with Ada.Containers.Vectors;
3
4 package Measurement_Defs is
5
6     package Vectors is new Ada.Containers.Vectors
7         (Index_Type => Positive,
8          Element_Type => Float);
9
10    subtype Measurements is Vectors.Vector;
11
12 end Measurement_Defs;
```

Listing 4: `show_measurements.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Measurement_Defs; use Measurement_Defs;
4
5 procedure Show_Measurements is
6     use Measurement_Defs.Vectors;
7
8     M : Measurements := To_Vector (10);
9     -- ^ Creating 10-element
10    -- vector.
11 begin
```

(continues on next page)

⁶⁹ <https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-type-instance-bound>

⁷⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

(continued from previous page)

```

12 Put_Line ("First index: "
13           & M.First_Index'Image);
14 Put_Line ("Last index: "
15           & M.Last_Index'Image);
16
17 Put_Line ("Adding element...");
18 M.Append (1.0);
19
20 Put_Line ("First index: "
21           & M.First_Index'Image);
22 Put_Line ("Last index: "
23           & M.Last_Index'Image);
24 end Show_Measurements;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Unconstrained_Arrays.Unconstrained_Array_Example
 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](#)⁷¹
 - [A.18.2 The Generic Package Containers.Vectors](#)⁷²
-

5.2 Multidimensional Arrays

So far, we've discussed unidimensional arrays, since they are very common in Ada. However, Ada also supports multidimensional arrays using the same facilities as for unidimensional arrays. For example, we can use the `First`, `Last`, `Range` and `Length` attributes for each dimension of a multidimensional array. This section presents more details on this topic.

To create a multidimensional array, we simply separate the ranges of each dimension with a comma. The following example presents the one-dimensional array A1, the two-dimensional array A2 and the three-dimensional array A3:

⁷¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

⁷² <http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html>

Listing 5: multidimensional_arrays_decl.ads

```
1 package Multidimensional_Arrays_Decl is
2
3     A1 : array (1 .. 10) of Float;
4     A2 : array (1 .. 5, 1 .. 10) of Float;
5         --         ^ first dimension
6         --         ^ second dimension
7     A3 : array (1 .. 2, 1 .. 5, 1 .. 10) of Float;
8         --         ^ first dimension
9         --         ^ second dimension
10        --         ^ third dimension
11 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 6: show_multidimensional_arrays.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Multidimensional_Arrays_Decl;
4 use Multidimensional_Arrays_Decl;
5
6 procedure Show_Multidimensional_Arrays is
7 begin
8     Put_Line ("A1'Length:      "
9             & A1'Length'Image);
10    Put_Line ("A1'Length (1): "
11            & A1'Length (1)'Image);
12    Put_Line ("A2'Length (1): "
13            & A2'Length (1)'Image);
14    Put_Line ("A2'Length (2): "
15            & A2'Length (2)'Image);
16    Put_Line ("A3'Length (1): "
17            & A3'Length (1)'Image);
18    Put_Line ("A3'Length (2): "
19            & A3'Length (2)'Image);
20    Put_Line ("A3'Length (3): "
21            & A3'Length (3)'Image);
22 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 7: measurement_defs.ads

```
1 package Measurement_Defs is
2
3   type Days is
4     (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
5
6   type Hours is range 0 .. 11;
7
8   subtype Measurement is Float;
9
10  type Measurements is
11    array (Days, Hours) of Measurement;
12
13  procedure Show_Indices (M : Measurements);
14
15  procedure Show_Values (M : Measurements);
16
17  procedure Reset (M : out Measurements);
18
19 end Measurement_Defs;
```

Listing 8: measurement_defs.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 package body Measurement_Defs is
4
5   procedure Show_Indices (M : Measurements) is
6   begin
7     Put_Line ("---- Indices ----");
8
```

(continues on next page)

(continued from previous page)

```

 9     for D in M'Range (1) loop
10         Put (D'Image & " ");
11
12         for H in M'First (2) ..
13             M'Last (2) - 1
14         loop
15             Put (H'Image & " ");
16         end loop;
17         Put_Line (M'Last (2)'Image);
18     end loop;
19 end Show_Indices;
20
21 procedure Show_Values (M : Measurements) is
22     package H_IO is
23         new Ada.Text_IO.Integer_IO (Hours);
24     package M_IO is
25         new Ada.Text_IO.Float_IO (Measurement);
26
27     procedure Set_IO_Defaults is
28     begin
29         H_IO.Default_Width := 5;
30
31         M_IO.Default_Fore := 1;
32         M_IO.Default_Aft := 2;
33         M_IO.Default_Exp := 0;
34     end Set_IO_Defaults;
35 begin
36     Set_IO_Defaults;
37
38     Put_Line ("---- Values ----");
39     Put (" ");
40     for H in M'Range (2) loop
41         H_IO.Put (H);
42     end loop;
43     New_Line;
44
45     for D in M'Range (1) loop
46         Put (D'Image & " ");
47
48         for H in M'Range (2) loop
49             M_IO.Put (M (D, H));
50             Put (" ");
51         end loop;
52         New_Line;
53     end loop;
54 end Show_Values;
55
56 procedure Reset (M : out Measurements) is
57 begin
58     M := (others => (others => 0.0));
59 end Reset;
60
61 end Measurement_Defs;

```

Listing 9: show_measurements.adb

```

1 with Measurement_Defs; use Measurement_Defs;
2
3 procedure Show_Measurements is
4     M : Measurements;
5 begin

```

(continues on next page)

(continued from previous page)

```

6   Reset (M);
7   Show_Indices (M);
8   Show_Values (M);
9   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](#)⁷³

⁷³ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

5.2.1 Unconstrained Multidimensional Arrays

Previously, we've discussed unconstrained arrays for the unidimensional case. It's possible to declare unconstrained multidimensional arrays as well. For example:

Listing 10: multidimensional_arrays_decl.ads

```
1 package Multidimensional_Arrays_Decl is
2
3     type F1 is array (Positive range <>) of Float;
4     type F2 is array (Positive range <>,
5                     Positive range <>) of Float;
6     type F3 is array (Positive range <>,
7                     Positive range <>,
8                     Positive range <>) of Float;
9
10 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 11: show_multidimensional_arrays.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Multidimensional_Arrays_Decl;
4 use Multidimensional_Arrays_Decl;
5
6 procedure Show_Multidimensional_Arrays is
7     A1 : F1 (1 .. 2);
8     A2 : F2 (1 .. 4, 10 .. 20);
9     A3 : F3 (2 .. 3, 1 .. 5, 1 .. 2);
10 begin
11     Put_Line ("A1'Length (1): "
12             & A1'Length (1)'Image);
13     Put_Line ("A2'Length (1): "
14             & A2'Length (1)'Image);
15     Put_Line ("A2'Length (2): "
16             & A2'Length (2)'Image);
17     Put_Line ("A3'Length (1): "
18             & A3'Length (1)'Image);
19     Put_Line ("A3'Length (2): "
20             & A3'Length (2)'Image);
21     Put_Line ("A3'Length (3): "
22             & A3'Length (3)'Image);
23 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
```

5.2.2 Arrays of arrays

It's important to distinguish between multidimensional arrays and arrays of arrays. Both are supported in Ada, but they're very distinct from each other. We can create an array of an array by first specifying a one-dimensional array type T1, and then specifying another one-dimensional array type T2 where each component of T2 is of T1 type:

Listing 12: array_of_arrays_decl.ads

```
1 package Array_Of_Arrays_Decl is
2
3     type T1 is
4         array (Positive range <>) of Float;
5
6     type T2 is
7         array (Positive range <>) of T1 (1 .. 10);
8         --           ^^^^^^^
9         --           bounds must be set!
10
11 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 13: measurement_defs.ads

```
1 package Measurement_Defs is
2
3     type Days is
4         (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
5
6     type Hours is range 0 .. 11;
7
8     subtype Measurement is Float;
```

(continues on next page)

(continued from previous page)

```

9
10 type Hourly_Measurements is
11     array (Hours) of Measurement;
12
13 type Measurements is
14     array (Days) of Hourly_Measurements;
15
16 procedure Show_Indices (M : Measurements);
17
18 procedure Show_Values (M : Measurements);
19
20 procedure Reset (M : out Measurements);
21
22 end Measurement_Defs;

```

Listing 14: measurement_defs.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 package body Measurement_Defs is
4
5     procedure Show_Indices (M : Measurements) is
6     begin
7         Put_Line ("---- Indices ----");
8
9         for D in M'Range loop
10            Put (D'Image & " ");
11
12            for H in M (D)'First ..
13                M (D)'Last - 1
14            loop
15                Put (H'Image & " ");
16            end loop;
17            Put_Line (M (D)'Last'Image);
18        end loop;
19    end Show_Indices;
20
21    procedure Show_Values (M : Measurements) is
22    package H_IO is
23        new Ada.Text_IO.Integer_IO (Hours);
24    package M_IO is
25        new Ada.Text_IO.Float_IO (Measurement);
26
27        procedure Set_IO_Defaults is
28        begin
29            H_IO.Default_Width := 5;
30
31            M_IO.Default_Fore  := 1;
32            M_IO.Default_Aft   := 2;
33            M_IO.Default_Exp   := 0;
34        end Set_IO_Defaults;
35    begin
36        Set_IO_Defaults;
37
38        Put_Line ("---- Values ----");
39        Put (" ");
40        for H in M (M'First)'Range loop
41            H_IO.Put (H);
42        end loop;
43        New_Line;
44

```

(continues on next page)

(continued from previous page)

```

45     for D in M'Range loop
46         Put (D'Image & " ");
47
48         for H in M (D)'Range loop
49             M_IO.Put (M (D) (H));
50             Put (" ");
51         end loop;
52         New_Line;
53     end loop;
54 end Show_Values;
55
56 procedure Reset (M : out Measurements) is
57 begin
58     M := (others => (others => 0.0));
59 end Reset;
60
61 end Measurement_Defs;

```

Listing 15: show_measurements.adb

```

1 with Measurement_Defs; use Measurement_Defs;
2
3 procedure Show_Measurements is
4     M : Measurements;
5 begin
6     Reset (M);
7     Show_Indices (M);
8     Show_Values (M);
9 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))`.

STRINGS

6.1 Wide and Wide-Wide Strings

We've seen many source-code examples so far that includes strings. In most of them, we were using the standard string type: **String**. This type is useful for the common use-case of displaying messages or dealing with information in plain English. Here, we define "plain English" as the use of the language that avoids French accents or German umlaut, for example, and doesn't make use of any characters in non-Latin alphabets.

There are two additional string types in Ada: **Wide_String**, and **Wide_Wide_String**. These types are particularly important when dealing with textual information in non-standard English, or in various other languages, non-Latin alphabets and special symbols.

These string types use different bit widths for their characters. This becomes more apparent when looking at the type definitions:

```
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 1: show_wide_char_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Wide_Char_Types is
4 begin
5   Put_Line ("Character'Size:           ")
6             & Integer'Image
7             (Character'Size));
8   Put_Line ("Wide_Character'Size:       ")
```

(continues on next page)

(continued from previous page)

```
9         & Integer'Image
10         (Wide_Character'Size));
11     Put_Line ("Wide_Wide_Character'Size: "
12             & Integer'Image
13             (Wide_Wide_Character'Size));
14 end Show_Wide_Char_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Wide_Char_
↳Types
MD5: a0e9fb9e8d43e9fa707dc8c57f7562f8
```

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 2: show_wide_string_types.adb

```
1  with Ada.Text_IO;
2  with Ada.Wide_Text_IO;
3  with Ada.Wide_Wide_Text_IO;
4
5  procedure Show_Wide_String_Types is
6      package TI   renames Ada.Text_IO;
7      package WTI  renames Ada.Wide_Text_IO;
8      package WWTI renames Ada.Wide_Wide_Text_IO;
9
10     S   : constant String      := "hello";
11     WS  : constant Wide_String := "hello";
12     WWS : constant Wide_Wide_String := "hello";
13 begin
14     TI.Put_Line ("String:      " & S);
15     TI.Put_Line ("Length:      "
16                 & Integer'Image (S'Length));
17     TI.Put_Line ("Size:          "
18                 & Integer'Image (S'Size));
19     TI.Put_Line ("Component_Size:  "
20                 & Integer'Image
21                 (S'Component_Size));
22     TI.Put_Line ("-----");
23
24     WTI.Put_Line ("Wide string:      " & WS);
25     TI.Put_Line ("Length:          "
26                 & Integer'Image (WS'Length));
27     TI.Put_Line ("Size:            "
28                 & Integer'Image (WS'Size));
29     TI.Put_Line ("Component_Size:  "
30                 & Integer'Image
31                 (WS'Component_Size));
32     TI.Put_Line ("-----");
33
34     WWTI.Put_Line ("Wide-wide string: " & WWS);
35     TI.Put_Line ("Length:          "
36                 & Integer'Image (WWS'Length));
37     TI.Put_Line ("Size:            "
38                 & Integer'Image (WWS'Size));
```

(continues on next page)

(continued from previous page)

```

39   TI.Put_Line ("Component_Size:  "
40               & Integer'Image
41               (WWS'Component_Size));
42   TI.Put_Line ("-----");
43 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](#)⁷⁴
-

6.1.1 Text I/O

Note that, in the previous example, we were using different versions of the `Ada.Text_IO` package depending on the string type we were using:

- `Ada.Text_IO` for objects of **String** type,
- `Ada.Wide_Text_IO` for objects of **Wide_String** type,
- `Ada.Wide_Wide_Text_IO` for objects of **Wide_Wide_String** type.

In that example, we were also using package renaming to differentiate among those packages.

Similarly, there are different versions of text I/O packages for individual types. For example, if we want to display the value of a **Long_Integer** variable based on the **Wide_Wide_String**

⁷⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-6-3.html>

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:

Scalar Type	Text I/O Packages
Integer	<ul style="list-style-type: none">• <code>Ada.Integer_Text_IO</code>• <code>Ada.Integer_Wide_Text_IO</code>• <code>Ada.Integer_Wide_Wide_Text_IO</code>
Long_Integer	<ul style="list-style-type: none">• <code>Ada.Long_Integer_Text_IO</code>• <code>Ada.Long_Integer_Wide_Text_IO</code>• <code>Ada.Long_Integer_Wide_Wide_Text_IO</code>
Long_Long_Integer	<ul style="list-style-type: none">• <code>Ada.Long_Long_Integer_Text_IO</code>• <code>Ada.Long_Long_Integer_Wide_Text_IO</code>• <code>Ada.Long_Long_Integer_Wide_Wide_Text_IO</code>
Float	<ul style="list-style-type: none">• <code>Ada.Float_Text_IO</code>• <code>Ada.Float_Wide_Text_IO</code>• <code>Ada.Float_Wide_Wide_Text_IO</code>
Long_Float	<ul style="list-style-type: none">• <code>Ada.Long_Float_Text_IO</code>• <code>Ada.Long_Float_Wide_Text_IO</code>• <code>Ada.Long_Float_Wide_Wide_Text_IO</code>
Long_Long_Float	<ul style="list-style-type: none">• <code>Ada.Long_Long_Float_Text_IO</code>• <code>Ada.Long_Long_Float_Wide_Text_IO</code>• <code>Ada.Long_Long_Float_Wide_Wide_Text_IO</code>

Also, there are different versions of the generic packages `Integer_IO` and `Float_IO`:

Scalar Type	Text I/O Packages
Integer types	<ul style="list-style-type: none">• <code>Ada.Text_IO.Integer_IO</code>• <code>Ada.Wide_Text_IO.Integer_IO</code>• <code>Ada.Wide_Wide_Text_IO.Integer_IO</code>
Real types	<ul style="list-style-type: none">• <code>Ada.Text_IO.Float_IO</code>• <code>Ada.Wide_Text_IO.Float_IO</code>• <code>Ada.Wide_Wide_Text_IO.Float_IO</code>

In the Ada Reference Manual

- [A.10 Text Input-Output⁷⁵](#)

⁷⁵ <http://www.ada-auth.org/standards/22rm/html/RM-A-10.html>

- A.10.1 The Package Text_IO⁷⁶
- A.10.8 Input-Output for Integer Types⁷⁷
- A.10.9 Input-Output for Real Types⁷⁸
- A.11 Wide Text Input-Output and Wide Wide Text Input-Output⁷⁹

6.1.2 Wide and Wide-Wide String Handling

As we've just seen, we have different versions of the Ada.Text_IO package. The same applies to string handling packages. As we've seen in the [Introduction to Ada course](#)⁸⁰, 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](#)⁸¹ from the Introduction to Ada course, which we adapted for wide-wide strings:

Listing 3: show_find_words.adb

```

1  with Ada.Strings; use Ada.Strings;
2
3  with Ada.Strings.Wide_Wide_Fixed;
4  use  Ada.Strings.Wide_Wide_Fixed;
5
6  with Ada.Strings.Wide_Wide_Maps;
7  use  Ada.Strings.Wide_Wide_Maps;
8
9  with Ada.Wide_Wide_Text_IO;
10 use  Ada.Wide_Wide_Text_IO;
11
12 procedure Show_Find_Words is
13
14     S   : constant Wide_Wide_String :=
15         "Hello" & 3 * " World";
16     F   : Positive;
17     L   : Natural;
18     I   : Natural := 1;
19
20     Whitespace : constant
21         Wide_Wide_Character_Set :=
22         To_Set ( ' ');
23 begin
24     Put_Line ("String: " & S);
25     Put_Line ("String length: "
26               & Integer'Wide_Wide_Image

```

(continues on next page)

⁷⁶ <http://www.ada-auth.org/standards/22rm/html/RM-A-10-1.html>

⁷⁷ <http://www.ada-auth.org/standards/22rm/html/RM-A-10-8.html>

⁷⁸ <http://www.ada-auth.org/standards/22rm/html/RM-A-10-9.html>

⁷⁹ <http://www.ada-auth.org/standards/22rm/html/RM-A-11.html>

⁸⁰ https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-string-operations

⁸¹ https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-string-operations-show-find-words

(continued from previous page)

```
27         (S'Length));
28
29     while I in S'Range loop
30         Find-Token
31         (Source => S,
32          Set    => Whitespace,
33          From   => I,
34          Test   => Outside,
35          First  => F,
36          Last   => L);
37
38         exit when L = 0;
39
40         Put_Line ("Found word instance at position "
41                 & F'Wide_Wide_Image
42                 & ": '" & S (F .. L) & "'");
43
44         I := L + 1;
45     end loop;
46
47 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 248).)

In the Ada Reference Manual

- [A.4.6 String-Handling Sets and Mappings](#)⁸²
- [A.4.7 Wide_String Handling](#)⁸³
- [A.4.8 Wide_Wide_String Handling](#)⁸⁴

⁸² <http://www.ada-auth.org/standards/22rm/html/RM-A-4-6.html>

⁸³ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-7.html>

⁸⁴ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-8.html>

6.1.3 Bounded and Unbounded Wide and Wide-Wide Strings

We've seen in the Introduction to Ada course that other kinds of **String** types are available. For example, we can use `bounded`⁸⁵ and `unbounded strings`⁸⁶ — those correspond to the `Bounded_String` and `Unbounded_String` types.

Those kinds of string types are available for **Wide_String**, and `Wide_Wide_String`. The following table shows the available types and corresponding packages:

Type	Package
<code>Bounded_Wide_String</code>	<code>Ada.Strings.Wide_Bounded</code>
<code>Bounded_Wide_Wide_String</code>	<code>Ada.Strings.Wide_Wide_Bounded</code>
<code>Unbounded_Wide_String</code>	<code>Ada.Strings.Wide_Unbounded</code>
<code>Unbounded_Wide_Wide_String</code>	<code>Ada.Strings.Wide_Wide_Unbounded</code>

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
<code>Bounded_Wide_String</code>	<code>Ada.Wide_Text_IO.Wide_Bounded_IO</code>
<code>Bounded_Wide_Wide_String</code>	<code>Ada.Wide_Wide_Text_IO.Wide_Wide_Bounded_IO</code>
<code>Unbounded_Wide_String</code>	<code>Ada.Wide_Text_IO.Wide_Unbounded_IO</code>
<code>Unbounded_Wide_Wide_String</code>	<code>Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO</code>

Let's look at a simple example:

Listing 4: `show_unbounded_wide_wide_string.adb`

```

1 with Ada.Strings.Wide_Wide_Unbounded;
2 use  Ada.Strings.Wide_Wide_Unbounded;
3
4 with Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO;
5 use  Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO;
6
7 procedure Show_Unbounded_Wide_Wide_String is
8   S : Unbounded_Wide_Wide_String
9     := To_Unbounded_Wide_Wide_String ("Hello");
10 begin
11   S := S & Wide_Wide_String'(" hello");
12   Put_Line ("Unbounded wide-wide string: " & S);
13 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

⁸⁵ https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-bounded-strings

⁸⁶ https://learn.adacore.com/courses/intro-to-ada/chapters/standard_library_strings.html#intro-ada-unbounded-strings

```
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](#)⁸⁷
 - [A.4.8 Wide_Wide_String Handling](#)⁸⁸
 - [A.11 Wide Text Input-Output and Wide Wide Text Input-Output](#)⁸⁹
-

6.2 String Encoding

Unicode is one of the most widespread standards for encoding writing systems other than the Latin alphabet. It defines a format called [Unicode Transformation Format \(UTF\)](#)⁹⁰ 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](#)⁹¹
-

6.2.1 UTF-8 encoding and decoding

A common UTF format is UTF-8, which encodes strings using up to four (8-bit) bytes and is backwards-compatible with the ASCII format. While encoding of ASCII characters requires only one byte, Chinese characters require three bytes, for example.

In Ada applications, UTF-8 strings are indicated by using the `UTF_8_String` from the `Ada.Strings.UTF_Encoding` package. In order to encode from and to UTF-8 strings, we can use the `Encode` and `Decode` functions. Those functions are specified in the child packages of the `Ada.Strings.UTF_Encoding` package. We select the appropriate child package depending on the string type we're using, as you can see in the following table:

Child Package of <code>Ada.Strings.UTF_Encoding</code>	Convert from / to
<code>.Strings</code>	String type
<code>.Wide_Strings</code>	Wide_String type
<code>.Wide_Wide_Strings</code>	<code>Wide_Wide_String</code> type

Let's look at an example:

⁸⁷ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-7.html>

⁸⁸ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-8.html>

⁸⁹ <http://www.ada-auth.org/standards/22rm/html/RM-A-11.html>

⁹⁰ https://unicode.org/faq/utf_bom.html#gen2

⁹¹ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-11.html>

Listing 5: show_ww_utf_string.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  with Ada.Strings.Wide_Wide_Unbounded;
10 use  Ada.Strings.Wide_Wide_Unbounded;
11
12 procedure Show_WW_UTF_String is
13
14     function To_UWWS
15         (Source : Wide_Wide_String)
16         return Unbounded_Wide_Wide_String
17         renames To_Unbounded_Wide_Wide_String;
18
19     function To_WWS
20         (Source : Unbounded_Wide_Wide_String)
21         return Wide_Wide_String
22         renames To_Wide_Wide_String;
23
24     Hello_World_Arabic : constant
25         UTF_8_String := "عالم يا مرحبا";
26     WWS_Hello_World_Arabic : constant
27         Wide_Wide_String :=
28         Decode (Hello_World_Arabic);
29
30     UWWS : Unbounded_Wide_Wide_String;
31 begin
32     UWWS := "Hello World: "
33         & To_UWWS (WWS_Hello_World_Arabic);
34
35     Show_WW_String : declare
36         WWS : constant Wide_Wide_String :=
37             To_WWS (UWWS);
38     begin
39         Put_Line ("Wide_Wide_String Length: "
40             & WWS'Length'Image);
41         Put_Line ("Wide_Wide_String Size: "
42             & WWS'Size'Image);
43     end Show_WW_String;
44
45     Put_Line
46     ("-----");
47     Put_Line
48     ("Converting Wide_Wide_String to UTF-8...");
49
50     Show_UTF_8_String : declare
51         S_UTF_8 : constant UTF_8_String :=
52             Encode (To_WWS (UWWS));
53     begin
54         Put_Line ("UTF-8 String: "
55             & S_UTF_8);
56         Put_Line ("UTF-8 String Length: "
57             & S_UTF_8'Length'Image);
58         Put_Line ("UTF-8 String Size: "
59             & S_UTF_8'Size'Image);
60     end Show_UTF_8_String;

```

(continues on next page)

(continued from previous page)

```
61  
62 end Show_WW_UTF_String;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WW_UTF_String  
MD5: cecfb420bb804f42e7a65b793abcbe5
```

Runtime output

```
Wide_Wide_String Length: 26  
Wide_Wide_String Size: 832  
-----  
Converting Wide_Wide_String to UTF-8...  
UTF-8 String: Hello World: عالم يا مرحبا  
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.

6.2.2 UTF-8 size and length

As you can see when running the last code example from the previous subsection, we have different sizes and lengths depending on the string type:

String type	Size	Length
<code>Wide_Wide_String</code>	832	26
<code>UTF_8_String</code>	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 6: show_utf_8.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  procedure Show_UTF_8 is
10
11     Symbol_UTF_8 : constant UTF_8_String := "x";
12     Symbol_WWS   : constant Wide_Wide_String :=
13                   Decode (Symbol_UTF_8);
14
15  begin
16     Put_Line ("Wide_Wide_String Length: "
17              & Symbol_WWS'Length'Image);
18     Put_Line ("Wide_Wide_String Size:   "
19              & Symbol_WWS'Size'Image);
20     Put_Line ("UTF-8 String Length:    "
21              & Symbol_UTF_8'Length'Image);
22     Put_Line ("UTF-8 String Size:      "
23              & Symbol_UTF_8'Size'Image);
24     New_Line;
25     Put_Line ("UTF-8 String:          "
26              & Symbol_UTF_8);
27  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](https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols_block)⁹², not the standard `"x"` from the [Basic Latin block](https://en.wikipedia.org/wiki/Basic_Latin_(Unicode_block))⁹³.)

⁹² https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols

⁹³ [https://en.wikipedia.org/wiki/Basic_Latin_\(Unicode_block\)](https://en.wikipedia.org/wiki/Basic_Latin_(Unicode_block))

6.2.3 UTF-8 encoding in source-code files

In the past, it was common to use different character sets in text files when writing in different (human) languages. By default, Ada source-code files are expected to use the Latin-1 coding, which is a 8-bit character set.

Nowadays, however, using UTF-8 coding for text files — including source-code files — is very common. If your Ada code only uses standard ASCII characters, but you're saving it in a UTF-8 coded file, there's no need to worry about character sets, as UTF-8 is backwards compatible with ASCII.

However, you might want to use Unicode symbols in your Ada source code to declare constants — as we did in the previous sections — and store the source code in a UTF-8 coded file. In this case, you need be careful about how this file is parsed by the compiler.

Let's look at this source-code example:

Listing 7: show_utf_8_strings.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.UTF_Encoding;
4 use Ada.Strings.UTF_Encoding;
5
6 procedure Show_UTF_8_Strings is
7
8     Symbols_UTF_8 : constant
9         UTF_8_String := "♥♪";
10
11 begin
12     Put_Line ("UTF_8_String: "
13             & Symbols_UTF_8);
14
15     Put_Line ("Length:      "
16             & Symbols_UTF_8'Length'Image);
17
18 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*⁹⁴ to be encoded in UTF-8 format.) Also, in this case, the UTF-8 encoding process is using two additional bytes. Therefore, the total length of the string is six, which matches what we see when running the `Show_UTF_8_Strings` procedure. In other words, the length of the `Symbols_UTF_8` string doesn't refer to those two characters ("♥♪") that we were using in the constant declaration, but the length of the encoded bytes in its UTF-8 representation.

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

The UTF-8 format is very useful for storing and transmitting texts. However, if we want to process Unicode symbols, it's probably better to use string types with 32-bit characters — such as `Wide_Wide_String`. For example, let's say we want to use the "♥♪" string again to initialize a constant of `Wide_Wide_String` type:

Listing 8: `show_wws_strings.adb`

```
1 with Ada.Text_IO;
2 with Ada.Wide_Wide_Text_IO;
3
4 procedure Show_WWS_Strings is
5
6     package TIO  renames Ada.Text_IO;
7     package WWTIO renames Ada.Wide_Wide_Text_IO;
8
9     Symbols_WWS : constant
10        Wide_Wide_String := "♥♪";
11
12 begin
13     WWTIO.Put_Line ("Wide_Wide_String: "
14                   & Symbols_WWS);
15
16     TIO.Put_Line ("Length:           "
17                 & Symbols_WWS'Length'Image);
18
19 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 `Wide_Wide_String`. For this case, GNAT offers the `-gnatw8` switch. Let's run the previous example using this switch:

Listing 9: `show_wws_strings.adb`

```
1 with Ada.Text_IO;
2 with Ada.Wide_Wide_Text_IO;
3
4 procedure Show_WWS_Strings is
5
6     package TIO  renames Ada.Text_IO;
7     package WWTIO renames Ada.Wide_Wide_Text_IO;
8
```

(continues on next page)

(continued from previous page)

```
9     Symbols_WWS : constant
10         Wide_Wide_String := "♥♪";
11
12 begin
13     WWTIO.Put_Line ("Wide_Wide_String: "
14                   & Symbols_WWS);
15
16     TIO.Put_Line ("Length:           "
17                 & Symbols_WWS'Length'Image);
18
19 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 10: show_wws_strings.adb

```
1 with Ada.Text_IO;
2 with Ada.Wide_Wide_Text_IO;
3
4 procedure Show_WWS_Strings is
5
6     package TIO   renames Ada.Text_IO;
7     package WWTIO renames Ada.Wide_Wide_Text_IO;
8
9     Symbols_WWS : constant
10         Wide_Wide_String := "♥♪";
11
12 begin
13     WWTIO.Put_Line ("Wide_Wide_String: "
14                   & Symbols_WWS);
15
16     TIO.Put_Line ("Length:           "
17                 & Symbols_WWS'Length'Image);
18
19 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 11: show_utf_8.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.UTF_Encoding;
4 use   Ada.Strings.UTF_Encoding;
5
6 procedure Show_UTF_8 is
7
8     Symbol_UTF_8 : constant
9         UTF_8_String :=
10             Character'Val (16#e2#)
11             & Character'Val (16#98#)
12             & Character'Val (16#85#);
13
14 begin
15     Put_Line ("UTF-8 String: "
16             & Symbol_UTF_8);
17 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.

6.2.4 UTF-16 encoding and decoding

So far, we've discussed the UTF-8 encoding scheme. However, other encoding schemes exist and are supported as well. In fact, the `Ada.Strings.UTF_Encoding` package defines three encoding schemes:

```
type Encoding_Scheme is (UTF_8,
                        UTF_16BE,
                        UTF_16LE);
```

For example, instead of using UTF-8 encoding, we can use UTF-16 encoding — either in the big-endian or in the little-endian version. To convert between UTF-8 and UTF-16 encoding schemes, we can make use of the conversion functions from the `Ada.Strings.UTF_Encoding.Conversions` package.

To declare a UTF-16 encoded string, we can use one of the following data types:

- the 8-bit-character based `UTF_String` type, or
- the 16-bit-character based `UTF_16_Wide_String` type.

When using the 8-bit version, though, we have to specify the input and output schemes when converting between UTF-8 and UTF-16 encoding schemes.

Let's see a code example that makes use of both `UTF_String` and `UTF_16_Wide_String` types:

Listing 12: `show_utf16_types.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.UTF_Encoding;
4 use Ada.Strings.UTF_Encoding;
5
6 with Ada.Strings.UTF_Encoding.Conversions;
7 use Ada.Strings.UTF_Encoding.Conversions;
8
9 procedure Show_UTF16_Types is
10   Symbols_UTF_8 : constant
11     UTF_8_String := "♥♪";
12
13   Symbols_UTF_16 : constant
14     UTF_16_Wide_String :=
15       Convert (Symbols_UTF_8);
16   -- ^ Calling Convert for UTF_8_String
17   --   to UTF_16_Wide_String conversion.
18
19   Symbols_UTF_16BE : constant
20     UTF_String :=
21       Convert (Item      => Symbols_UTF_8,
22               Input_Scheme => UTF_8,
23               Output_Scheme => UTF_16BE);
24   -- ^ Calling Convert for UTF_8_String
25   --   to UTF_String conversion in UTF-16BE
26   --   encoding.
27 begin
28   Put_Line ("UTF_8_String:      "
29             & Symbols_UTF_8);
30
31   Put_Line ("UTF_16_Wide_String:  "
32             & Convert (Symbols_UTF_16));
33   -- ^ Calling Convert for
34   --   the UTF_16_Wide_String to
```

(continues on next page)

(continued from previous page)

```

35  --          UTF_8_String conversion.
36
37  Put_Line
38  ("UTF_String / UTF_16BE: "
39   & Convert
40   (Item          => Symbols_UTF_16BE,
41    Input_Scheme => UTF_16BE,
42    Output_Scheme => UTF_8));
43 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 13: show_ww_utf16_string.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  with Ada.Strings.UTF_Encoding.Conversions;
10 use  Ada.Strings.UTF_Encoding.Conversions;
11
12 procedure Show_WW_UTF16_String is
13   Symbols_UTF_16 : constant
14     UTF_16_Wide_String :=
15     Wide_Character'Val (16#2665#) &
16     Wide_Character'Val (16#266B#);
17   -- ^ Calling Wide_Character'Val
18   -- to specify the UTF-16 BE code
19   -- for "♥" and "♪".
20
21   Symbols_WWS : constant
22     Wide_Wide_String :=

```

(continues on next page)

(continued from previous page)

```
23     Decode (Symbols_UTF_16);
24     -- ^ Calling Decode for UTF_16_Wide_String
25     --   to Wide_Wide_String conversion.
26 begin
27     Put_Line ("UTF_16_Wide_String: "
28             & Convert (Symbols_UTF_16));
29     --     ^ Calling Convert for the
30     --       UTF_16_Wide_String to
31     --       UTF_8_String conversion.
32
33     Put_Line ("Wide_Wide_String:  "
34             & Encode (Symbols_WWS));
35     --     ^ Calling Encode for the
36     --       Wide_Wide_String to
37     --       UTF_8_String conversion.
38 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.

6.3 Image attribute

6.3.1 Overview

In the [Introduction to Ada](#)⁹⁵ 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 14: show_simple_image.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Image is
4     V : Integer;
5 begin
6     V := 10;
7     Put_Line ("V: " & Integer'Image (V));
8 end Show_Simple_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Simple_Image
MD5: e38f6f1a0808f12bd53c1f3cf4983353

⁹⁵ https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-image-attribute

Runtime output

```
V: 10
```

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

Listing 15: show_simple_image.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Image is
4   type Status is (Unknown, Off, On);
5
6   V : Float;
7   S : Status;
8 begin
9   V := 10.0;
10  S := Unknown;
11
12  Put_Line ("V: " & Float'Image (V));
13  Put_Line ("S: " & Status'Image (S));
14 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.

In the Ada Reference Manual

- [Image Attributes](#)⁹⁶
-

6.3.2 Type'Image and Obj'Image

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

Listing 16: show_simple_image.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Image is
4   V : Integer;
5 begin
6   V := 10;
7   Put_Line ("V: " & V'Image);
8
9   -- Equivalent to:
10  -- Put_Line ("V: " & Integer'Image (V));
11 end Show_Simple_Image;
```

⁹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-4-10.html>

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.

6.3.3 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
<code>Image</code>	<code>String</code>
<code>Wide_Image</code>	<code>Wide_String</code>
<code>Wide_Wide_Image</code>	<code>Wide_Wide_String</code>

Let's see a simple example:

Listing 17: `show_wide_wide_image.adb`

```
1 with Ada.Wide_Wide_Text_IO;  
2 use Ada.Wide_Wide_Text_IO;  
3  
4 procedure Show_Wide_Wide_Image is  
5   F : Float;  
6 begin  
7   F := 100.0;  
8   Put_Line ("F = "  
9             & F'Wide_Wide_Image);  
10 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`.

6.3.4 Image attribute for non-scalar types

Note: This feature was introduced in Ada 2022.

In the previous code examples, we were using the Image attribute with scalar types, but it isn't restricted to those types. In fact, we can also use this attribute when dealing with non-scalar types. For example:

Listing 18: simple_records.ads

```

1 package Simple_Records is
2
3     type Rec is limited private;
4
5     type Rec_Access is access Rec;
6
7     function Init return Rec;
8
9     type Null_Rec is null record;
10
11 private
12
13     type Rec is limited record
14         F : Float;
15         I : Integer;
16     end record;
17
18     function Init return Rec is
19         ((F => 10.0, I => 4));
20
21 end Simple_Records;
```

Listing 19: show_non_scalar_image.adb

```

1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4 with Ada.Unchecked_Deallocation;
5
6 with Simple_Records;
7 use Simple_Records;
8
9 procedure Show_Non_Scalar_Image is
10
11     procedure Free is
12         new Ada.Unchecked_Deallocation
13             (Object => Rec,
14              Name   => Rec_Access);
15
16     R_A : Rec_Access :=
17         new Rec'(Init);
18
19     N_R : Null_Rec :=
20         (null record);
21 begin
22     R_A := new Rec'(Init);
23     N_R := (null record);
24
25     Put_Line ("R_A:      " & R_A'Image);
26     Put_Line ("R_A.all: " & R_A.all'Image);
```

(continues on next page)

(continued from previous page)

```
27   Put_Line ("N_R:      " & N_R'Image);
28
29   Free (R_A);
30   Put_Line ("R_A:      " & R_A'Image);
31 end Show_Non_Scalar_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Non_Scalar_Image
MD5: d7d15e96a03c882995262a5cfca5e771
```

Runtime output

```
R_A:      (access ea02c0)
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 Types⁹⁷](#) section of the Ada Reference Manual, as it was only applied to those types. Now, it is part of the new [Image Attributes⁹⁸](#) section.

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

Listing 20: `show_array_image.adb`

```
1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Array_Image is
6
7      type Float_Array is
8          array (Positive range <>) of Float;
9
10     FA_3C   : Float_Array (1 .. 3);
11     FA_Null : Float_Array (1 .. 0);
12
13 begin
14     FA_3C   := [1.0, 3.0, 2.0];
15     FA_Null := [];
16
17     Put_Line ("FA_3C:  " & FA_3C'Image);
18     Put_Line ("FA_Null: " & FA_Null'Image);
19 end Show_Array_Image;
```

⁹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-5.html>

⁹⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-10.html>

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Array_Image
 MD5: 2d3fcdd5e57451f08185618d357b705f

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.

6.3.5 Image attribute for tagged types

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

Listing 21: simple_records.ads

```
1 package Simple_Records is
2
3     type Rec is tagged limited private;
4
5     function Init return Rec;
6
7     type Rec_Child is new Rec with private;
8
9     overriding function Init return Rec_Child;
10
11 private
12
13     type Status is (Unknown, Off, On);
14
15     type Rec is tagged limited record
16         F : Float;
17         I : Integer;
18     end record;
19
20     function Init return Rec is
21         ((F => 10.0, I => 4));
22
23     type Rec_Child is new Rec with record
24         Z : Status;
25     end record;
26
27     function Init return Rec_Child is
28         (Rec'(Init) with Z => Off);
29
30 end Simple_Records;
```

Listing 22: show_tagged_image.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO;    use Ada.Text_IO;
4
5 with Simple_Records; use Simple_Records;
```

(continues on next page)

(continued from previous page)

```
6
7 procedure Show_Tagged_Image is
8   R      : constant Rec      := Init;
9   R_Class : constant Rec'Class := Rec'(Init);
10  R_C     : constant Rec_Child := Init;
11 begin
12   Put_Line ("R:      " & R'Image);
13   Put_Line ("R_Class: " & R_Class'Image);
14   Put_Line ("R_A:     " & R_C'Image);
15 end Show_Tagged_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Tagged_Image
MD5: 164bd17c99115acafb09c99f40c1578c

Runtime output

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

In the `Show_Tagged_Image` procedure from this example, we display the contents of the `R` object of `Rec` type and the `R_Class` object of `Rec'Class` type. Also, we display the contents of the `R_C` object of the `Rec_Child` type, which is derived from the `Rec` type.

6.3.6 Image attribute for task and protected types

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

Listing 23: simple_tasking.ads

```
1 package Simple_Tasking is
2
3   protected type Protected_Float (I : Integer) is
4
5   private
6     V : Float := Float (I);
7   end Protected_Float;
8
9   protected type Protected_Null is
10  private
11  end Protected_Null;
12
13  task type T is
14    entry Start;
15  end T;
16
17 end Simple_Tasking;
```

Listing 24: simple_tasking.adb

```
1 package body Simple_Tasking is
2
3   protected body Protected_Float is
4
5   end Protected_Float;
6
7   protected body Protected_Null is
```

(continues on next page)

(continued from previous page)

```

8
9   end Protected_Null;
10
11  task body T is
12  begin
13    accept Start;
14  end T;
15
16 end Simple_Tasking;

```

Listing 25: show_protected_task_image.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO;    use Ada.Text_IO;
4
5  with Simple_Tasking; use Simple_Tasking;
6
7  procedure Show_Protected_Task_Image is
8
9    PF : Protected_Float (0);
10   PN : Protected_Null;
11   T1 : T;
12
13  begin
14    Put_Line ("PF: " & PF'Image);
15    Put_Line ("PN: " & PN'Image);
16    Put_Line ("T1: " & T1'Image);
17
18    T1.Start;
19  end Show_Protected_Task_Image;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Protected_Task_↵Image
 MD5: 9d8c667015878eb14e5b3950a70b86b1

Runtime output

```

PF: (protected object)
PN: (protected object)
T1: (task t1_0000000000E94090)

```

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

6.4 Put_Image aspect

Note: This feature was introduced in Ada 2022.

6.4.1 Overview

In the previous section, we discussed many details about the Image attribute. In the code examples from that section, we've seen the default behavior of this attribute: the string returned by the calls to Image was always in the format defined by the Ada standard.

In some situations, however, we might want to customize the string that is returned by the Image attribute of a type T. Ada allows us to do that via the Put_Image aspect. This is what we have to do:

1. Specify the Put_Image aspect for the type T and indicate a procedure with a specific parameter profile — let's say, for example, a procedure named P.
2. Implement the procedure P and write the information we want to use into a buffer (by calling the routines defined for Root_Buffer_Type, such as the Put procedure).

We can see these steps performed in the code example below:

Listing 26: show_put_image.ads

```
1 pragma Ada_2022;
2
3 with Ada.Strings.Text_Buffers;
4
5 package Show_Put_Image is
6
7     type T is null record
8         with Put_Image => Put_Image_T;
9         -- ^ Custom version of Put_Image
10
11     use Ada.Strings.Text_Buffers;
12
13     procedure Put_Image_T
14         (Buffer : in out Root_Buffer_Type'Class;
15          Arg    : T);
16
17 end Show_Put_Image;
```

Listing 27: show_put_image.adb

```
1 package body Show_Put_Image is
2
3     procedure Put_Image_T
4         (Buffer : in out Root_Buffer_Type'Class;
5          Arg    : T) is
6         pragma Unreferenced (Arg);
7     begin
8         -- Call Put with customized
9         -- information
10        Buffer.Put ("<custom info>");
11    end Put_Image_T;
12
13 end Show_Put_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Simple_Put_Image
MD5: cbdd77a9e6cc30f3604c0901536d87aa
```

In the Show_Put_Image package, we use the Put_Image aspect in the declaration of the T type. There, we indicate that the Image attribute shall use the Put_Image_T procedure instead of the default version.

In the body of the `Put_Image_T` procedure, we implement our custom version of the `Image` attribute. We do that by calling the `Put` procedure with the information we want to provide in the `Image` attribute. Here, we access a buffer of `Root_Buffer_Type` type, which is defined in the `Ada.Strings.Text_Buffers` package. (We discuss more about this package *later on* (page 261).)

In the Ada Reference Manual

- [Image Attributes](#)⁹⁹

6.4.2 Complete Example of `Put_Image`

Let's see a complete example in which we use the `Put_Image` aspect and write useful information to the buffer:

Listing 28: `custom_numerics.ads`

```

1  pragma Ada_2022;
2
3  with Ada.Strings.Text_Buffers;
4
5  package Custom_Numerics is
6
7      type Float_Integer is record
8          F : Float := 0.0;
9          I : Integer := 0;
10     end record
11     with Dynamic_Predicate =>
12         Integer (Float_Integer.F) =
13             Float_Integer.I,
14         Put_Image => Put_Float_Integer;
15     -- ^ Custom version of Put_Image
16
17     use Ada.Strings.Text_Buffers;
18
19     procedure Put_Float_Integer
20         (Buffer : in out Root_Buffer_Type'Class;
21          Arg    : Float_Integer);
22
23 end Custom_Numerics;
```

Listing 29: `custom_numerics.adb`

```

1  package body Custom_Numerics is
2
3      procedure Put_Float_Integer
4          (Buffer : in out Root_Buffer_Type'Class;
5           Arg    : Float_Integer) is
6      begin
7          -- Call Wide_Wide_Put with customized
8          -- information
9          Buffer.Wide_Wide_Put
10             ("(F : " & Arg.F'Wide_Wide_Image & ", "
11              & "I : " & Arg.I'Wide_Wide_Image & ")");
12     end Put_Float_Integer;
13
14 end Custom_Numerics;
```

⁹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-4-10.html>

Listing 30: show_put_image.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO;      use Ada.Text_IO;
4
5 with Custom_Numerics; use Custom_Numerics;
6
7 procedure Show_Put_Image is
8   V : Float_Integer;
9 begin
10  V := (F => 100.2,
11        I => 100);
12  Put_Line ("V = "
13           & V'Image);
14 end Show_Put_Image;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Put_Image_Custom_
↳Numerics
MD5: 18d31150d7a9ff9af0359495543c011f
```

Runtime output

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

In the `Custom_Numerics` package of this example, we specify the `Put_Image` aspect and indicate the `Put_Float_Integer` procedure. In that procedure, we display the information of components `F` and `I`. Then, in the `Show_Put_Image` procedure, we use the `Image` attribute for the `V` variable and see the information in the exact format we specified. (If you like to see the default version of the `Put_Image` instead, you may comment out the `Put_Image` aspect part in the declaration of `Float_Integer`.)

6.4.3 Relation to the Image attribute

Note that we cannot override the `Image` attribute directly — there's no *Image aspect* that we could specify. However, as we've just seen, we can do this indirectly by using our own version of the `Put_Image` procedure for a type `T`.

The `Image` attribute of a type `T` makes use of the procedure indicated in the `Put_Image` aspect. Let's say we have the following declaration:

```
type T is null record
  with Put_Image => Put_Image_T;
```

When we then use the `T'Image` attribute in our code, the custom `Put_Image_T` procedure is automatically called. This is a simplified example of how the `Image` function is implemented:

```
function Image (V : T)
  return String is
  Buffer : Custom_Buffer;
  --   ^ of Root_Buffer_Type'Class
begin
  -- Calling Put_Image procedure
  -- for type T
  Put_Image_T (Buffer, V);

  -- Retrieving the text from the
```

(continues on next page)

(continued from previous page)

```
-- 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 246) about the Image attribute.)

6.4.4 Put_Image and derived types

Types that were derived from untagged types (or null extensions) make use of the Put_Image procedure that was specified for their parent type — either a custom procedure indicated in the Put_Image aspect or the default one. Naturally, if a derived type has the Put_Image aspect, the procedure indicated in the aspect is used instead. For example:

Listing 31: untagged_put_image.ads

```
1 pragma Ada_2022;
2
3 with Ada.Strings.Text_Buffers;
4
5 package Untagged_Put_Image is
6
7     use Ada.Strings.Text_Buffers;
8
9     type T is null record
10         with Put_Image => Put_Image_T;
11
12     procedure Put_Image_T
13         (Buffer : in out Root_Buffer_Type'Class;
14          Arg    : T);
15
16     type T_Derived_1 is new T;
17
18     type T_Derived_2 is new T
19         with Put_Image => Put_Image_T_Derived_2;
20
21     procedure Put_Image_T_Derived_2
22         (Buffer : in out Root_Buffer_Type'Class;
23          Arg    : T_Derived_2);
24
25 end Untagged_Put_Image;
```

Listing 32: untagged_put_image.adb

```
1 package body Untagged_Put_Image is
2
3     procedure Put_Image_T
4         (Buffer : in out Root_Buffer_Type'Class;
5          Arg    : T) is
6         pragma Unreferenced (Arg);
```

(continues on next page)

(continued from previous page)

```

7  begin
8      Buffer.Wide_Wide_Put ("Put_Image_T");
9  end Put_Image_T;
10
11  procedure Put_Image_T_Derived_2
12      (Buffer : in out Root_Buffer_Type'Class;
13       Arg    : T_Derived_2) is
14      pragma Unreferenced (Arg);
15  begin
16      Buffer.Wide_Wide_Put
17          ("Put_Image_T_Derived_2");
18  end Put_Image_T_Derived_2;
19
20 end Untagged_Put_Image;

```

Listing 33: show_untagged_put_image.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO;          use Ada.Text_IO;
4
5  with Untagged_Put_Image; use Untagged_Put_Image;
6
7  procedure Show_Untagged_Put_Image is
8      Obj_T           : T;
9      Obj_T_Derived_1 : T_Derived_1;
10     Obj_T_Derived_2 : T_Derived_2;
11  begin
12     Put_Line ("T'Image : "
13             & Obj_T'Image);
14     Put_Line ("T_Derived_1'Image : "
15             & Obj_T_Derived_1'Image);
16     Put_Line ("T_Derived_2'Image : "
17             & Obj_T_Derived_2'Image);
18  end Show_Untagged_Put_Image;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Untagged_Put_Image
MD5: b0a115967ec5f2deaea19967d22266b4

Runtime output

```

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

```

In this example, we declare the type `T` and its derived types `T_Derived_1` and `T_Derived_2`. When running this code, we see that:

- `T_Derived_1` makes use of the `Put_Image_T` procedure from its parent.
 - Note that, if we remove the `Put_Image` aspect from the declaration of `T`, the default version of the `Put_Image` procedure is used for both `T` and `T_Derived_1` types.
- `T_Derived_2` makes use of the `Put_Image_T_Derived_2` procedure, which was indicated in the `Put_Image` aspect of that type, instead of its parent's procedure.

6.4.5 Put_Image and tagged types

Types that are derived from a tagged type may also inherit the Put_Image aspect. However, there are a couple of small differences in comparison to untagged types, as we can see in the following example:

Listing 34: tagged_put_image.ads

```

1  pragma Ada_2022;
2
3  with Ada.Strings.Text_Buffers;
4
5  package Tagged_Put_Image is
6
7      use Ada.Strings.Text_Buffers;
8
9      type T is tagged record
10         I : Integer := 0;
11     end record
12         with Put_Image => Put_Image_T;
13
14     procedure Put_Image_T
15         (Buffer : in out Root_Buffer_Type'Class;
16          Arg    : T);
17
18     type T_Child_1 is new T with record
19         I1 : Integer;
20     end record;
21
22     type T_Child_2 is new T with null record;
23
24     type T_Child_3 is new T with record
25         I3 : Integer := 0;
26     end record
27         with Put_Image => Put_Image_T_Child_3;
28
29     procedure Put_Image_T_Child_3
30         (Buffer : in out Root_Buffer_Type'Class;
31          Arg    : T_Child_3);
32
33 end Tagged_Put_Image;
```

Listing 35: tagged_put_image.adb

```

1  package body Tagged_Put_Image is
2
3      procedure Put_Image_T
4         (Buffer : in out Root_Buffer_Type'Class;
5          Arg    : T) is
6          pragma Unreferenced (Arg);
7      begin
8          Buffer.Wide_Wide_Put ("Put_Image_T");
9      end Put_Image_T;
10
11     procedure Put_Image_T_Child_3
12         (Buffer : in out Root_Buffer_Type'Class;
13          Arg    : T_Child_3) is
14         pragma Unreferenced (Arg);
15     begin
16         Buffer.Wide_Wide_Put
17             ("Put_Image_T_Child_3");
18     end Put_Image_T_Child_3;
```

(continues on next page)

(continued from previous page)

```
19
20 end Tagged_Put_Image;
```

Listing 36: show_tagged_put_image.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO;      use Ada.Text_IO;
4
5 with Tagged_Put_Image; use Tagged_Put_Image;
6
7 procedure Show_Tagged_Put_Image is
8   Obj_T      : T;
9   Obj_T_Child_1 : T_Child_1;
10  Obj_T_Child_2 : T_Child_2;
11  Obj_T_Child_3 : T_Child_3;
12 begin
13   Put_Line ("T'Image :      "
14            & Obj_T'Image);
15   Put_Line ("-----");
16   Put_Line ("T_Child_1'Image : "
17            & Obj_T_Child_1'Image);
18   Put_Line ("-----");
19   Put_Line ("T_Child_2'Image : "
20            & Obj_T_Child_2'Image);
21   Put_Line ("-----");
22   Put_Line ("T_Child_3'Image : "
23            & Obj_T_Child_3'Image);
24   Put_Line ("-----");
25   Put_Line ("T'Class'Image :    "
26            & T'Class (Obj_T_Child_1)'Image);
27 end Show_Tagged_Put_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Tagged_Put_Image
MD5: 74d29ea54f1ad79fea7de2ad7c1dcb31

Runtime output

```
T'Image :      Put_Image_T
-----
T_Child_1'Image :
(Put_Image_T with I1 => 1701066593)
-----
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 => 1701066593)
```

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

- for both `T_Child_1` and `T_Child_2`, the parent's `Put_Image` aspect (the `Put_Image_T` procedure) is called and its information is combined with the information from the type extension;
 - The information from the parent's `Put_Image_T` procedure is presented in an aggregate syntax — in this case, this results in `(Put_Image_T)`.

- For the `T_Child_1` type, the `I1` component of the type extension is displayed by calling a default version of the `Put_Image` procedure for that component — (`Put_Image_T` with `I1 => 0`) is displayed.
- For the `T_Child_2` type, no additional information is displayed because this type has a null extension.
- for the `T_Child_3` type, the `Put_Image_T_Child_3` procedure, which was indicated in the `Put_Image` aspect of the type, is used.

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

6.5 Universal text buffer

In the *previous section* (page 253), we've seen that the first parameter of the procedure indicated in the `Put_Image` aspect has the `Root_Buffer_Type'Class` type, which is defined in the `Ada.Strings.Text_Buffers` package. In this section, we talk more about this type and additional procedures associated with this type.

Note: This feature was introduced in Ada 2022.

6.5.1 Overview

We use the `Root_Buffer_Type'Class` type to implement a universal text buffer that is used to store and retrieve information about data types. Because this text buffer isn't associated with specific data types, it is universal — in the sense that we can really use it for any data type, regardless of the characteristics of this type.

In theory, we could use Ada's universal text buffer to implement applications that actually process text in some form — for example, when implementing a text editor. However, in general, Ada programmers are only expected to make use of the `Root_Buffer_Type'Class` type when implementing a procedure for the `Put_Image` aspect. For this reason, we won't discuss any kind of type derivation — or any other kind of usages of this type — in this section. Instead, we'll just focus on additional subprograms from the `Ada.Strings.Text_Buffers` package.

In the Ada Reference Manual

- [Universal Text Buffers](#)¹⁰⁰

¹⁰⁰ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-12.html>

6.5.2 Additional procedures

In the previous section, we used the `Put` procedure — and the related `Wide_Put` and `Wide_Wide_Put` procedures — from the `Ada.Strings.Text_Buffers` package. In addition to these procedures, the package also includes:

- the `New_Line` procedure, which writes a new line marker to the text buffer;
- the `Increase_Indent` procedure, which increases the indentation in the text buffer; and
- the `Decrease_Indent` procedure, which decreases the indentation in the text buffer.

The `Ada.Strings.Text_Buffers` package also includes the `Current_Indent` function, which retrieves the current indentation counter.

Let's revisit an example from the previous section and use the procedures mentioned above:

Listing 37: `custom_numerics.ads`

```
1 pragma Ada_2022;
2
3 with Ada.Strings.Text_Buffers;
4
5 package Custom_Numerics is
6
7     type Float_Integer is record
8         F : Float;
9         I : Integer;
10    end record
11    with Dynamic_Predicate =>
12         Integer (Float_Integer.F) =
13             Float_Integer.I,
14         Put_Image      => Put_Float_Integer;
15    -- ^ Custom version of Put_Image
16
17    use Ada.Strings.Text_Buffers;
18
19    procedure Put_Float_Integer
20        (Buffer : in out Root_Buffer_Type'Class;
21         Arg    : Float_Integer);
22
23 end Custom_Numerics;
```

Listing 38: `custom_numerics.adb`

```
1 package body Custom_Numerics is
2
3     procedure Put_Float_Integer
4         (Buffer : in out Root_Buffer_Type'Class;
5          Arg    : Float_Integer) is
6     begin
7         Buffer.Wide_Wide_Put ("(");
8         Buffer.New_Line;
9
10        Buffer.Increase_Indent;
11
12        Buffer.Wide_Wide_Put
13            ("F : "
14             & Arg.F'Wide_Wide_Image);
15        Buffer.New_Line;
16
17        Buffer.Wide_Wide_Put
```

(continues on next page)

(continued from previous page)

```

18     ("I : "
19     & Arg.I'Wide_Wide_Image);
20
21     Buffer.Decrease_Indent;
22     Buffer.New_Line;
23
24     Buffer.Wide_Wide_Put ("");
25 end Put_Float_Integer;
26
27 end Custom_Numerics;

```

Listing 39: show_put_image.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO;    use Ada.Text_IO;
4
5  with Custom_Numerics; use Custom_Numerics;
6
7  procedure Show_Put_Image is
8     V : Float_Integer;
9  begin
10     V := (F => 100.2,
11           I => 100);
12     Put_Line ("V = "
13              & V'Image);
14 end Show_Put_Image;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Universal_Text_Buffer.Put_Image_
↳ Custom_Numerics
MD5: af95f9fe4064e8a9d7aebe14d7f561f7

Runtime output

```

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

```

In the body of the `Put_Float_Integer` procedure, we're using the `New_Line`, `Increase_Indent` and `Decrease_Indent` procedures to improve the format of the string returned by the `Float_Integer`' `Image` attribute. Using these procedures, you can create any kind of output format for your custom type.

NUMERICS

7.1 Modular Types

In the Introduction to Ada course, we've seen that Ada has two kinds of integer type: `signed`¹⁰¹ and `modular`¹⁰² types. For example:

Listing 1: num_types.ads

```
1 package Num_Types is
2
3     type Signed_Integer is range 1 .. 1_000_000;
4     type Modular is mod 2**32;
5
6 end Num_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_1
MD5: 2dff9fe22c6bbe52f964befccf68deb

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](#)¹⁰³
-

7.1.1 Modulus Attribute

The `Modulus` attribute returns the modulus of the modular type as a universal integer value. Let's get the modulus of the 32-bit `Modular` type that we've declared in the `Num_Types` package of the previous example:

Listing 2: show_modular.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Num_Types;   use Num_Types;
4
5 procedure Show_Modular is
```

(continues on next page)

¹⁰¹ https://learn.adacore.com/courses/intro-to-ada/chapters/strongly_typed_language.html#intro-ada-integers

¹⁰² https://learn.adacore.com/courses/intro-to-ada/chapters/strongly_typed_language.html#intro-ada-unsigned-types

¹⁰³ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-4.html>

(continued from previous page)

```
6   Modulus_Value : constant := Modular'Modulus;
7   begin
8     Put_Line (Modulus_Value'Image);
9   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} .

7.1.2 Mod Attribute

Note: This section was originally written by Robert A. Duff and published as [Gem #26: The Mod Attribute](#)¹⁰⁴.

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 11).)

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

Listing 3: show_modular.adb

```
1   with Ada.Text_IO; use Ada.Text_IO;
2
3   with Num_Types;   use Num_Types;
4
5   procedure Show_Modular is
6     X : Modular;
7   begin
8     X := 1;
9     Put_Line (X'Image);
10
11    X := -X;
12    Put_Line (X'Image);
13  end Show_Modular;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_1
MD5: e9ac61d2e43585f002fe2b79544ef9d7

Runtime output

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

```
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 4: show_modular.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Num_Types; use Num_Types;
4
5 procedure Show_Modular is
6   I : Integer := -1;
7   X : Modular := 1;
8 begin
9   X := Modular (I); -- raises Constraint_Error
10  Put_Line (X'Image);
11 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 5: show_modular.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Num_Types; use Num_Types;
4
5 procedure Show_Modular is
6   I : constant Integer := -1;
7   X : Modular := 1;
8 begin
9   X := Modular'Mod (I);
10  Put_Line (X'Image);
11 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 6: mod_attribute.ads

```
1 generic
2   type Formal_Modular is mod <>;
3 package Mod_Attribute is
4   function F return Formal_Modular;
5 end Mod_Attribute;
```

Listing 7: mod_attribute.adb

```
1 package body Mod_Attribute is
2
3   A_Signed_Integer : Integer := -1;
4
5   function F return Formal_Modular is
6   begin
7     return Formal_Modular'Mod
8       (A_Signed_Integer);
9   end F;
10
11 end Mod_Attribute;
```

Code block metadata

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

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

7.1.3 Operations on modular types

Modular types are particularly useful for bit manipulation. For example, we can use the **and**, **or**, **xor** and **not** operators for modular types.

Also, we can perform bit-shifting by multiplying or dividing a modular object with a power of two. For example, if `M` is a variable of modular type, then `M := M * 2 ** 3`; shifts the bits to the left by three bits. Likewise, `M := M / 2 ** 3` shifts the bits to the right. Note that the compiler selects the appropriate shifting operator when translating these operations to machine code — no actual multiplication or division will be performed.

Let's see a simple implementation of the CRC-CCITT (0x1D0F) algorithm:

Listing 8: crc_defs.ads

```

1 package Crc_Defs is
2
3   type Byte is mod 2 ** 8;
4   type Crc is mod 2 ** 16;
5
6   type Byte_Array is
7     array (Positive range <>) of Byte;
8
9   function Crc_CCITT (A : Byte_Array)
10     return Crc;
11
12   procedure Display (Crc_A : Crc);
13
14   procedure Display (A : Byte_Array);
15
16 end Crc_Defs;
```

Listing 9: crc_defs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Crc_Defs is
4
5   package Byte_IO is new Modular_IO (Byte);
6   package Crc_IO is new Modular_IO (Crc);
7
8   function Crc_CCITT (A : Byte_Array)
9     return Crc
10  is
11    X      : Byte;
12    Crc_A : Crc := 16#1d0f#;
13  begin
14    for I in A'Range loop
15      X := Byte (Crc_A / 2 ** 8) xor A (I);
16      X := X xor (X / 2 ** 4);
17      declare
18        Crc_X : constant Crc := Crc (X);
19      begin
20        Crc_A := Crc_A * 2 ** 8 xor
21          Crc_X * 2 ** 12 xor
22          Crc_X * 2 ** 5 xor
23          Crc_X;
24      end;
25    end loop;
26
27    return Crc_A;
28  end Crc_CCITT;
29
30  procedure Display (Crc_A : Crc) is
31  begin
32    Crc_IO.Put (Crc_A);
33    New_Line;
34  end Display;
35
36  procedure Display (A : Byte_Array) is
37  begin
38    for E of A loop
39      Byte_IO.Put (E);
40      Put (" ", " ");
```

(continues on next page)

(continued from previous page)

```
41     end loop;
42     New_Line;
43     end Display;
44
45 begin
46     Byte_IO.Default_Width := 1;
47     Byte_IO.Default_Base := 16;
48     Crc_IO.Default_Width := 1;
49     Crc_IO.Default_Base := 16;
50 end Crc_Defs;
```

Listing 10: show_crc.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Crc_Defs;    use Crc_Defs;
3
4 procedure Show_Crc is
5     AA : constant Byte_Array :=
6         (16#0#, 16#20#, 16#30#);
7     Crc_A : Crc;
8 begin
9     Crc_A := Crc_CCITT (AA);
10
11     Put ("Input array: ");
12     Display (AA);
13
14     Put ("CRC-CCITT: ");
15     Display (Crc_A);
16 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.

7.2 Numeric Literals

7.2.1 Classification

We've already discussed basic characteristics of numeric literals in the Introduction to Ada course — although we haven't used this terminology there. There are two kinds of numeric literals in Ada: integer literals and real literals. They are distinguished by the absence or presence of a radix point. For example:

Listing 11: real_integer_literals.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
```

(continues on next page)

(continued from previous page)

```

3 procedure Real_Integer_Literals is
4   Integer_Literal : constant := 365;
5   Real_Literal    : constant := 365.2564;
6 begin
7   Put_Line ("Integer Literal: "
8             & Integer_Literal'Image);
9   Put_Line ("Real Literal: "
10            & Real_Literal'Image);
11 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:    3.652564000000000000E+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 12: decimal_based_literals.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Decimal_Based_Literals is
4
5   package F_IO is new
6     Ada.Text_IO.Float_IO (Float);
7
8   --
9   --  DECIMAL LITERALS
10  --
11
12  Dec_Integer : constant := 365;
13
14  Dec_Real    : constant := 365.2564;
15  Dec_Real_Exp : constant := 0.365_256_4e3;
16
17  --
18  --  BASED LITERALS
19  --
20
21  Based_Integer    : constant := 16#16D#;
22  Based_Integer_Exp : constant := 5#243#e1;
23
24  Based_Real    : constant :=
25    2#1_0110_1101.0100_0001_1010_0011_0111#;
26  Based_Real_Exp : constant :=
27    7#1.031_153_643#e3;
28 begin
29   F_IO.Default_Fore := 3;
30   F_IO.Default_Aft  := 4;
31   F_IO.Default_Exp  := 0;
32
33   Put_Line ("Dec_Integer: "
34            & Dec_Integer'Image);

```

(continues on next page)

(continued from previous page)

```
35
36   Put ("Dec_Real:           ");
37   F_IO.Put (Item => Dec_Real);
38   New_Line;
39
40   Put ("Dec_Real_Exp:       ");
41   F_IO.Put (Item => Dec_Real_Exp);
42   New_Line;
43
44   Put_Line ("Based_Integer:   "
45            & Based_Integer'Image);
46   Put_Line ("Based_Integer_Exp: "
47            & Based_Integer_Exp'Image);
48
49   Put ("Based_Real:         ");
50   F_IO.Put (Item => Based_Real);
51   New_Line;
52
53   Put ("Based_Real_Exp:     ");
54   F_IO.Put (Item => Based_Real_Exp);
55   New_Line;
56 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
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.

7.2.2 Features and Flexibility

Note: This section was originally written by Franco Gasperoni and published as [Gem #7: The Beauty of Numeric Literals in Ada](#)¹⁰⁵.

Ada provides a simple and elegant way of expressing numeric literals. One of those simple, yet powerful aspects is the ability to use underscores to separate groups of digits. For example, `3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510` is more readable and less error prone to type than `3.14159265358979323846264338327950288419716939937510`. Here's the complete code:

¹⁰⁵ <https://www.adacore.com/gems/ada-gem-7>

Listing 13: ada_numeric_literals.adb

```

1  with Ada.Text_IO;
2
3  procedure Ada_Numeric_Literals is
4      Pi    : constant :=
5          3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510;
6
7      Pi2   : constant :=
8          3.14159265358979323846264338327950288419716939937510;
9
10     Z     : constant := Pi - Pi2;
11     pragma Assert (Z = 0.0);
12
13     use Ada.Text_IO;
14     begin
15         Put_Line ("Z = " & Float'Image (Z));
16     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:

Listing 14: ada_numeric_literals.adb

```

1  with Ada.Text_IO;
2
3  procedure Ada_Numeric_Literals is
4      Bin_136 : constant := 2#1000_1000#;
5      Oct_136 : constant := 8#210#;
6      Dec_136 : constant := 10#136#;
7      Hex_136 : constant := 16#88#;
8      pragma Assert (Bin_136 = 136);
9      pragma Assert (Oct_136 = 136);
10     pragma Assert (Dec_136 = 136);
11     pragma Assert (Hex_136 = 136);
12
13     use Ada.Text_IO;
14
15     begin
16         Put_Line ("Bin_136 = "
17             & Integer'Image (Bin_136));
18         Put_Line ("Oct_136 = "
19             & Integer'Image (Oct_136));
20         Put_Line ("Dec_136 = "
21             & Integer'Image (Dec_136));
22         Put_Line ("Hex_136 = "
23             & Integer'Image (Hex_136));
24     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 15: `ada_numeric_literals.adb`

```
1 with Ada.Text_IO;
2
3 procedure Ada_Numeric_Literals is
4   Lights_On  : constant := 2#1000_1000#;
5   Lights_Off : constant := 2#0111_0111#;
6
7   type Byte is mod 256;
8   Output_Devices : Byte := 0;
9
10  -- for Output_Devices'Address
11  --   use 16#DEAD_BEEF#;
12  -- ~~~~~
13  -- Memory mapped Output
14
15  use Ada.Text_IO;
16 begin
17   Output_Devices := Output_Devices or
18                     Lights_On;
19
20   Put_Line ("Output_Devices (lights on ) = "
21            & Byte'Image (Output_Devices));
22
23   Output_Devices := Output_Devices and
24                     Lights_Off;
25
26   Put_Line ("Output_Devices (lights off) = "
27            & Byte'Image (Output_Devices));
28 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 97) to do the above, which is even more elegant.

The notion of base in Ada allows for exponents, which is particularly pleasant. For instance, we can write:

Listing 16: literal_binaries.ads

```
1 package Literal_Binaries is
2   Kilobyte   : constant := 2#1#e+10;
3   Megabyte   : constant := 2#1#e+20;
4   Gigabyte   : constant := 2#1#e+30;
5   Terabyte   : constant := 2#1#e+40;
6   Petabyte   : constant := 2#1#e+50;
7   Exabyte    : constant := 2#1#e+60;
8   Zettabyte  : constant := 2#1#e+70;
9   Yottabyte  : constant := 2#1#e+80;
10 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 $2\#1\#e+10 = 1 \times 2^{10} = 1_024$ (in base 10), whereas $16\#F\#e+2 = 15 \times 16^2 = 15 \times 256 = 3_840$ (in base 10).

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

```
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 :=  
  0.33333333333333333333333333333333;  
Zero_Approx      : constant :=  
  1.0 - 3.0 * One_Third_Approx;
```

where Zero_Approx is really $1.0e-29$ — and that will show up in your numerical computations. The above is quite handy when we want to write fractions without any loss of precision. Here's the complete code:

Listing 17: ada_numeric_literals.adb

```
1 with Ada.Text_IO;  
2  
3 procedure Ada_Numeric_Literals is  
4   One_Third : constant := 3#1.0#e-1;  
5   -- same as 1.0/3.0  
6  
7   Zero      : constant := 1.0 - 3.0 * One_Third;  
8   pragma Assert (Zero = 0.0);  
9  
10  One_Third_Approx : constant :=  
11    0.33333333333333333333333333333333;  
12  Zero_Approx      : constant :=  
13    1.0 - 3.0 * One_Third_Approx;  
14  
15  use Ada.Text_IO;  
16  
17  begin  
18    Put_Line ("Zero      = "  
19              & Float'Image (Zero));  
20    Put_Line ("Zero_Approx = "  
21              & Float'Image (Zero_Approx));  
22  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 18: ada_numeric_literals.adb

```
1 with Ada.Text_IO;  
2  
3 with Literal_Binaries; use Literal_Binaries;  
4  
5 procedure Ada_Numeric_Literals is  
6  
7   Big_Sum : constant := 1      +  
8              Kilobyte +
```

(continues on next page)

(continued from previous page)

```

9           Megabyte +
10          Gigabyte +
11          Terabyte +
12          Petabyte +
13          Exabyte +
14          Zettabyte;
15
16  Result : constant := (Yottabyte - 1) /
17                      (Kilobyte - 1);
18
19  Nil    : constant := Result - Big_Sum;
20  pragma Assert (Nil = 0);
21
22  use Ada.Text_IO;
23
24  begin
25      Put_Line ("Nil          = "
26              & Integer'Image (Nil));
27  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.

7.3 Floating-Point Types

In this section, we discuss various attributes related to floating-point types.

In the Ada Reference Manual

- [3.5.8 Operations of Floating Point Types¹⁰⁶](#)
- [A.5.3 Attributes of Floating Point Types¹⁰⁷](#)

7.3.1 Representation-oriented attributes

In this section, we discuss attributes related to the representation of floating-point types.

¹⁰⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-8.html>

¹⁰⁷ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-3.html>

Attribute: Machine_Radix

Machine_Radix is an attribute that returns the radix of the hardware representation of a type. For example:

Listing 19: show_machine_radix.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Machine_Radix is
4 begin
5     Put_Line
6         ("Float'Machine_Radix:      "
7          & Float'Machine_Radix'Image);
8     Put_Line
9         ("Long_Float'Machine_Radix:  "
10         & Long_Float'Machine_Radix'Image);
11    Put_Line
12        ("Long_Long_Float'Machine_Radix: "
13         & Long_Long_Float'Machine_Radix'Image);
14 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 20: show_machine_mantissa.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Machine_Mantissa is
4 begin
5     Put_Line
6         ("Float'Machine_Mantissa:    "
7          & Float'Machine_Mantissa'Image);
8     Put_Line
9         ("Long_Float'Machine_Mantissa:  "
10         & Long_Float'Machine_Mantissa'Image);
11    Put_Line
12        ("Long_Long_Float'Machine_Mantissa: "
13         & Long_Long_Float'Machine_Mantissa'Image);
14 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 21: show_machine_emin_emax.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Machine_Emin_Emax is
4 begin
5   Put_Line
6     ("Float'Machine_Emin:      "
7     & Float'Machine_Emin'Image);
8   Put_Line
9     ("Float'Machine_Emax:      "
10    & Float'Machine_Emax'Image);
11  Put_Line
12    ("Long_Float'Machine_Emin:  "
13    & Long_Float'Machine_Emin'Image);
14  Put_Line
15    ("Long_Float'Machine_Emax:  "
16    & Long_Float'Machine_Emax'Image);
17  Put_Line
18    ("Long_Long_Float'Machine_Emin:  "
19    & Long_Long_Float'Machine_Emin'Image);
20  Put_Line
21    ("Long_Long_Float'Machine_Emax:  "
22    & Long_Long_Float'Machine_Emax'Image);
23 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
Long_Float'Machine_Emin: -1021
Long_Float'Machine_Emax: 1024
Long_Long_Float'Machine_Emin: -16381
Long_Long_Float'Machine_Emax: 16384
```

On a typical desktop PC, the value of `Float'Machine_Emin` and `Float'Machine_Emax` is -125 and 128, respectively.

To get the actual minimum and maximum value of the exponent for a specific type, we need to use the `Machine_Radix` attribute that we've seen previously. Let's calculate the minimum and maximum value of the exponent for the `Float` type on a typical PC:

- Value of minimum exponent: `Float'Machine_Radix ** Float'Machine_Emin`.
 - In our target platform, this is $2^{-125} = 2.35098870164457501594 \times 10^{-38}$.
- Value of maximum exponent: `Float'Machine_Radix ** Float'Machine_Emax`.
 - In our target platform, this is $2^{128} = 3.40282366920938463463 \times 10^{38}$.

Attribute: Digits

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

Listing 22: show_digits.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Digits is
4 begin
5   Put_Line ("Float'Digits:           "
6             & Float'Digits'Image);
7   Put_Line ("Long_Float'Digits:      "
8             & Long_Float'Digits'Image);
9   Put_Line ("Long_Long_Float'Digits: "
10            & Long_Long_Float'Digits'Image);
11 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 23: show_base_digits.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Base_Digits is
4   type Float_D3 is new Float digits 3;
5 begin
6   Put_Line ("Float_D3'Digits:         "
7             & Float_D3'Digits'Image);
8   Put_Line ("Float_D3'Base'Digits:    "
9             & Float_D3'Base'Digits'Image);
10 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**¹⁰⁸.
- `Signed_Zeros` is an attribute that indicates whether the type uses a sign for zero values, so it can represent both `-0.0` and `0.0`.
- `Machine_Rounds` is an attribute that indicates whether rounding-to-nearest is used, rather than some other choice (such as rounding-toward-zero).
- `Machine_Overflows` is an attribute that indicates whether a `Constraint_Error` exception is (or is not) guaranteed to be raised when an operation with that type produces an overflow or divide-by-zero.

Listing 24: `show_boolean_attributes.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Boolean_Attributes is
4  begin
5      Put_Line
6          ("Float'Denorm:           "
7           & Float'Denorm'Image);
8      Put_Line
9          ("Long_Float'Denorm:      "
10          & Long_Float'Denorm'Image);
11     Put_Line
12         ("Long_Long_Float'Denorm: "
13          & Long_Long_Float'Denorm'Image);
14     Put_Line
15         ("Float'Signed_Zeros:      "
16          & Float'Signed_Zeros'Image);
17     Put_Line
18         ("Long_Float'Signed_Zeros:  "
19          & Long_Float'Signed_Zeros'Image);
20     Put_Line
21         ("Long_Long_Float'Signed_Zeros: "
22          & Long_Long_Float'Signed_Zeros'Image);
23     Put_Line
24         ("Float'Machine_Rounds:    "
25          & Float'Machine_Rounds'Image);
26     Put_Line
27         ("Long_Float'Machine_Rounds:  "

```

(continues on next page)

¹⁰⁸ https://en.wikipedia.org/wiki/Subnormal_number

(continued from previous page)

```
28     & Long_Float'Machine_Rounds'Image);
29 Put_Line
30     ("Long_Long_Float'Machine_Rounds: "
31     & Long_Long_Float'Machine_Rounds'Image);
32 Put_Line
33     ("Float'Machine_Overflows:      "
34     & Float'Machine_Overflows'Image);
35 Put_Line
36     ("Long_Float'Machine_Overflows:  "
37     & Long_Float'Machine_Overflows'Image);
38 Put_Line
39     ("Long_Long_Float'Machine_Overflows: "
40     & Long_Long_Float'Machine_Overflows'Image);
41 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).

7.3.2 Primitive function attributes

In this section, we discuss attributes that we can use to manipulate floating-point values.

Attributes: Fraction, Exponent and Compose

The Exponent and Fraction attributes return "parts" of a floating-point value:

- Exponent returns the machine exponent, and
- Fraction returns the mantissa part.

Compose is used to return a floating-point value based on a fraction (the mantissa part) and the machine exponent.

Let's see some examples:

Listing 25: show_exponent_fraction_compose.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Exponent_Fraction_Compose is
4 begin
5   Put_Line
6     ("Float'Fraction (1.0):      "
7     & Float'Fraction (1.0)'Image);
8   Put_Line
9     ("Float'Fraction (0.25):    "
10    & Float'Fraction (0.25)'Image);
11  Put_Line
12    ("Float'Fraction (1.0e-25): "
13    & Float'Fraction (1.0e-25)'Image);
14  Put_Line
15    ("Float'Exponent (1.0):      "
16    & Float'Exponent (1.0)'Image);
17  Put_Line
18    ("Float'Exponent (0.25):    "
19    & Float'Exponent (0.25)'Image);
20  Put_Line
21    ("Float'Exponent (1.0e-25): "
22    & Float'Exponent (1.0e-25)'Image);
23  Put_Line
24    ("Float'Compose (5.00000e-01, 1): "
25    & Float'Compose (5.00000e-01, 1)'Image);
26  Put_Line
27    ("Float'Compose (5.00000e-01, -1): "
28    & Float'Compose (5.00000e-01, -1)'Image);
29  Put_Line
30    ("Float'Compose (9.67141E-01, -83): "
31    & Float'Compose (9.67141E-01, -83)'Image);
32 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

```

(continues on next page)

(continued from previous page)

```
Float'Fraction (1.0e-25): 9.67141E-01
Float'Exponent (1.0): 1
Float'Exponent (0.25): -1
Float'Exponent (1.0e-25): -83
Float'Compose (5.00000e-01, 1): 1.00000E+00
Float'Compose (5.00000e-01, -1): 2.50000E-01
Float'Compose (9.67141E-01, -83): 1.00000E-25
```

To understand this code example, we have to take this formula into account:

$$\text{Value} = \text{Fraction} \times \text{Machine_Radix}^{\text{Exponent}}$$

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:

$$0.5 \times 2^1 = 1.0$$

For the value 0.25, we get a fraction of 0.5 and a machine exponent of -1, which is the result of $0.5 \times 2^{-1} = 0.25$. We can use the `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 26: show_scaling.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Scaling is
4 begin
5   Put_Line ("Float'Scaling (0.25, 1): "
6             & Float'Scaling (0.25, 1)'Image);
7   Put_Line ("Float'Scaling (0.25, 2): "
8             & Float'Scaling (0.25, 2)'Image);
9   Put_Line ("Float'Scaling (0.25, 3): "
10            & Float'Scaling (0.25, 3)'Image);
11 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:

$$\text{scaling} = \text{value} \times \text{Machine_Radix}^{\text{machine exponent}}$$

For example, on a typical PC with a machine radix of two, `Float'Scaling (0.25, 3) = 2.0` corresponds to

$$0.25 \times 2^3 = 2.0$$

Round-up and round-down attributes

Floor and Ceiling are attributes that returned the rounded-down or rounded-up value, respectively, of a floating-point value. For example:

Listing 27: show_floor_ceiling.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Floor_Ceiling is
4 begin
5     Put_Line ("Float'Floor (0.25):  "
6               & Float'Floor (0.25)'Image);
7     Put_Line ("Float'Ceiling (0.25): "
8               & Float'Ceiling (0.25)'Image);
9 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 28: show_roundings.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Roundings is
4 begin
5     Put_Line
6         ("Float'Rounding (0.5):  "
7           & Float'Rounding (0.5)'Image);
8     Put_Line
9         ("Float'Rounding (1.5):  "
10          & Float'Rounding (1.5)'Image);
11    Put_Line
12        ("Float'Rounding (4.5):  "
13          & Float'Rounding (4.5)'Image);
14    Put_Line
15        ("Float'Rounding (-4.5): "
16          & Float'Rounding (-4.5)'Image);
17    Put_Line
18        ("Float'Unbiased_Rounding (0.5): "
```

(continues on next page)

(continued from previous page)

```
19     & Float'Unbiased_Rounding (0.5)'Image);
20   Put_Line
21     ("Float'Unbiased_Rounding (1.5): "
22     & Float'Unbiased_Rounding (1.5)'Image);
23   Put_Line
24     ("Float'Machine_Rounding (0.5): "
25     & Float'Machine_Rounding (0.5)'Image);
26   Put_Line
27     ("Float'Machine_Rounding (1.5): "
28     & Float'Machine_Rounding (1.5)'Image);
29 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 $2 \times 0.5 + 0.25 = 1.25$.

The Adjacent attribute is the next machine value towards another value. For example, on a typical PC, the adjacent value of a small value — say, 1.0×10^{-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 1.40130×10^{-45} . Note that the first parameter of the Adjacent attribute is the value we want to analyze and the second parameter is the Towards value.

Let's see a code example:

Listing 29: show_truncation_remainder_adjacent.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Truncation_Remainder_Adjacent is
4  begin
5      Put_Line
6          ("Float'Truncation (1.55): "
7           & Float'Truncation (1.55)'Image);
8      Put_Line
9          ("Float'Truncation (-1.55): "
10         & Float'Truncation (-1.55)'Image);
11     Put_Line
12         ("Float'Remainder (1.25, 0.25): "
13          & Float'Remainder (1.25, 0.25)'Image);
14     Put_Line
15         ("Float'Remainder (1.25, 0.5): "
16          & Float'Remainder (1.25, 0.5)'Image);
17     Put_Line
18         ("Float'Remainder (1.25, 1.0): "
19          & Float'Remainder (1.25, 1.0)'Image);
20     Put_Line
21         ("Float'Remainder (1.25, 2.0): "
22          & Float'Remainder (1.25, 2.0)'Image);
23     Put_Line
24         ("Float'Adjacent (1.0e-83, 0.0): "
25          & Float'Adjacent (1.0e-83, 0.0)'Image);
26     Put_Line
27         ("Float'Adjacent (1.0e-83, 1.0): "
28          & Float'Adjacent (1.0e-83, 1.0)'Image);
29  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 30: show_copy_sign_leading_part_machine.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Copy_Sign_Leading_Part_Machine is
4 begin
5   Put_Line
6     ("Float'Copy_Sign (1.0, -10.0): "
7     & Float'Copy_Sign (1.0, -10.0)'Image);
8   Put_Line
9     ("Float'Copy_Sign (-1.0, -10.0): "
10    & Float'Copy_Sign (-1.0, -10.0)'Image);
11  Put_Line
12    ("Float'Copy_Sign (1.0, 10.0): "
13    & Float'Copy_Sign (1.0, 10.0)'Image);
14  Put_Line
```

(continues on next page)

(continued from previous page)

```

15     ("Float'Copy_Sign (1.0, -0.0): "
16     & Float'Copy_Sign (1.0, -0.0)'Image);
17 Put_Line
18     ("Float'Copy_Sign (1.0, 0.0): "
19     & Float'Copy_Sign (1.0, 0.0)'Image);
20 Put_Line
21     ("Float'Leading_Part (1.75, 1): "
22     & Float'Leading_Part (1.75, 1)'Image);
23 Put_Line
24     ("Float'Leading_Part (1.75, 2): "
25     & Float'Leading_Part (1.75, 2)'Image);
26 Put_Line
27     ("Float'Leading_Part (1.75, 3): "
28     & Float'Leading_Part (1.75, 3)'Image);
29 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 1.0×10^{15} (or 1,000,000,000,000,000):

Listing 31: show_float_value.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Float_Value is
4      package F_IO is new
5          Ada.Text_IO.Float_IO (Float);
6
7      V : Float;
8  begin
9      F_IO.Default_Fore := 3;
10     F_IO.Default_Aft  := 1;
11     F_IO.Default_Exp  := 0;
12
13     V := 1.0E+15;
14     Put ("1.0E+15 = ");
15     F_IO.Put (Item => V);
16     New_Line;
17
18 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 1.0×10^{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 1.0×10^{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 (1.0×10^{15}) and the actual value that is assigned to V. We can do this by using the Machine attribute in the calculation:

Listing 32: show_machine_attribute.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Machine_Attribute is
4   package F_IO is new
5     Ada.Text_IO.Float_IO (Float);
6
7   V : Float;
8 begin
9   F_IO.Default_Fore := 3;
10  F_IO.Default_Aft  := 1;
11  F_IO.Default_Exp  := 0;
12
13  Put_Line
14    ("Original value: 1_000_000_000_000_000.0");
15
16  V := 1.0E+15;
17  Put ("Machine value: ");
18  F_IO.Put (Item => V);
19  New_Line;
20
21  V := 1.0E+15 - Float'Machine (1.0E+15);
22  Put ("Difference: ");
23  F_IO.Put (Item => V);
24  New_Line;
25
26 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 1.3009×10^7 . (Actually, the value that we might see is 1.3008896×10^7 , which is a version of 1.3009×10^7 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 1.0×10^{15} , while
- the second value in the operation is a version of the universal real value 1.0×10^{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 1.3008896×10^7 instead of 1.3009×10^7 when we run this application.

7.4 Fixed-Point Types

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

In the Ada Reference Manual

- 3.5.10 Operations of Fixed Point Types¹⁰⁹
- A.5.4 Attributes of Fixed Point Types¹¹⁰

7.4.1 Attributes of fixed-point types

Attribute: Machine_Radix

`Machine_Radix` is an attribute that returns the radix of the hardware representation of a type. For example:

Listing 33: `show_fixed_machine_radix.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Fixed_Machine_Radix is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5
6   D : constant := 2.0 ** (-31);
7   type TQ31 is delta D range -1.0 .. 1.0 - D;
8 begin
9   Put_Line ("T3_D3'Machine_Radix: "
10            & T3_D3'Machine_Radix'Image);
11   Put_Line ("TQ31'Machine_Radix: "
12            & TQ31'Machine_Radix'Image);
13 end Show_Fixed_Machine_Radix;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Fixed_Machine_
 ↪Radix
 MD5: a09d060a58f76550e948a8245ffb5fde

¹⁰⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-10.html>

¹¹⁰ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-4.html>

Runtime output

```
T3_D3'Machine_Radix: 2
TQ31'Machine_Radix: 2
```

Usually, this value is two, as the radix is based on a binary system.

Attribute: Machine_Rounds and Machine_Overflows

In this section, we discuss attributes that return **Boolean** values indicating whether a feature is available or not in the target architecture:

- `Machine_Rounds` is an attribute that indicates what happens when the result of a fixed-point operation is inexact:
 - `T'Machine_Rounds = True`: inexact result is rounded;
 - `T'Machine_Rounds = False`: inexact result is truncated.
- `Machine_Overflows` is an attribute that indicates whether a `Constraint_Error` is guaranteed to be raised when a fixed-point operation with that type produces an overflow or divide-by-zero.

Listing 34: show_boolean_attributes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Boolean_Attributes is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5
6   D : constant := 2.0 ** (-31);
7   type TQ31 is delta D range -1.0 .. 1.0 - D;
8 begin
9   Put_Line ("T3_D3'Machine_Rounds:  "
10            & T3_D3'Machine_Rounds'Image);
11   Put_Line ("TQ31'Machine_Rounds:  "
12            & TQ31'Machine_Rounds'Image);
13   Put_Line ("T3_D3'Machine_Overflows: "
14            & T3_D3'Machine_Overflows'Image);
15   Put_Line ("TQ31'Machine_Overflows: "
16            & TQ31'Machine_Overflows'Image);
17 end Show_Boolean_Attributes;
```

Attribute: Small and Delta

The `Small` and `Delta` attributes return numbers that indicate the numeric precision of a fixed-point type. In many cases, the `Small` of a type `T` is equal to the `Delta` of that type — i.e. `T'Small = T'Delta`. Let's discuss each attribute and how they distinguish from each other.

The `Delta` attribute returns the value of the `delta` that was used in the type definition. For example, if we declare `type T3_D3 is delta 10.0 ** (-3) digits D`, then the value of `T3_D3'Delta` is the `10.0 ** (-3)` that we used in the type definition.

The `Small` attribute returns the "small" of a type, i.e. the smallest value used in the machine representation of the type. The `small` must be at least equal to or smaller than the `delta` — in other words, it must conform to the `T'Small <= T'Delta` rule.

For further reading...

The `Small` and the `Delta` need not actually be small numbers. They can be arbitrarily large. For instance, they could be 1.0, or 1000.0. Consider the following example:

Listing 35: `fixed_point_defs.ads`

```

1 package Fixed_Point_Defs is
2   S      : constant := 32;
3   Exp    : constant := 128;
4   D      : constant := 2.0 ** (-S + Exp + 1);
5
6   type Fixed is delta D
7     range -1.0 * 2.0 ** Exp ..
8           1.0 * 2.0 ** Exp - D;
9
10  pragma Assert (Fixed'Size = S);
11 end Fixed_Point_Defs;
```

Listing 36: `show_fixed_type_info.adb`

```

1 with Fixed_Point_Defs; use Fixed_Point_Defs;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Show_Fixed_Type_Info is
5 begin
6   Put_Line ("Size : "
7             & Fixed'Size'Image);
8   Put_Line ("Small : "
9             & Fixed'Small'Image);
10  Put_Line ("Delta : "
11           & Fixed'Delta'Image);
12  Put_Line ("First : "
13           & Fixed'First'Image);
14  Put_Line ("Last : "
15           & Fixed'Last'Image);
16 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: 1.58456325028528675²⁹. (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 -3.4028³⁸ and 3.4028³⁸.)

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: $T'_{Small} \leq T'_{Delta}$.
- If we want, however, to specify the *small*, we can do that by using the `Small` aspect. In this case, it doesn't need to be a power of two.

For decimal fixed-point types, we cannot specify the *small*. In this case, it's automatically selected by the compiler, and it's always equal to the *delta*.

Let's see an example:

Listing 37: fixed_small_delta.ads

```
1 package Fixed_Small_Delta is
2   D3 : constant := 10.0 ** (-3);
3
4   type T3_D3 is delta D3 digits 3;
5
6   type TD3 is delta D3 range -1.0 .. 1.0 - D3;
7
8   D31 : constant := 2.0 ** (-31);
9   D15 : constant := 2.0 ** (-15);
10
11  type TQ31 is delta D31 range -1.0 .. 1.0 - D31;
12
13  type TQ15 is delta D15 range -1.0 .. 1.0 - D15
14     with Small => D31;
15 end Fixed_Small_Delta;
```

Listing 38: show_fixed_small_delta.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Fixed_Small_Delta; use Fixed_Small_Delta;
4
5 procedure Show_Fixed_Small_Delta is
6 begin
7   Put_Line ("T3_D3'Small: "
8     & T3_D3'Small'Image);
9   Put_Line ("T3_D3'Delta: "
10    & T3_D3'Delta'Image);
11  Put_Line ("T3_D3'Size: "
12    & T3_D3'Size'Image);
13  Put_Line ("-----");
14
15  Put_Line ("TD3'Small: "
16    & TD3'Small'Image);
17  Put_Line ("TD3'Delta: "
18    & TD3'Delta'Image);
19  Put_Line ("TD3'Size: "
20    & TD3'Size'Image);
21  Put_Line ("-----");
22
23  Put_Line ("TQ31'Small: "
24    & TQ31'Small'Image);
25  Put_Line ("TQ31'Delta: "
26    & TQ31'Delta'Image);
27  Put_Line ("TQ32'Size: "
28    & TQ31'Size'Image);
29  Put_Line ("-----");
30
31  Put_Line ("TQ15'Small: "
```

(continues on next page)

(continued from previous page)

```

32         & TQ15'Small'Image);
33     Put_Line ("TQ15'Delta: "
34             & TQ15'Delta'Image);
35     Put_Line ("TQ15'Size: "
36             & TQ15'Size'Image);
37 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:  1.000000000000000000E-03
T3_D3'Delta:  1.000000000000000000E-03
T3_D3'Size:  11
-----
TD3'Small:   9.765625000000000000E-04
TD3'Delta:   1.000000000000000000E-03
TD3'Size:   11
-----
TQ31'Small:  4.65661287307739258E-10
TQ31'Delta:  4.65661287307739258E-10
TQ32'Size:   32
-----
TQ15'Small:  4.65661287307739258E-10
TQ15'Delta:  3.051757812500000000E-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 39: show_fixed_small_delta.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Fixed_Small_Delta; use Fixed_Small_Delta;
4
5  procedure Show_Fixed_Small_Delta is
6      V : TQ15;
7  begin
8      Put_Line ("V'Size: " & V'Size'Image);
9
10     V := TQ15'Small;
11     Put_Line ("V: " & V'Image);
12
13     V := TQ15'Delta;
14     Put_Line ("V: " & V'Image);
15 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 40: show_fixed_fore_aft.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Fixed_Fore_Aft is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5
6   D : constant := 2.0 ** (-31);
7   type TQ31 is delta D range -1.0 .. 1.0 - D;
8
9   Dec : constant T3_D3 := -0.123;
10  Fix : constant TQ31 := -TQ31'Delta;
11 begin
12   Put_Line ("T3_D3'Fore: "
13     & T3_D3'Fore'Image);
14   Put_Line ("T3_D3'Aft: "
15     & T3_D3'Aft'Image);
16
17   Put_Line ("TQ31'Fore: "
18     & TQ31'Fore'Image);
19   Put_Line ("TQ31'Aft: "
20     & TQ31'Aft'Image);
21   Put_Line ("----");
22   Put_Line ("Dec: "
23     & Dec'Image);
24   Put_Line ("Fix: "
25     & Fix'Image);
26 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"`.

7.4.2 Attributes of decimal fixed-point types

The attributes presented in this subsection are only available for decimal fixed-point types.

Attribute: Digits

Digits is an attribute that returns the number of significant decimal digits of a decimal fixed-point subtype. This corresponds to the value that we use for the **digits** in the definition of a decimal fixed-point type.

Let's see an example:

Listing 41: show_decimal_digits.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Decimal_Digits is
4   type T3_D6 is delta 10.0 ** (-3) digits 6;
5   subtype T3_D2 is T3_D6 digits 2;
6 begin
7   Put_Line ("T3_D6'Digits: "
8             & T3_D6'Digits'Image);
9   Put_Line ("T3_D2'Digits: "
10            & T3_D2'Digits'Image);
11 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 42: show_decimal_scale.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Decimal_Scale is
4   type TM3_D6 is delta 10.0 ** 3 digits 6;
5   type T3_D6  is delta 10.0 ** (-3) digits 6;
6   type T9_D12 is delta 10.0 ** (-9) digits 12;
7 begin
8   Put_Line ("TM3_D6'Scale: "
9             & TM3_D6'Scale'Image);
10  Put_Line ("T3_D6'Scale: "
11           & T3_D6'Scale'Image);
12  Put_Line ("T9_D12'Scale: "
13           & T9_D12'Scale'Image);
14 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 43: show_decimal_round.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Decimal_Round is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5 begin
6   Put_Line ("T3_D3'Round (0.2774): "
7             & T3_D3'Round (0.2774)'Image);
8   Put_Line ("T3_D3'Round (0.2777): "
9             & T3_D3'Round (0.2777)'Image);
10 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.

7.5 Big Numbers

As we've seen before, we can define numeric types in Ada with a high degree of precision. However, these normal numeric types in Ada are limited to what the underlying hardware actually supports. For example, any signed integer type — whether defined by the language or the user — cannot have a range greater than that of System.Min_Int .. System.Max_Int because those constants reflect the actual hardware's signed integer types. In certain applications, that precision might not be enough, so we have to rely on [arbitrary-precision arithmetic](#)¹¹¹. 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:

¹¹¹ https://en.wikipedia.org/wiki/arbitrary-precision_arithmetic

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](#)¹¹²
 - [Big Integers](#)¹¹³
 - [Big Reals](#)¹¹⁴
-

7.5.1 Overview

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

Listing 44: show_simple_big_numbers.adb

```
1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  with Ada.Numerics.Big_Numbers.Big_Integers;
6  use  Ada.Numerics.Big_Numbers.Big_Integers;
7
8  with Ada.Numerics.Big_Numbers.Big_Reals;
9  use  Ada.Numerics.Big_Numbers.Big_Reals;
10
11 procedure Show_Simple_Big_Numbers is
12     BI : Big_Integer;
13     BR : Big_Real;
14 begin
15     BI := 12345678901234567890;
16     BR := 2.0 ** 1234;
17
18     Put_Line ("BI: " & BI'Image);
19     Put_Line ("BR: " & BR'Image);
20
21     BI := BI + 1;
22     BR := BR + 1.0;
23
24     Put_Line ("BI: " & BI'Image);
25     Put_Line ("BR: " & BR'Image);
26 end Show_Simple_Big_Numbers;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Numbers
MD5: d25e0c73ef04b6c950f2ab63fc96a353

Runtime output

¹¹² <http://www.ada-auth.org/standards/22rm/html/RM-A-5-5.html>

¹¹³ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-6.html>

¹¹⁴ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-7.html>

```
BI: 12345678901234567890
BR: ↵
↪29581122460809862906004469571610359078633968713537299223955620705065735079623892426105383724837
↪000
BI: 12345678901234567891
BR: ↵
↪29581122460809862906004469571610359078633968713537299223955620705065735079623892426105383724837
↪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 45: show_simple_big_numbers_operators.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Ada.Numerics.Big_Numbers.Big_Integers;
6 use Ada.Numerics.Big_Numbers.Big_Integers;
7
8 procedure Show_Simple_Big_Numbers_Operators is
9     BI : Big_Integer;
10 begin
11     BI := 12345678901234567890;
12
13     Put_Line ("BI: " & BI'Image);
14
15     BI := -BI + BI / 2;
16     BI := BI - BI * 2;
17
18     Put_Line ("BI: " & BI'Image);
19 end Show_Simple_Big_Numbers_Operators;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Numbers_
↪Operators
MD5: c4f405e3ea916bc8a3f309acdeb0606a
```

Runtime output

```
BI: 12345678901234567890
BI: 6172839450617283945
```

In this example, we're applying the four basic operators (+, -, *, /) on big integers.

7.5.2 Factorial

A typical example is the `factorial`¹¹⁵: a sequence of the factorial of consecutive small numbers can quickly lead to big numbers. Let's take this implementation as an example:

Listing 46: factorial.ads

```
1 function Factorial (N : Integer)
2   return Long_Long_Integer;
```

Listing 47: factorial.adb

```
1 function Factorial (N : Integer)
2   return Long_Long_Integer is
3   Fact : Long_Long_Integer := 1;
4 begin
5   for I in 2 .. N loop
6     Fact := Fact * Long_Long_Integer (I);
7   end loop;
8
9   return Fact;
10 end Factorial;
```

Listing 48: show_factorial.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Factorial;
4
5 procedure Show_Factorial is
6 begin
7   for I in 1 .. 50 loop
8     Put_Line (I'Image & "! = "
9               & Factorial (I)'Image);
10  end loop;
11 end Show_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
```

(continues on next page)

¹¹⁵ <https://en.wikipedia.org/wiki/Factorial>

(continued from previous page)

```
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 49: factorial.ads

```
1 pragma Ada_2022;
2
3 with Ada.Numerics.Big_Numbers.Big_Integers;
4 use Ada.Numerics.Big_Numbers.Big_Integers;
5
6 function Factorial (N : Integer)
7     return Big_Integer;
```

Listing 50: factorial.adb

```
1 function Factorial (N : Integer)
2     return Big_Integer is
3     Fact : Big_Integer := 1;
4 begin
5     for I in 2 .. N loop
6         Fact := Fact * To_Big_Integer (I);
7     end loop;
8
9     return Fact;
10 end Factorial;
```

Listing 51: show_big_number_factorial.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Factorial;
6
7 procedure Show_Big_Number_Factorial is
8 begin
9     for I in 1 .. 50 loop
10        Put_Line (I'Image & "! = "
11                & Factorial (I)'Image);
12    end loop;
13 end Show_Big_Number_Factorial;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Factorial_Big_Numbers
MD5: 18b6e168dac40422a1f0334fe5e4486e
```

Runtime output

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
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! = 604152630633738356373551320685139975072645120000000000
44! = 26582715747884487680436258110146158903196385280000000000
45! = 1196222208654801945619631614956577150643837337600000000000
46! = 55026221598120889498503054288002548929616517529600000000000
47! = 2586232415111681806429643551536119799691976323891200000000000
48! = 124139155925360726708622890473733750385214863546777600000000000
49! = 6082818640342675608722521633212953768875528313792102400000000000
50! = 304140932017133780436126081660647688443776415689605120000000000000
```

As we can see in this example, replacing the **Long_Long_Integer** type by the `Big_Integer` type fixes the problem (the runtime exception) that we had in the previous version. (Note that we're using the `To_Big_Integer` function to convert from **Integer** to `Big_Integer`: we discuss these conversions next.)

Note that there is a limit to the upper bounds for big integers. However, this limit isn't dependent on the hardware types — as it's the case for normal numeric types —, but rather compiler specific. In other words, the compiler can decide how much memory it wants to use to represent big integers.

7.5.3 Conversions

Most probably, we want to mix big numbers and *standard* numbers (i.e. integer and real numbers) in our application. In this section, we talk about the conversion between big numbers and standard types.

Validity

The package specifications of big numbers include subtypes that *ensure* that 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	<ul style="list-style-type: none"> To_Big_Integer 	<ul style="list-style-type: none"> To_Integer (Integer) From_Big_Integer (other integer types)
Big Reals	<ul style="list-style-type: none"> To_Big_Real (floating-point types or fixed-point types) 	<ul style="list-style-type: none"> From_Big_Real
	<ul style="list-style-type: none"> To_Big_Real (Valid_Big_Integer) To_Real (Integer) 	<ul style="list-style-type: none"> 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 52: `show_simple_big_integer_conversion.adb`

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Ada.Numerics.Big_Numbers.Big_Integers;
6 use Ada.Numerics.Big_Numbers.Big_Integers;
7
8 procedure Show_Simple_Big_Integer_Conversion is
9     BI : Big_Integer;
10    I  : Integer := 10000;
11 begin
12     BI := To_Big_Integer (I);
13     Put_Line ("BI: " & BI'Image);
14
15     I := To_Integer (BI + 1);
16     Put_Line ("I: " & I'Image);
17 end Show_Simple_Big_Integer_Conversion;
```

Code block metadata

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

Runtime output

```
BI: 10000
I: 10001
```

In addition, we can use the generic `Signed_Conversions` and `Unsigned_Conversions` packages to convert between `Big_Integer` and any signed or unsigned integer types:

Listing 53: `show_arbitrary_big_integer_conversion.adb`

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Ada.Numerics.Big_Numbers.Big_Integers;
6 use Ada.Numerics.Big_Numbers.Big_Integers;
7
8 procedure Show_Arbitrary_Big_Integer_Conversion is
9
10    type Mod_32_Bit is mod 2 ** 32;
11
12    package Long_Long_Integer_Conversions is new
13        Signed_Conversions (Long_Long_Integer);
14    use Long_Long_Integer_Conversions;
15
16    package Mod_32_Bit_Conversions is new
17        Unsigned_Conversions (Mod_32_Bit);
18    use Mod_32_Bit_Conversions;
```

(continues on next page)

(continued from previous page)

```

19
20   BI   : Big_Integer;
21   LLI  : Long_Long_Integer := 10000;
22   U_32 : Mod_32_Bit      := 2 ** 32 + 1;
23
24 begin
25   BI := To_Big_Integer (LLI);
26   Put_Line ("BI: " & BI'Image);
27
28   LLI := From_Big_Integer (BI + 1);
29   Put_Line ("LLI: " & LLI'Image);
30
31   BI := To_Big_Integer (U_32);
32   Put_Line ("BI: " & BI'Image);
33
34   U_32 := From_Big_Integer (BI + 1);
35   Put_Line ("U_32: " & U_32'Image);
36
37 end Show_Arbitrary_Big_Integer_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Arbitrary_Big_Integer_Conversion
 ↪ Integer_Conversion
 MD5: 21466010594cf09f37776bc8cb61ee9c

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 54: show_big_real_floating_point_conversion.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  with Ada.Numerics.Big_Numbers.Big_Reals;
6  use  Ada.Numerics.Big_Numbers.Big_Reals;
7
8  procedure Show_Big_Real_Floating_Point_Conversion
9  is
10     type D10 is digits 10;
11

```

(continues on next page)

(continued from previous page)

```
12 package D10_Conversions is new
13   Float_Conversions (D10);
14 use D10_Conversions;
15
16 package Long_Float_Conversions is new
17   Float_Conversions (Long_Float);
18 use Long_Float_Conversions;
19
20 BR : Big_Real;
21 LF : Long_Float := 2.0;
22 F10 : D10      := 1.999;
23
24 begin
25   BR := To_Big_Real (LF);
26   Put_Line ("BR:  " & BR'Image);
27
28   LF := From_Big_Real (BR + 1.0);
29   Put_Line ("LF:  " & LF'Image);
30
31   BR := To_Big_Real (F10);
32   Put_Line ("BR:  " & BR'Image);
33
34   F10 := From_Big_Real (BR + 0.1);
35   Put_Line ("F10: " & F10'Image);
36
37 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: 531c59a06b46c2074bc5378b5dcddd35

Runtime output

```
BR:    2.000
LF:    3.000000000000000E+00
BR:    1.999
F10:   2.0990000000E+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 55: `show_big_real_fixed_point_conversion.adb`

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Ada.Numerics.Big_Numbers.Big_Reals;
6 use Ada.Numerics.Big_Numbers.Big_Reals;
7
```

(continues on next page)

(continued from previous page)

```

8  procedure Show_Big_Real_Fixed_Point_Conversion
9  is
10     D : constant := 2.0 ** (-31);
11     type TQ31 is delta D range -1.0 .. 1.0 - D;
12
13     package TQ31_Conversions is new
14         Fixed_Conversions (TQ31);
15     use TQ31_Conversions;
16
17     BR   : Big_Real;
18     FQ31 : TQ31 := 0.25;
19
20 begin
21     BR := To_Big_Real (FQ31);
22     Put_Line ("BR: " & BR'Image);
23
24     FQ31 := From_Big_Real (BR * 2.0);
25     Put_Line ("FQ31: " & FQ31'Image);
26
27 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: 94a87bfc6ffad70f757cfc8b6ae32530

```

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 56: `show_big_real_big_integer_conversion.adb`

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  with Ada.Numerics.Big_Numbers.Big_Integers;
6  use Ada.Numerics.Big_Numbers.Big_Integers;
7
8  with Ada.Numerics.Big_Numbers.Big_Reals;
9  use Ada.Numerics.Big_Numbers.Big_Reals;
10
11 procedure Show_Big_Real_Big_Integer_Conversion
12 is
13     I : Integer;

```

(continues on next page)

(continued from previous page)

```
14 BI : Big_Integer;
15 BR : Big_Real;
16
17 begin
18   I := 12345;
19   BR := To_Real (I);
20   Put_Line ("BR (from I): " & BR'Image);
21
22   BI := 123456;
23   BR := To_Big_Real (BI);
24   Put_Line ("BR (from BI): " & BR'Image);
25
26 end Show_Big_Real_Big_Integer_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Big_Integer_
↳Conversion
MD5: 9a217c0551bc80269596d7217d2be879
```

Runtime output

```
BR (from I): 12345.000
BR (from BI): 123456.000
```

Here, we use the `To_Real` and the `To_Big_Real` and functions for the conversions.

String conversions

In addition to that, we can use string conversions:

Listing 57: `show_big_number_string_conversion.adb`

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Ada.Numerics.Big_Numbers.Big_Integers;
6 use Ada.Numerics.Big_Numbers.Big_Integers;
7
8 with Ada.Numerics.Big_Numbers.Big_Reals;
9 use Ada.Numerics.Big_Numbers.Big_Reals;
10
11 procedure Show_Big_Number_String_Conversion
12 is
13   BI : Big_Integer;
14   BR : Big_Real;
15 begin
16   BI := From_String ("12345678901234567890");
17   BR := From_String ("12345678901234567890.0");
18
19   Put_Line ("BI: "
20             & To_String (Arg => BI,
21                           Width => 5,
22                           Base => 2));
23   Put_Line ("BR: "
24             & To_String (Arg => BR,
25                           Fore => 2,
26                           Aft => 6,
```

(continues on next page)

(continued from previous page)

```
27         Exp => 18));
28 end Show_Big_Number_String_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Number_String_
↳Conversion
MD5: 3819df198ec140b457fa56a65d8876f9
```

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 68), which we discussed previously. In fact, these are the corresponding type declarations:

```
-- Declaration from
-- Ada.Numerics.Big_Numbers.Big_Integers;

type Big_Integer is private
  with Integer_Literal => From_Universal_Image,
       Put_Image       => Put_Image;

function From_Universal_Image
  (Arg : String)
  return Valid_Big_Integer
  renames From_String;

-- Declaration from
-- Ada.Numerics.Big_Numbers.Big_Reals;

type Big_Real is private
  with Real_Literal => From_Universal_Image,
       Put_Image    => Put_Image;

function From_Universal_Image
  (Arg : String)
  return Valid_Big_Real
  renames From_String;
```

As we can see in these declarations, the `From_String` function renames the `From_Universal_Image` function, which is being used for the user-defined literals.

Also, we call the `To_String` function to get a string for the big numbers. Naturally, using the `To_String` function instead of the `Image` attribute — as we did in previous examples — allows us to customize the format of the string that we display in the user message.

7.5.4 Other features of big integers

Now, let's look at two additional features of big integers:

- the natural and positive subtypes, and
- other available operators and functions.

Big positive and natural subtypes

Similar to integer types, big integers have the `Big_Natural` and `Big_Positive` subtypes to indicate natural and positive numbers. However, in contrast to the **Natural** and **Positive** subtypes, the `Big_Natural` and `Big_Positive` subtypes are defined via predicates rather than the simple ranges of normal (ordinary) numeric types:

```
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 58: show_big_positive_natural.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Ada.Numerics.Big_Numbers.Big_Integers;
6 use Ada.Numerics.Big_Numbers.Big_Integers;
7
8 procedure Show_Big_Positive_Natural is
9   BI, D, N : Big_Integer;
10 begin
11   D := 3;
12   N := 2;
13   BI := Big_Natural (D / Big_Positive (N));
14
15   Put_Line ("BI: " & BI'Image);
16 end Show_Big_Positive_Natural;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Positive_Natural
MD5: 6debfb86e11c7bfa3dbaf2d81eb24360

Runtime output

```
BI: 1
```

By using the `Big_Natural` and `Big_Positive` subtypes in the calculation above (in the assignment to `BI`), we ensure that we don't perform a division by zero, and that the result of the calculation is a natural number.

7.5.5 Other operators for big integers

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

Listing 59: `show_big_integer_rem_mod.adb`

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  with Ada.Numerics.Big_Numbers.Big_Integers;
6  use  Ada.Numerics.Big_Numbers.Big_Integers;
7
8  procedure Show_Big_Integer_Rem_Mod is
9      BI : Big_Integer;
10     begin
11         BI := 145 mod (-4);
12         Put_Line ("BI (mod): " & BI'Image);
13
14         BI := 145 rem (-4);
15         Put_Line ("BI (rem): " & BI'Image);
16     end Show_Big_Integer_Rem_Mod;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Integer_Rem_Mod
MD5: 079f2f88f98f52e81ae7719d4629ca08

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 60: `show_big_integer_greatest_common_divisor.adb`

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  with Ada.Numerics.Big_Numbers.Big_Integers;
6  use  Ada.Numerics.Big_Numbers.Big_Integers;
7
8  procedure Show_Big_Integer_Greatest_Common_Divisor
9      is
10     BI : Big_Integer;
11     begin
12         BI := Greatest_Common_Divisor (145, 25);
13         Put_Line ("BI: " & BI'Image);

```

(continues on next page)

(continued from previous page)

```
14
15 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: b2d0098fccaf949f228276b4d862b56
```

Runtime output

```
BI: 5
```

In this example, we retrieve the greatest common divisor of 145 and 25 (i.e.: 5).

7.5.6 Big real and quotients

An interesting feature of big reals is that they support quotients. In fact, we can simply assign $2/3$ to a big real variable. (Note that we're able to omit the decimal points, as we write $2/3$ instead of $2.0 / 3.0$.) For example:

Listing 61: show_big_real_quotient_conversion.adb

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Ada.Numerics.Big_Numbers.Big_Reals;
6 use Ada.Numerics.Big_Numbers.Big_Reals;
7
8 procedure Show_Big_Real_Quotient_Conversion
9 is
10 BR : Big_Real;
11 begin
12 BR := 2 / 3;
13 -- Same as:
14 -- BR := From_Quotient_String ("2 / 3");
15
16 Put_Line ("BR: " & BR'Image);
17
18 Put_Line ("Q: "
19 & To_Quotient_String (BR));
20
21 Put_Line ("Q numerator: "
22 & Numerator (BR)'Image);
23 Put_Line ("Q denominator: "
24 & Denominator (BR)'Image);
25 end Show_Big_Real_Quotient_Conversion;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Quotient_
↳Conversion
MD5: 4ef8355332e73a1f7da036b8e1e4b898
```

Runtime output

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

(continues on next page)

(continued from previous page)

```
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.

7.5.7 Range checks

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

Listing 62: show_big_numbers_in_range.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  with Ada.Numerics.Big_Numbers.Big_Integers;
6  use  Ada.Numerics.Big_Numbers.Big_Integers;
7
8  with Ada.Numerics.Big_Numbers.Big_Reals;
9  use  Ada.Numerics.Big_Numbers.Big_Reals;
10
11 procedure Show_Big_Numbers_In_Range is
12
13     BI : Big_Integer;
14     BR : Big_Real;
15
16     BI_From : constant Big_Integer := 0;
17     BI_To   : constant Big_Integer := 1024;
18
19     BR_From : constant Big_Real := 0.0;
20     BR_To   : constant Big_Real := 1024.0;
21
22 begin
23     BI := 1023;
24     BR := 1023.9;
25
26     if In_Range (BI, BI_From, BI_To) then
27         Put_Line ("BI ("
28                 & BI'Image
29                 & ") is in the "
30                 & BI_From'Image
31                 & " .. "
32                 & BI_To'Image
33                 & " range");
34     end if;
35
36     if In_Range (BR, BR_From, BR_To) then
37         Put_Line ("BR ("
38                 & BR'Image
39                 & ") is in the "
40                 & BR_From'Image
41                 & " .. "
42                 & BR_To'Image
43                 & " range");

```

(continues on next page)

(continued from previous page)

```
44     end if;  
45  
46 end Show_Big_Numbers_In_Range;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Numbers_In_Range
MD5: 9c85e8374db1095142260f45c4c4e7e1

Runtime output

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

In this example, we call the `In_Range` function to check whether the big integer number (BI) and the big real number (BR) are in the range between 0 and 1024.

Part II

Control Flow

EXPRESSIONS

8.1 Expressions: Definition

According to the Ada Reference Manual, an expression "is a formula that defines the computation or retrieval of a value." Also, when an expression is evaluated, the computed or retrieved value always has an associated type known at compile-time.

Even though the definition above is very simple, Ada expressions are actually very flexible — and they can also be very complex. In fact, if you read the [corresponding section](#)¹¹⁶ 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](#)¹¹⁷
-

8.1.1 Relations and simple expressions

Expressions usually consist of relations, which in turn consist of simple expressions. (There are more details to this, but we'll keep it simple for the moment.) Let's see a code example with a few expressions, which we dissect into the corresponding grammatical elements (we're going to discuss them later):

Listing 1: show_expression_elements.adb

```
1 procedure Show_Expression_Elements is
2   type Mode is (Off, A, B, C, D);
3
4   pragma Unreferenced (B, C, D);
5
6   subtype Active_Mode is Mode
7     range Mode'Succ (Off) .. Mode'Last;
8
9   M1, M2 : Mode;
10  Dummy   : Boolean;
11 begin
12   M1 := A;
13
```

(continues on next page)

¹¹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

¹¹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

(continued from previous page)

```

14 Dummy :=
15     M1 in Active_Mode
16         and then M2 in Off | A;
17
18     -- ^^^^^^^^^^^^^^^^^^ relation
19
20     --             ^^^^^^^^^^^^^^^^^^ relation
21     -- ^^^^^^^^^^^^^^^^^^
22     --                                 expression
23
24 Dummy :=
25     M1 in Active_Mode;
26     -- ^^ name
27     -- ^^ primary
28     -- ^^ factor
29     -- ^^ term
30     -- ^^ simple expression
31
32     --             ^^^^^^^^^^^^^ membership choice
33     --             ^^^^^^^^^^^^^ membership choice list
34
35     -- ^^^^^^^^^^^^^^^^^^ relation
36     -- ^^^^^^^^^^^^^^^^^^ expression
37
38 Dummy :=
39     M2 in Off | A;
40     -- ^^ name
41     -- ^^ primary
42     -- ^^ factor
43     -- ^^ term
44     -- ^^ simple expression
45
46     --             ^^ membership choice
47     --             ^ membership choice
48     --             ^^^^^ membership choice list
49
50     -- ^^^^^^^^^^^^^^^^^^ relation
51     -- ^^^^^^^^^^^^^^^^^^ expression
52
53 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 167);
 - a name;
 - an allocator (like **new Integer**);
 - *a parenthesized expression* (page 323);
 - *a conditional expression* (page 326);
 - *a quantified expression* (page 328);
 - *a declare expression* (page 332).

For further reading...

The definition of simple expressions we've just seen is very simplified. In actuality, these are the grammatical elements specified in the Ada Reference Manual:

```
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 326), *quantified expressions* (page 328) and *declare expressions* (page 332) 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}
```

(continues on next page)

(continued from previous page)

```

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 expressions¹¹⁸](#) of the Ada Reference Manual.

In the Ada Reference Manual

- [4.4 Expressions¹¹⁹](#)
- [4.5.2 Relational Operators and Membership Tests¹²⁰](#)

8.1.2 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

```

1  procedure Show_Numeric_Expressions is
2    C1   : constant Integer := 5;
3    Dummy : Integer;
4  begin
5    Dummy :=
6      -2 ** 4 + 3 * C1 ** 8;
7      --           ^ numeric literal
8      --           ^ primary
9      --           ^^ name
10     --           ^^ primary
11     --           ^^^^^ factor
12     --           ^ multiplying operator
13     --           ^ numeric literal
14     --           ^ primary
15     --           ^ factor
16     --           ^^^^^^^^^ term
17     --
18     --           ^ numeric literal
19     --           ^ primary
20     --           ^ numeric literal
21     --           ^ primary
22     --           ^^^^^ factor
23     --           ^^^^^ term
24     --           ^ binary adding operator
25     --           ^ unary adding operator
26     --
27     --           ^^^^^^^^^^^^^^^^^ simple expression
28     --
    
```

(continues on next page)

¹¹⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>
¹¹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>
¹²⁰ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-2.html>

(continued from previous page)

```
29  -- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expression
30  end Show_Numeric_Expressions;
```

Code block metadata

```
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](#)¹²¹

8.1.3 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](#)¹²² of the Ada Reference Manual for further details.

8.1.4 Parenthesized expression

An interesting aspect of primaries is that, by using parentheses, we can embed an expression inside another expression. As an example, let's discuss the following expression and its elements:

Listing 3: show_parenthesized_expressions.adb

```
1  procedure Show_Parenthesized_Expressions is
2     C1 : constant Integer := 4;
3     C2 : constant Integer := 5;
4
5     Dummy : Integer;
6  begin
7     Dummy :=
8         (2 + C1) * C2;
9     --     ^^      name
10    --     ^^      primary
11    --     ^^      factor
12    --     ^^      term
13    --
```

(continues on next page)

¹²¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>¹²² <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

(continued from previous page)

```

14  -- ^           numeric literal
15  -- ^           primary
16  -- ^           factor
17  -- ^           term
18  --
19  -- ^           binary adding operator
20  -- ^^^^^^^^^ simple expression
21  --
22  -- ^^^^^^^^^ expression
23  -- ^^^^^^^^^ primary
24  -- ^^^^^^^^^ factor
25  --
26  -- ^^^ factor
27  -- ^^^^^^^^^ term
28  --
29  -- ^^^^^^^^^ simple expression
30  --
31  -- ^^^^^^^^^ expression
32  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

```

1  procedure Show_Name_In_Expression is
2     type Mode is (Off, A, B, C, D);
3
4     M1 : Mode;
5  begin
6     M1 := A;
7
8     case M1 is
9         when Off | D =>
10            null;
11         when A | B | C =>
12            M1 := D;

```

(continues on next page)

(continued from previous page)

```

13   end case;
14
15 end Show_Name_In_Expression;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.Name_
↳In_Expression
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 351).

Let's look at another example, this time with a subprogram call:

Listing 5: increment_by_one.ads

```

1 procedure Increment_By_One (I : in out Integer);

```

Listing 6: increment_by_one.adb

```

1 procedure Increment_By_One (I : in out Integer) is
2 begin
3   I := I + 1;
4 end Increment_By_One;

```

Listing 7: show_name_in_expression.adb

```

1 with Increment_By_One;
2
3 procedure Show_Name_In_Expression is
4   V : Integer := 0;
5 begin
6   Increment_By_One ((V));
7 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.)

8.2 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`

```

1 package Expr_Func is
2
3     function Max (A, B : Integer) return Integer is
4         (if A >= B then A else B);
5
6 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`

```

1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State is
6         (if S = Off
7            then On
8            elsif S = On
9            then Off
10           else S);
11
12 end Expr_Func;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
 ↳ Conditional_If_Expressions_1
 MD5: 7a99711afec0b481557f9874dfbf4de

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`

```

1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State;
6
7 end Expr_Func;
```

Listing 11: expr_func.adb

```

1 package body Expr_Func is
2
3     function Toggled (S : State) return State is
4     begin
5         if S = Off then
6             return On;
7         elsif S = On then
8             return Off;
9         else
10            return S;
11        end if;
12    end Toggled;
13
14 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

```

1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State is
6     (case S is
7         when Off    => On,
8         when On     => Off,
9         when others => S);
10
11 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

```

1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State;
6
7 end Expr_Func;

```

Listing 14: expr_func.adb

```
1 package body Expr_Func is
2
3     function Toggled (S : State) return State is
4     begin
5         case S is
6             when Off    => return On;
7             when On     => return Off;
8             when others => return S;
9         end case;
10    end Toggled;
11
12 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](#)¹²³
-

8.3 Quantified Expressions

Quantified expressions are **for** expressions using a quantifier — which can be either **all** or **some** — and a predicate. This kind of expressions let us formalize statements such as:

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

In the quantified expression **for all I in A'Range => A (I) = 0**, the quantifier is **all** and the predicate is **A (I) = 0**. In the second expression, the quantifier is **some**. The result of a quantified expression is always a Boolean value.

For example, we could use the quantified expressions above and implement these two functions:

- **Is_Zero**, which checks whether all components of an array A are zero, and
- **Has_Zero**, which checks whether array A has at least one component of the array A is zero.

This is the complete code:

¹²³ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-7.html>

Listing 15: int_arrays.ads

```

1 package Int_Arrays is
2
3   type Integer_Arr is
4     array (Positive range <>) of Integer;
5
6   function Is_Zero (A : Integer_Arr)
7     return Boolean is
8     (for all I in A'Range => A (I) = 0);
9
10  function Has_Zero (A : Integer_Arr)
11    return Boolean is
12    (for some I in A'Range => A (I) = 0);
13
14  procedure Display_Array (A : Integer_Arr;
15                          Name : String);
16
17 end Int_Arrays;

```

Listing 16: int_arrays.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Int_Arrays is
4
5   procedure Display_Array (A : Integer_Arr;
6                           Name : String) is
7   begin
8     Put (Name & ": ");
9     for E of A loop
10      Put (E'Image & " ");
11    end loop;
12    New_Line;
13  end Display_Array;
14
15 end Int_Arrays;

```

Listing 17: test_int_arrays.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Int_Arrays; use Int_Arrays;
4
5 procedure Test_Int_Arrays is
6   A : Integer_Arr := (0, 0, 1);
7 begin
8   Display_Array (A, "A");
9   Put_Line ("Is_Zero: "
10            & Boolean'Image (Is_Zero (A)));
11   Put_Line ("Has_Zero: "
12            & Boolean'Image (Has_Zero (A)));
13
14   A := (0, 0, 0);
15
16   Display_Array (A, "A");
17   Put_Line ("Is_Zero: "
18            & Boolean'Image (Is_Zero (A)));
19   Put_Line ("Has_Zero: "
20            & Boolean'Image (Has_Zero (A)));
21 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

```
1 package Int_Arrays is  
2  
3   type Integer_Arr is  
4     array (Positive range <>) of Integer;  
5  
6   function Is_Zero (A : Integer_Arr)  
7     return Boolean;  
8  
9   function Has_Zero (A : Integer_Arr)  
10    return Boolean;  
11  
12   procedure Display_Array (A : Integer_Arr;  
13     Name : String);  
14  
15 end Int_Arrays;
```

Listing 19: int_arrays.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 package body Int_Arrays is  
4  
5   function Is_Zero (A : Integer_Arr)  
6     return Boolean is  
7   begin  
8     for I in A'Range loop  
9       if A (I) /= 0 then  
10        return False;  
11      end if;  
12    end loop;  
13  
14    return True;  
15  end Is_Zero;  
16  
17   function Has_Zero (A : Integer_Arr)  
18     return Boolean is  
19   begin  
20     for I in A'Range loop  
21       if A (I) = 0 then  
22        return True;  
23      end if;
```

(continues on next page)

(continued from previous page)

```

24     end loop;
25
26     return False;
27 end Has_Zero;
28
29 procedure Display_Array (A    : Integer_Arr;
30                          Name : String) is
31 begin
32     Put (Name & ": ");
33     for E of A loop
34         Put (E'Image & " ");
35     end loop;
36     New_Line;
37 end Display_Array;
38
39 end Int_Arrays;

```

Listing 20: test_int_arrays.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Int_Arrays; use Int_Arrays;
4
5  procedure Test_Int_Arrays is
6      A : Integer_Arr := (0, 0, 1);
7  begin
8      Display_Array (A, "A");
9      Put_Line ("Is_Zero: "
10              & Boolean'Image (Is_Zero (A)));
11     Put_Line ("Has_Zero: "
12              & Boolean'Image (Has_Zero (A)));
13
14     A := (0, 0, 0);
15
16     Display_Array (A, "A");
17     Put_Line ("Is_Zero: "
18             & Boolean'Image (Is_Zero (A)));
19     Put_Line ("Has_Zero: "
20             & Boolean'Image (Has_Zero (A)));
21 end Test_Int_Arrays;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Quantified_Expression.
↳ Quantified_Expression_2
MD5: a957a8fd60e1849248efela84eae6afa

```

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

```
1 package Int_Arrays is
2
3   type Integer_Arr is
4     array (Positive range <>) of Integer;
5
6   function Is_Zero (A : Integer_Arr)
7     return Boolean is
8     (for all E of A => E = 0);
9
10  function Has_Zero (A : Integer_Arr)
11    return Boolean is
12    (for some E of A => E = 0);
13
14 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 Expressions¹²⁴
-

8.4 Declare Expressions

So far, we've seen expressions that make use of existing objects declared outside of the expression. Sometimes, we might want to declare constant objects inside the expression, so we can use them locally in the expression. Similarly, we might want to rename an object and use the renamed object in an expression. In those cases, we can use a declare expression.

A declare expression allows for declaring or renaming objects within an expression:

Listing 22: p.ads

```
1 pragma Ada_2022;
2
3 package P is
4
5   function Max (A, B : Integer) return Integer is
6     (declare
7       Bigger_A : constant Boolean := (A >= B);
8     begin
9       (if Bigger_A then A else B));
10
11 end P;
```

Code block metadata

¹²⁴ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-8.html>

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Simple_Declare_Expression
 MD5: 5da80e76393645d6eb1cb8cfe88e190a

The declare expression starts with the **declare** keyword and the usual object declarations, and it's followed by the **begin** keyword and the body. In this example, the body of the declare expression is a conditional expression.

Of course, the code above isn't really useful, so let's look at a more complete example:

Listing 23: integer_arrays.ads

```

1 pragma Ada_2022;
2
3 package Integer_Arrays is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     function Sum (Arr : Integer_Array)
9         return Integer;
10
11     --
12     -- Expression function using
13     -- declare expression:
14     --
15     function Avg (Arr : Integer_Array)
16         return Float is
17         (declare
18             A : Integer_Array renames Arr;
19             S : constant Float := Float (Sum (A));
20             L : constant Float := Float (A'Length);
21         begin
22             S / L);
23
24 end Integer_Arrays;
```

Listing 24: integer_arrays.adb

```

1 package body Integer_Arrays is
2
3     function Sum (Arr : Integer_Array)
4         return Integer is
5     begin
6         return Acc : Integer := 0 do
7             for V of Arr loop
8                 Acc := Acc + V;
9             end loop;
10        end return;
11    end Sum;
12
13 end Integer_Arrays;
```

Listing 25: show_integer_arrays.adb

```

1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 with Integer_Arrays; use Integer_Arrays;
6
```

(continues on next page)

(continued from previous page)

```
7 procedure Show_Integer_Arrays is
8   Arr : constant Integer_Array := [1, 2, 3];
9 begin
10  Put_Line ("Sum: "
11           & Sum (Arr)'Image);
12  Put_Line ("Avg: "
13           & Avg (Arr)'Image);
14 end Show_Integer_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Integer_
Arrays
MD5: 8e96d49b1676f0aaf95437e271069690
```

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](#)¹²⁵

8.4.1 Restrictions in the declarative part

The declarative part of a declare expression is more restricted than the declarative part of a subprogram or declare block. In fact, we cannot:

- declare variables;
- declare constants of limited types;
- rename an object of limited type that is constructed within the declarative part;
- declare aliased constants;
- declare constants that make use of the **Access** or **Unchecked_Access** attributes in the initialization;
- declare constants of anonymous access type.

Let's see some examples of erroneous declarations:

Listing 26: integer_arrays.ads

```
1 pragma Ada_2022;
2
3 package Integer_Arrays is
4
5   type Integer_Array is
6     array (Positive range <>) of Integer;
7
8   type Integer_Sum is limited private;
```

(continues on next page)

¹²⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-9.html>

(continued from previous page)

```

9
10 type Const_Integer_Access is
11     access constant Integer;
12
13 function Sum (Arr : Integer_Array)
14     return Integer;
15
16 function Sum (Arr : Integer_Array)
17     return Integer_Sum;
18
19 --
20 -- Expression function using
21 -- declare expression:
22 --
23 function Avg (Arr : Integer_Array)
24     return Float is
25     (declare
26         A : Integer_Array renames Arr;
27
28         S1 : aliased constant Integer := Sum (A);
29         -- ERROR: aliased constant
30
31         S : Float := Float (S1);
32         L : Float := Float (A'Length);
33         -- ERROR: declaring variables
34
35         S2 : constant Integer_Sum := Sum (A);
36         -- ERROR: declaring constant of
37         -- limited type
38
39         A1 : Const_Integer_Access :=
40             S1'Unchecked_Access;
41         -- ERROR: using 'Unchecked_Access
42         -- attribute
43
44         A2 : access Integer := null;
45         -- ERROR: declaring object of
46         -- anonymous access type
47     begin
48         S / L);
49
50 private
51
52     type Integer_Sum is new Integer;
53
54 end Integer_Arrays;
```

Listing 27: integer_arrays.adb

```

1 package body Integer_Arrays is
2
3     function Sum (Arr : Integer_Array)
4         return Integer is
5     begin
6         return Acc : Integer := 0 do
7             for V of Arr loop
8                 Acc := Acc + V;
9             end loop;
10        end return;
11    end Sum;
12
```

(continues on next page)

(continued from previous page)

```
13   function Sum (Arr : Integer_Array)
14       return Integer_Sum is
15       (Integer_Sum (Integer'(Sum (Arr))));
16
17 end Integer_Arrays;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Integer_
↳Arrays_Error
MD5: e1f72f817baea87f66fb34b6aa8d1949
```

Build output

```
integer_arrays.ads:28:10: error: "aliased" not allowed in declare_expression
integer_arrays.ads:31:10: error: object renaming or constant declaration expected
integer_arrays.ads:32:10: error: object renaming or constant declaration expected
integer_arrays.ads:35:10: error: object renaming or constant declaration expected
integer_arrays.ads:40:19: error: "Unchecked_Access" attribute cannot occur in a_
↳declare_expression
integer_arrays.ads:44:15: error: anonymous access type not allowed in declare_
↳expression
gprbuild: *** compilation phase failed
```

In this version of the Avg function, we see many errors in the declarative part of the declare expression. If we convert the declare expression into an actual function implementation, however, those declarations won't trigger compilation errors. (Feel free to try this out!)

8.5 Reduction Expressions

Note: This feature was introduced in Ada 2022.

A reduction expression reduces a list of values into a single value. For example, we can reduce the list [2, 3, 4] to a single value:

- by adding the values of the list: $2 + 3 + 4 = 9$, or
- by multiplying the values of the list: $2 * 3 * 4 = 24$.

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 as the initial value, we get the reduction expression: $0 + 2 + 3 + 4 = 9$. This can be implemented using an array:

Listing 28: show_reduction_expression.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   A : array (1 .. 3) of Integer;
5   I : Integer;
6 begin
7   A := [2, 3, 4];
```

(continues on next page)

(continued from previous page)

```

8   I := A'Reduce ("+", 0);
9
10  Put_Line ("A = "
11           & A'Image);
12  Put_Line ("I = "
13           & I'Image);
14  end Show_Reduction_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↳ Simple_Reduction_Expression
MD5: 1a0164b3c4768125c8dbbe8a0f4955a1

Runtime output

```

A =
[ 2,  3,  4]
I =  9
```

Here, we have the array A with a list of values. The A'Reduce ("+", 0) expression reduces the list of values of A into a single value — in this case, an integer value that is stored in I. This statement is equivalent to:

```

I := 0;
for E of A loop
  I := I + E;
end loop;
```

Naturally, we can reduce the array using the * operator:

Listing 29: show_reduction_expression.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Reduction_Expression is
4     A : array (1 .. 3) of Integer;
5     I : Integer;
6  begin
7     A := [2, 3, 4];
8     I := A'Reduce ("*", 1);
9
10    Put_Line ("A = "
11            & A'Image);
12    Put_Line ("I = "
13            & I'Image);
14  end Show_Reduction_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↳ Simple_Reduction_Expression
MD5: 415b1ee8b21cca6d2438a34c88e7e2df

Runtime output

```

A =
[ 2,  3,  4]
I = 24
```

In this example, we call A'Reduce ("*", 1) to reduce the list. (Note that we use an

initial value of one because it is the [identity element](#)¹²⁶ of a multiplication, so the complete operation is: $1 * 2 * 3 * 4 = 24$.)

In the Ada Reference Manual

- [Reduction Expressions](#)¹²⁷
-

8.5.1 Value sequences

In addition to arrays, we can apply reduction expression to value sequences, which consist of an iterated element association — for example, `[for I in 1 .. 3 => I + 1]`. We can simply *append* the reduction expression to a value sequence:

Listing 30: show_reduction_expression.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   I : Integer;
5 begin
6   I := [for I in 1 .. 3 =>
7         I + 1]'Reduce ("+", 0);
8   Put_Line ("I = "
9             & I'Image);
10
11  I := [for I in 1 .. 3 =>
12        I + 1]'Reduce ("*", 1);
13  Put_Line ("I = "
14          & I'Image);
15 end Show_Reduction_Expression;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↳Reduction_Expression_Value_Sequences
MD5: e714f69700e3f0387314ee0e531620c4
```

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.)

¹²⁶ https://en.wikipedia.org/wiki/Identity_element

¹²⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-10.html>

8.5.2 Custom reducers

In the previous examples, we've used standard operators such as `+` and `*` as the reducer. We can, however, write our own reducers and pass them to the `Reduce` attribute. For example:

Listing 31: `show_reduction_expression.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   type Integer_Array is
5     array (Positive range <>) of Integer;
6
7   A : Integer_Array (1 .. 3);
8   I : Long_Integer;
9
10  procedure Accumulate
11    (Accumulator : in out Long_Integer;
12     Value       : Integer) is
13  begin
14    Accumulator := Accumulator
15                  + Long_Integer (Value);
16  end Accumulate;
17
18 begin
19   A := [2, 3, 4];
20   I := A'Reduce (Accumulate, 0);
21
22   Put_Line ("A = "
23            & A'Image);
24   Put_Line ("I = "
25            & I'Image);
26 end Show_Reduction_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
 ↪ Custom_Reducer_Procedure
 MD5: 3190a1ff6a8027268ca96a75cf214714

Runtime output

```

A =
[ 2,  3,  4]
I = 9
```

In this example, we implement the `Accumulate` procedure as our reducer, which is called to accumulate the individual elements (integer values) of the list. We pass this procedure to the `Reduce` attribute in the `I := A'Reduce (Accumulate, 0)` statement, which is equivalent to:

```

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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   type Integer_Array is
5     array (Positive range <>) of Integer;
6
7   A : Integer_Array (1 .. 3);
8   I : Long_Integer;
9
10  function Accumulate
11    (Accumulator : Long_Integer;
12     Value       : Integer)
13    return Long_Integer is
14  begin
15    return Accumulator + Long_Integer (Value);
16  end Accumulate;
17
18 begin
19   A := [2, 3, 4];
20   I := A'Reduce (Accumulate, 0);
21
22   Put_Line ("A = "
23             & A'Image);
24   Put_Line ("I = "
25             & I'Image);
26 end Show_Reduction_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↳Custom_Reducer_Function
MD5: ee5d5bb2b151ef7552d752c7e452127d

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
```

(continues on next page)

(continued from previous page)

```

I := Accumulate (I, E);
end loop;

```

8.5.3 Other accumulator types

The accumulator type isn't restricted to scalars: in fact, we could use record types as well. For example:

Listing 33: show_reduction_expression.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   type Integer_Array is
5     array (Positive range <>) of Integer;
6
7   A : Integer_Array (1 .. 3);
8
9   type Integer_Accumulator is record
10    Value : Long_Integer;
11    Count : Integer;
12  end record;
13
14  function Accumulate
15    (Accumulator : Integer_Accumulator;
16     Value       : Integer)
17    return Integer_Accumulator is
18  begin
19    return (Value => Accumulator.Value
20           + Long_Integer (Value),
21           Count => Accumulator.Count + 1);
22  end Accumulate;
23
24  function Zero return Integer_Accumulator is
25    (Value => 0, Count => 0);
26
27  function Average (Acc : Integer_Accumulator)
28    return Float is
29    (Float (Acc.Value) / Float (Acc.Count));
30
31  Acc : Integer_Accumulator;
32
33  begin
34    A := [2, 3, 4];
35
36    Acc := A'Reduce (Accumulate, Zero);
37    Put_Line ("Acc = "
38             & Acc'Image);
39    Put_Line ("Avg = "
40             & Average (Acc)'Image);
41  end Show_Reduction_Expression;

```

In this example, we're using the `Integer_Accumulator` record type in our reducer — the `Accumulate` function. In this case, we're not only accumulating the values, but also counting the number of elements in the list. (Of course, we could have used `A'Length` for that as well.)

Also, we're not limited to numeric types: we can also create a reducer using strings as the accumulator type. In fact, we can display the initial value and the elements of the list by using unbounded strings:

Listing 34: show_reduction_expression.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.Unbounded;
4 use Ada.Strings.Unbounded;
5
6 procedure Show_Reduction_Expression is
7   type Integer_Array is
8     array (Positive range <>) of Integer;
9
10  A : Integer_Array (1 .. 3);
11
12  function Unbounded_String_List
13    (Accumulator : Unbounded_String;
14     Value       : Integer)
15    return Unbounded_String is
16  begin
17    return Accumulator
18      & ", " & Value'Image;
19  end Unbounded_String_List;
20
21 begin
22  A := [2, 3, 4];
23
24  Put_Line ("A = "
25           & A'Image);
26  Put_Line ("L = "
27           & To_String (A'Reduce
28                       (Unbounded_String_List,
29                        To_Unbounded_String ("0"))));
30 end Show_Reduction_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↳ Reducer_String_Accumulator
MD5: 43c54e93e404a235c8721db7c691a864

Runtime output

```
A =
[ 2,  3,  4]
L = 0,  2,  3,  4
```

In this case, the "accumulator" is concatenating the initial value and individual values of the list into a string.

STATEMENTS

9.1 Simple and Compound Statements

We can classify statements as either simple or compound. Simple statements don't contain other statements; think of them as "atomic units" that cannot be further divided. Compound statements, on the other hand, may contain other — simple or compound — statements.

Here are some examples from each category:

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](#)¹²⁸
-

9.2 Labels

We can use labels to identify statements in the code. They have the following format: `<<Some_Label>>`. We write them right before the statement we want to apply it to. Let's see an example of labels with simple statements:

Listing 1: show_statement_identifier.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Statement_Identifier is
4   pragma Warnings (Off, "is not referenced");
5 begin
6   <<Show_Hello>> Put_Line ("Hello World!");
7   <<Show_Test>> Put_Line ("This is a test.");
8
9   <<Show_Separator>>
10  <<Show_Block_Separator>>
11  Put_Line ("=====");
12 end Show_Statement_Identifier;
```

Code block metadata

¹²⁸ <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](#)¹²⁹

9.2.1 Labels and goto statements

Labels are mainly used in combination with **goto** statements. (Although pretty much uncommon, we could potentially use labels to indicate important statements in the code.) Let's see an example where we use a **goto** label; statement to *jump* to a specific label:

Listing 2: show_cleanup.adb

```
1 procedure Show_Cleanup is
2   pragma Warnings (Off, "always false");
3
4   Some_Error : Boolean;
5 begin
6   Some_Error := False;
7
8   if Some_Error then
9     goto Cleanup;
10  end if;
11
12  <<Cleanup>> null;
13 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.

¹²⁹ <http://www.ada-auth.org/standards/22rm/html/RM-5-1.html>

9.2.2 Use-case: Continue

Another use-case is that of a Continue label in a loop. Consider a loop where we want to skip further processing depending on a condition:

Listing 3: show_continue.adb

```

1 procedure Show_Continue is
2   function Is_Further_Processing_Needed
3     (Dummy : Integer)
4     return Boolean
5   is
6   begin
7     -- Dummy implementation
8     return False;
9   end Is_Further_Processing_Needed;
10
11   A : constant array (1 .. 10) of Integer :=
12     (others => 0);
13 begin
14   for E of A loop
15
16     -- Some stuff here...
17
18     if Is_Further_Processing_Needed (E) then
19
20       -- Do more stuff...
21
22       null;
23     end if;
24   end loop;
25 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

```

1 procedure Show_Continue is
2   function Is_Further_Processing_Needed
3     (Dummy : Integer)
4     return Boolean
5   is
6   begin
7     -- Dummy implementation
8     return False;
9   end Is_Further_Processing_Needed;
10
11   A : constant array (1 .. 10) of Integer :=
12     (others => 0);
13 begin
14   for E of A loop
15
16     -- Some stuff here...
17
```

(continues on next page)

(continued from previous page)

```
18     if not Is_Further_Processing_Needed (E) then
19         goto Continue;
20     end if;
21
22     -- Do more stuff...
23
24     <<Continue>>
25     end loop;
26 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;
```

9.2.3 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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Statement_Identifier is
4     pragma Warnings (Off, "is not referenced");
5
6     Arr  : constant array (1 .. 5) of Integer :=
7         (1, 4, 6, 42, 49);
8     Found : Boolean := False;
9 begin
10    <<Find_42>> for E of Arr loop
```

(continues on next page)

(continued from previous page)

```

11     if E = 42 then
12         Found := True;
13         exit;
14     end if;
15 end loop;
16
17 Put_Line ("Found: " & Found'Image);
18 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

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Statement_Identifier is
4     Arr : constant array (1 .. 5) of Integer :=
5         (1, 4, 6, 42, 49);
6     Found : Boolean := False;
7 begin
8     Find_42 : for E of Arr loop
9         if E = 42 then
10            Found := True;
11            exit Find_42;
12        end if;
13    end loop Find_42;
14
15    Put_Line ("Found: " & Found'Image);
16 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.

9.3 Exit loop statement

We've introduced bare loops back in the [Introduction to Ada course](#)¹³⁰. In this section, we'll briefly discuss loop names and exit loop statements.

A bare loop has this form:

```
loop
  exit when Some_Condition;
end loop;
```

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

```
Loop_Name:
loop
  exit Loop_Name when Some_Condition;
end loop Loop_Name;
```

In this case, we have to use the loop's name after **end loop**. Also, having a name for a loop allows us to indicate which loop we're exiting from: **exit Loop_Name when**.

Let's see a complete example:

Listing 7: show_vector_cursor_iteration.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Containers.Vectors;
3
4 procedure Show_Vector_Cursor_Iteration is
5
6   package Integer_Vectors is new
7     Ada.Containers.Vectors
8     (Index_Type => Positive,
9      Element_Type => Integer);
10
11  use Integer_Vectors;
12
13  V : constant Vector := 20 & 10 & 0 & 13;
14  C : Cursor;
15 begin
16  C := V.First;
17  Put_Line ("Vector elements are: ");
18
19  Show_Elements :
20  loop
21    exit Show_Elements when C = No_Element;
22
23    Put_Line ("Element: "
24             & Integer'Image (V (C)));
25    C := Next (C);
26  end loop Show_Elements;
27
28 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

¹³⁰ https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-bare-loops

```
Vector elements are:  
Element: 20  
Element: 10  
Element: 0  
Element: 13
```

Naming a loop is particularly useful when we have nested loops and we want to exit directly from the inner loop:

Listing 8: show_inner_loop_exit.adb

```
1 procedure Show_Inner_Loop_Exit is  
2   pragma Warnings (Off);  
3  
4   Cond : Boolean := True;  
5 begin  
6  
7   Outer_Processing : loop  
8  
9     Inner_Processing : loop  
10      exit Outer_Processing when Cond;  
11      end loop Inner_Processing;  
12  
13   end loop Outer_Processing;  
14  
15 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](#)¹³¹

9.4 If, case and loop statements

In the Introduction to Ada course, we talked about [if statements](#)¹³², [loop statements](#)¹³³, and [case statements](#)¹³⁴. This is a very simple code example with these statements:

Listing 9: show_if_case_loop_statements.adb

```
1 procedure Show_If_Case_Loop_Statements is  
2   pragma Warnings (Off);  
3  
4   Reset      : Boolean := False;
```

(continues on next page)

¹³¹ <http://www.ada-auth.org/standards/22rm/html/RM-5-7.html>

¹³² https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-if-statement

¹³³ https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-loop-statement

¹³⁴ https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-case-statement

(continued from previous page)

```
5   Increment : Boolean := True;
6   Val       : Integer := 0;
7   begin
8       --
9       -- If statement
10      --
11      if Reset then
12          Val := 0;
13      elsif Increment then
14          Val := Val + 1;
15      else
16          Val := Val - 1;
17      end if;
18
19      --
20      -- Loop statement
21      --
22      for I in 1 .. 5 loop
23          Val := Val * 2 - I;
24      end loop;
25
26      --
27      -- Case statement
28      --
29      case Val is
30          when 0 .. 5 =>
31              null;
32          when others =>
33              Val := 5;
34      end case;
35
36  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 Statements¹³⁵](#)
- [5.4 Case Statements¹³⁶](#)
- [5.5 Loop Statements¹³⁷](#)

¹³⁵ <http://www.ada-auth.org/standards/22rm/html/RM-5-3.html>

¹³⁶ <http://www.ada-auth.org/standards/22rm/html/RM-5-4.html>

¹³⁷ <http://www.ada-auth.org/standards/22rm/html/RM-5-5.html>

9.4.1 Case statements and expressions

As we know, the case statement has a choice expression (**case** Choice_Expression **is**), which is expected to be a discrete type. Also, this expression can be a function call or a type conversion, for example — in addition to being a variable or a constant.

As we discussed *earlier on* (page 323), if we use parentheses, the contents between those parentheses is parsed as an expression. In the context of case statements, the expression is first evaluated before being used as a choice expression. Consider the following code example:

Listing 10: scales.ads

```

1 package Scales is
2
3     type Satisfaction_Scale is (Very_Dissatisfied,
4                               Dissatisfied,
5                               OK,
6                               Satisfied,
7                               Very_Satisfied);
8
9     type Scale is range 0 .. 10;
10
11    function To_Satisfaction_Scale
12        (S : Scale)
13        return Satisfaction_Scale;
14
15 end Scales;
```

Listing 11: scales.adb

```

1 package body Scales is
2
3     function To_Satisfaction_Scale
4         (S : Scale)
5         return Satisfaction_Scale
6     is
7         Satisfaction : Satisfaction_Scale;
8     begin
9         case (S) is
10            when 0 .. 2 =>
11                Satisfaction := Very_Dissatisfied;
12            when 3 .. 4 =>
13                Satisfaction := Dissatisfied;
14            when 5 .. 6 =>
15                Satisfaction := OK;
16            when 7 .. 8 =>
17                Satisfaction := Satisfied;
18            when 9 .. 10 =>
19                Satisfaction := Very_Satisfied;
20        end case;
21
22        return Satisfaction;
23    end To_Satisfaction_Scale;
24
25 end Scales;
```

Listing 12: show_case_statement_expression.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
```

(continues on next page)

(continued from previous page)

```
3 with Scales;      use Scales;
4
5 procedure Show_Case_Statement_Expression is
6   Score : constant Scale := 0;
7 begin
8   Put_Line ("Score: "
9             & Scale'Image (Score)
10            & Satisfaction_Scale'Image (
11            To_Satisfaction_Scale (Score)));
12
13 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](#)¹³⁸, every possible value for the choice expression needs to be covered by a unique branch of the case statement. In principle, it *seems* that we're actually covering all possible values of the `Scale` type, which ranges from 0 to 10. However, we've written `case (S) 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 11) `Scale'Base'Range`.

In other languages

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

Listing 13: main.c

```
1
2 #include <stdio.h>
3
4 int main(int argc, const char * argv[])
5 {
6     int s = 0;
7
8     switch (s)
9     {
10        case 0:
11        case 1:
12            printf("Value in the 0 -- 1 range\n");
13        default:
14            printf("Value > 1\n");
15    }
16 }
```

Code block metadata

¹³⁸ https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-case-statement

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.

9.5 Block Statements

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

Listing 14: show_block_statement.adb

```
1 procedure Show_Block_Statement is  
2   pragma Warnings (Off);  
3   begin  
4  
5     -- BLOCK STARTS HERE:  
6     declare  
7       I : Integer;  
8     begin  
9       I := 0;  
10    end;  
11  
12 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

```
1 procedure Show_Block_Statement is  
2   pragma Warnings (Off);  
3   begin  
4  
5     Simple_Block : declare  
6       I : Integer;  
7     begin  
8       I := 0;  
9     end Simple_Block;
```

(continues on next page)

¹³⁹ https://learn.adacore.com/courses/intro-to-ada/chapters/imperative_language.html#intro-ada-block-statement

(continued from previous page)

```
10
11 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](#)¹⁴⁰
-

9.6 Extended return statement

A common idiom in Ada is to build up a function result in a local object, and then return that object:

Listing 16: show_return.adb

```
1 procedure Show_Return is
2
3     type Array_Of_Natural is
4         array (Positive range <>) of Natural;
5
6     function Sum (A : Array_Of_Natural)
7         return Natural
8     is
9         Result : Natural := 0;
10    begin
11        for Index in A'Range loop
12            Result := Result + A (Index);
13        end loop;
14        return Result;
15    end Sum;
16
17 begin
18     null;
19 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:

¹⁴⁰ <http://www.ada-auth.org/standards/22rm/html/RM-5-6.html>

Listing 17: show_extended_return.adb

```

1  procedure Show_Extended_Return is
2
3      type Array_Of_Natural is
4          array (Positive range <>) of Natural;
5
6      function Sum (A : Array_Of_Natural)
7          return Natural
8      is
9      begin
10         return Result : Natural := 0 do
11             for Index in A'Range loop
12                 Result := Result + A (Index);
13             end loop;
14         end return;
15     end Sum;
16
17 begin
18     null;
19 end Show_Extended_Return;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Extended_Return_Statements.
↳ Extended_Return
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](#)¹⁴¹

9.6.1 Other usages of extended return statements

Note: This section was originally written by Robert A. Duff and published as [Gem #10: Limited Types in Ada 2005](#)¹⁴².

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

Listing 18: show_string_construct.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_String_Construct is
4
5      function Make_String
6          (S           : String;
```

(continues on next page)

¹⁴¹ <http://www.ada-auth.org/standards/22rm/html/RM-6-5.html>¹⁴² <https://www.adacore.com/gems/ada-gem-10>

(continued from previous page)

```
7     Prefix      : String;
8     Use_Prefix  : Boolean) return String
9  is
10    Length : Natural := S'Length;
11  begin
12    if Use_Prefix then
13      Length := Length + Prefix'Length;
14    end if;
15
16    return Result : String (1 .. Length) do
17
18      -- fill in the characters
19      if Use_Prefix then
20        Result
21          (1 .. Prefix'Length) := Prefix;
22
23        Result
24          (Prefix'Length + 1 .. Length) := S;
25      else
26        Result := S;
27      end if;
28
29    end return;
30  end Make_String;
31
32  S1 : String := "Ada";
33  S2 : String := "Make_With_";
34  begin
35    Put_Line ("No prefix:  "
36      & Make_String (S1, S2, False));
37    Put_Line ("With prefix: "
38      & Make_String (S1, S2, True));
39  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.

SUBPROGRAMS

10.1 Parameter Modes and Associations

In this section, we discuss some details about parameter modes and associations. First of all, as we know, parameters can be either formal or actual:

- Formal parameters are the ones we see in a subprogram declaration and implementation;
- Actual parameters are the ones we see in a subprogram call.
 - Note that actual parameters are also called *subprogram arguments* in other languages.

We define parameter associations as the connection between an actual parameter in a subprogram call and its declaration as a formal parameter in a subprogram specification or body.

In the Ada Reference Manual

- [6.2 Formal Parameter Modes](#)¹⁴³
 - [6.4.1 Parameter Associations](#)¹⁴⁴
-

10.1.1 Formal Parameter Modes

We already discussed formal parameter modes in the [Introduction to Ada](#)¹⁴⁵ course:

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.

¹⁴³ <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

¹⁴⁴ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

¹⁴⁵ <https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#intro-ada-parameter-modes>

10.1.2 By-copy and by-reference

In the [Introduction to Ada](#)¹⁴⁶ course, we saw that parameter modes don't correspond directly to how parameters are actually passed. In fact, an **in out** parameter could be passed by copy. For example:

Listing 1: check_param_passing.ads

```
1 with System;
2
3 procedure Check_Param_Passing
4   (Formal : System.Address;
5    Actual : System.Address);
```

Listing 2: check_param_passing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System.Address_Image;
3
4 procedure Check_Param_Passing
5   (Formal : System.Address;
6    Actual : System.Address) is
7 begin
8   Put_Line ("Formal parameter at "
9             & System.Address_Image (Formal));
10  Put_Line ("Actual parameter at "
11           & System.Address_Image (Actual));
12  if System.Address_Image (Formal) =
13     System.Address_Image (Actual)
14  then
15    Put_Line
16      ("Parameter is passed by reference.");
17  else
18    Put_Line
19      ("Parameter is passed by copy.");
20  end if;
21 end Check_Param_Passing;
```

Listing 3: machine_x.ads

```
1 with System;
2
3 package Machine_X is
4
5   procedure Update_Value
6     (V : in out Integer;
7      AV : System.Address);
8
9 end Machine_X;
```

Listing 4: machine_x.adb

```
1 with Check_Param_Passing;
2
3 package body Machine_X is
4
5   procedure Update_Value
6     (V : in out Integer;
7      AV : System.Address) is
```

(continues on next page)

¹⁴⁶ <https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#intro-ada-parameter-modes>

(continued from previous page)

```

8   begin
9       V := V + 1;
10      Check_Param_Passing (Formal => V'Address,
11                          Actual => AV);
12  end Update_Value;
13
14  end Machine_X;

```

Listing 5: show_by_copy_by_ref_params.adb

```

1  with Machine_X; use Machine_X;
2
3  procedure Show_By_Copy_By_Ref_Params is
4      A : Integer := 5;
5  begin
6      Update_Value (A, A'Address);
7  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 00007FFFA43D791C
Actual parameter at 00007FFFA43D793C
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 132)).

As we know, when a parameter is passed by copy, it is first copied to a temporary object. In the case of a parameter with **in out** mode, the temporary object is copied back to the original (actual) parameter at the end of the subprogram call. In our example, the temporary object indicated by V is copied back to A at the end of the call to Update_Value.

In Ada, it's not the parameter mode that determines whether a parameter is passed by copy or by reference, but rather its type. We can distinguish between three categories:

1. By-copy types;
2. By-reference types;
3. *Unspecified* types.

Obviously, parameters of by-copy types are passed by copy and parameters of by-reference type are passed by reference. However, if a category isn't specified — i.e. when the type is neither a by-copy nor a by-reference type —, the decision is essentially left to the compiler.

As a rule of thumb, we can say that;

- elementary types — and any type that is essentially elementary, such as a private type whose full view is an elementary type — are passed by copy;

- 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	<ul style="list-style-type: none">• Elementary types• Descendant of a private type whose full type is a by-copy type
By reference	By reference	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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 716).

Let's see an example:

Listing 6: machine_x.ads

```
1 with System;
2
3 package Machine_X is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     type Rec is record
9         A : Integer;
10    end record;
11
12    type Rec_Array is record
13        A : Integer;
14        Arr : Integer_Array (1 .. 100);
15    end record;
16
17    type Tagged_Rec is tagged record
18        A : Integer;
```

(continues on next page)

(continued from previous page)

```

19  end record;
20
21  procedure Update_Value
22    (R : in out Rec;
23     AR : System.Address);
24
25  procedure Update_Value
26    (RA : in out Rec_Array;
27     ARA : System.Address);
28
29  procedure Update_Value
30    (R : in out Tagged_Rec;
31     AR : System.Address);
32
33  end Machine_X;

```

Listing 7: machine_x.adb

```

1  with Check_Param_Passing;
2
3  package body Machine_X is
4
5    procedure Update_Value
6      (R : in out Rec;
7       AR : System.Address)
8    is
9    begin
10     R.A := R.A + 1;
11     Check_Param_Passing (Formal => R'Address,
12                        Actual => AR);
13   end Update_Value;
14
15   procedure Update_Value
16     (RA : in out Rec_Array;
17      ARA : System.Address)
18   is
19   begin
20     RA.A := RA.A + 1;
21     Check_Param_Passing (Formal => RA'Address,
22                        Actual => ARA);
23   end Update_Value;
24
25   procedure Update_Value
26     (R : in out Tagged_Rec;
27      AR : System.Address)
28   is
29   begin
30     R.A := R.A + 1;
31     Check_Param_Passing (Formal => R'Address,
32                        Actual => AR);
33   end Update_Value;
34
35  end Machine_X;

```

Listing 8: show_by_copy_by_ref_params.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Machine_X;   use Machine_X;
3
4  procedure Show_By_Copy_By_Ref_Params is

```

(continues on next page)

(continued from previous page)

```
5   TR : Tagged_Rec := (A => 5);
6   R  : Rec       := (A => 5);
7   RA : Rec_Array := (A => 5,
8                       Arr => (others => 0));
9   begin
10  Put_Line ("Tagged record");
11  Update_Value (TR, TR'Address);
12
13  Put_Line ("Untagged record");
14  Update_Value (R, R'Address);
15
16  Put_Line ("Untagged record with array");
17  Update_Value (RA, RA'Address);
18 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 00007FFEE3FB1200
Actual parameter at 00007FFEE3FB1200
Parameter is passed by reference.
Untagged record
Formal parameter at 00007FFEE3FB104C
Actual parameter at 00007FFEE3FB11FC
Parameter is passed by copy.
Untagged record with array
Formal parameter at 00007FFEE3FB1060
Actual parameter at 00007FFEE3FB1060
Parameter is passed by reference.
```

When we run this example, we see that the object of tagged type (Tagged_Rec) is passed by reference to the Update_Value procedure. In the case of the objects of untagged record types, you might see this:

- the parameter of Rec type — which is an untagged record with a single component of integer type —, the parameter is passed by copy;
- the parameter of Rec_Array type — which is an untagged record with a large array of 100 components —, the parameter is passed by reference.

Because Rec and Rec_Array are neither by-copy nor by-reference types, the decision about how to pass them to the Update_Value procedure is made by the compiler. (Thus, it is possible that you see different results when running the code above.)

10.1.3 Bounded errors

When we use parameters of types that are neither by-copy nor by-reference types, we might encounter the situation where we have the same object bound to different names in a subprogram. For example, if:

- we use a global object `Global_R` of a record type `Rec`

and

- we have a subprogram with an in-out parameter of the same record type `Rec`

and

- we pass `Global_R` as the actual parameter for the in-out parameter of this subprogram,

then we have two access paths to this object: one of them using the global variable directly, and the other one using it indirectly via the in-out parameter. This situation could lead to undefined behavior or to a program error. Consider the following code example:

Listing 9: machine_x.ads

```

1  with System;
2
3  package Machine_X is
4
5     type Rec is record
6         A : Integer;
7     end record;
8
9     Global_R : Rec := (A => 0);
10
11    procedure Update_Value
12        (R : in out Rec;
13         AR : System.Address);
14
15 end Machine_X;
```

Listing 10: machine_x.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2
3  with Check_Param_Passing;
4
5  package body Machine_X is
6
7     procedure Update_Value
8         (R : in out Rec;
9          AR : System.Address)
10    is
11        procedure Show_Vars is
12            begin
13                Put_Line ("Global_R.A: "
14                          & Integer'Image (Global_R.A));
15                Put_Line ("R.A: "
16                          & Integer'Image (R.A));
17            end Show_Vars;
18        begin
19            Check_Param_Passing (Formal => R'Address,
20                               Actual => AR);
21
22            Put_Line ("Incrementing Global_R.A...");
```

(continues on next page)

(continued from previous page)

```
23     Global_R.A := Global_R.A + 1;
24     Show_Vars;
25
26     Put_Line ("Incrementing R.A...");
27     R.A := R.A + 5;
28     Show_Vars;
29     end Update_Value;
30
31 end Machine_X;
```

Listing 11: show_by_copy_by_ref_params.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Machine_X;   use Machine_X;
3
4  procedure Show_By_Copy_By_Ref_Params is
5  begin
6     Put_Line ("Calling Update_Value...");
7     Update_Value (Global_R, Global_R'Address);
8
9     Put_Line ("After call to Update_Value...");
10    Put_Line ("Global_R.A: "
11              & Integer'Image (Global_R.A));
12 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 00007FFCE420962C
Actual parameter at 00000000008003BC
Parameter is passed by copy.
Incrementing Global_R.A...
Global_R.A: 1
R.A:      0
Incrementing R.A...
Global_R.A: 1
R.A:      5
After call to Update_Value...
Global_R.A: 5
```

In the `Update_Value` procedure, because `Global_R` and `R` have a type that is neither a by-pass nor a by-reference type, the language does not specify whether the old or the new value would be read in the calls to `Put_Line`. In other words, the actual behavior is undefined. Also, this situation might raise the `Program_Error` exception.

Important

As a general advice:

- you should be very careful when using global variables and
 - you should avoid passing them as parameters in situations such as the one illustrated in the code example above.
-

10.1.4 Aliased parameters

When a parameter is specified as *aliased*, it is always passed by reference, independently of the type we're using. In this sense, we can use this keyword to circumvent the rules mentioned so far. (We discuss more about *aliasing* (page 523) and *aliased parameters* (page 532) later on.)

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

Listing 12: machine_x.ads

```

1 with System;
2
3 package Machine_X is
4
5     procedure Update_Value
6         (V : aliased in out Integer;
7          AV : System.Address);
8
9 end Machine_X;
```

Listing 13: machine_x.adb

```

1 with Check_Param_Passing;
2
3 package body Machine_X is
4
5     procedure Update_Value
6         (V : aliased in out Integer;
7          AV : System.Address)
8     is
9     begin
10        V := V + 1;
11        Check_Param_Passing (Formal => V'Address,
12                             Actual => AV);
13    end Update_Value;
14
15 end Machine_X;
```

Listing 14: show_by_copy_by_ref_params.adb

```

1 with Machine_X; use Machine_X;
2
3 procedure Show_By_Copy_By_Ref_Params is
4     A : aliased Integer := 5;
5 begin
6     Update_Value (A, A'Address);
7 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 00007FFFF01B01CC
Actual parameter at 00007FFFF01B01CC
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

```
1 with Machine_X; use Machine_X;
2
3 procedure Show_By_Copy_By_Ref_Params is
4   A : Integer := 5;
5 begin
6   Update_Value (A, A'Address);
7 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 532) and *aliased objects* (page 525) later on in the context of access types.

10.1.5 Parameter Associations

When actual parameters are associated with formal parameters, some rules are checked. As a typical example, the type of each actual parameter must match the type of the corresponding formal 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](#)¹⁴⁷
-

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

```
1 package Operations is
2
3   procedure Add (Left  : in out Integer;
4                 Right :          Float := 1.0);
5
6 end Operations;
```

¹⁴⁷ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

Listing 17: operations.adb

```

1 package body Operations is
2
3   procedure Add (Left  : in out Integer;
4                 Right :          Float := 1.0) is
5
6     begin
7       Left := Left + Integer (Right);
8     end Add;
9 end Operations;
```

Listing 18: show_param_association.adb

```

1 with Operations; use Operations;
2
3 procedure Show_Param_Association is
4   A : Integer := 5;
5 begin
6   -- Positional association
7   Add (A, 2.0);
8
9   -- Positional association
10  -- (using default value)
11  Add (A);
12
13  -- Named association
14  Add (Left => A,
15       Right => 2.0);
16
17  -- Named association (inversed order)
18  Add (Right => 2.0,
19       Left  => A);
20
21  -- Mixed positional / named association
22  Add (A, Right => 2.0);
23 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

```
1 package Operations is
2
3   procedure Add (Left : in out Integer);
4
5   procedure Add (Left : in out Integer;
6                 Right : Float := 1.0);
7
8 end Operations;
```

Listing 20: operations.adb

```
1 package body Operations is
2
3   procedure Add (Left : in out Integer) is
4   begin
5     Left := Left + 1;
6   end Add;
7
8   procedure Add (Left : in out Integer;
9                 Right : Float := 1.0) is
10  begin
11    Left := Left + Integer (Right);
12  end Add;
13
14 end Operations;
```

Listing 21: show_param_association.adb

```
1 with Operations; use Operations;
2
3 procedure Show_Param_Association is
4   A : Integer := 5;
5 begin
6   Add (A);
7   -- ERROR: cannot decide which
8   --           procedure to take
9 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

```
1 package Machine_X is
2
3     procedure Update_Value (V1 : in out Integer;
4                             V2 : out Integer);
5
6 end Machine_X;
```

Listing 23: machine_x.adb

```
1 package body Machine_X is
2
3     procedure Update_Value (V1 : in out Integer;
4                             V2 : out Integer) is
5     begin
6         V1 := V1 + 1;
7         V2 := V2 + 1;
8     end Update_Value;
9
10 end Machine_X;
```

Listing 24: show_by_copy_by_ref_params.adb

```
1 with Machine_X; use Machine_X;
2
3 procedure Show_By_Copy_By_Ref_Params is
4     A : Integer := 5;
5 begin
6     Update_Value (A, A);
7 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.

10.2 Operators

Operators are commonly used for variables of scalar types such as **Integer** and **Float**. In these cases, they replace *usual* function calls. (To be more precise, operators are function calls, but written in a different format.) For example, we simply write `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	<code>=, /=, <, <=, >, >=</code>
Unary adding	<code>+, -</code>
Binary adding	<code>+, -, &</code>
Multiplying	<code>*, /, mod, rem</code>
Highest precedence	<code>**</code> , abs, not

In the Ada Reference Manual

- [4.5 Operators and Expression Evaluation](#)¹⁴⁸
-

10.2.1 User-defined operators

For non-scalar types, not all operators are defined. For example, it wouldn't make sense to expect a compiler to include an addition operator for a record type with multiple components. Exceptions to this rule are the equality and inequality operators (`=` and `/=`), which are defined for any type (be it scalar, record types, and array types).

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

¹⁴⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-5.html>

Listing 25: integer_arrays.ads

```

1 package Integer_Arrays is
2
3     type Integer_Array is
4       array (Positive range <>) of Integer;
5
6 end Integer_Arrays;
```

Listing 26: show_array_concatenation.adb

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Integer_Arrays; use Integer_Arrays;
3
4 procedure Show_Array_Concatenation is
5   A, B : Integer_Array (1 .. 5);
6   R    : Integer_Array (1 .. 10);
7 begin
8   A := (1 & 2 & 3 & 4 & 5);
9   B := (6 & 7 & 8 & 9 & 10);
10  R := A & B;
11
12  for E of R loop
13    Put (E'Image & ' ');
14  end loop;
15  New_Line;
16 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

```

1 package Integer_Arrays is
2
3     type Integer_Array is
4       array (Positive range <>) of Integer;
5
6 end Integer_Arrays;
```

Listing 28: show_array_addition.adb

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Integer_Arrays; use Integer_Arrays;
3
4 procedure Show_Array_Addition is
5   A, B, R : Integer_Array (1 .. 5);
6 begin
7   A := (1 & 2 & 3 & 4 & 5);
```

(continues on next page)

(continued from previous page)

```
8   B := (6 & 7 & 8 & 9 & 10);
9   R := A + B;
10
11  for E of R loop
12      Put (E'Image & ' ');
13  end loop;
14  New_Line;
15
16 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

```
1 package Integer_Arrays is
2
3   type Integer_Array is
4       array (Positive range <>) of Integer;
5
6   function "+" (Left, Right : Integer_Array)
7       return Integer_Array
8   with Post =>
9       (for all I in "+"'Result'Range =>
10          "+"'Result (I) = Left (I) + Right (I));
11
12 end Integer_Arrays;
```

Listing 30: integer_arrays.adb

```

1 package body Integer_Arrays is
2
3     function "+" (Left, Right : Integer_Array)
4         return Integer_Array
5
6     is
7         R : Integer_Array (Left'Range);
8     begin
9         for I in Left'Range loop
10            R (I) := Left (I) + Right (I);
11        end loop;
12
13        return R;
14    end "+";
15 end Integer_Arrays;

```

Listing 31: show_array_addition.adb

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Integer_Arrays; use Integer_Arrays;
3
4 procedure Show_Array_Addition is
5     A, B, R : Integer_Array (1 .. 5);
6 begin
7     A := (1 & 2 & 3 & 4 & 5);
8     B := (6 & 7 & 8 & 9 & 10);
9     R := A + B;
10
11     for E of R loop
12         Put (E'Image & ' ');
13     end loop;
14     New_Line;
15
16 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 328).)

Note that, in this implementation, we assume that the range of `Right` is a subset of the range of `Left`. If that is not the case, the `Constraint_Error` exception will be raised at runtime in the loop. (You can test this by declaring `B` as `Integer_Array (5 .. 10)`, for example.)

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

```
1 package Addresses is
2
3   type Person is private;
4
5   function "+" (Name    : String;
6                 Address : String)
7                 return Person;
8   function "+" (Left, Right : Person)
9                 return Person;
10
11  procedure Display (P : Person);
12
13 private
14
15  subtype Name_String  is String (1 .. 40);
16  subtype Address_String is String (1 .. 100);
17
18  type Person is record
19    Name    : Name_String;
20    Address : Address_String;
21  end record;
22
23 end Addresses;
```

Listing 33: addresses.adb

```
1 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 package body Addresses is
5
6   function "+" (Name    : String;
7                 Address : String)
8                 return Person
9
10  is
11  begin
12    return (Name =>
13            Head (Name,
14                  Name_String'Length),
15            Address =>
16            Head (Address,
17                  Address_String'Length));
18  end "+";
19
20  function "+" (Left, Right : Person)
21              return Person
22
23  is
24  begin
25    return (Name => Left.Name,
26            Address => Right.Address);
27  end "+";
28
29  procedure Display (P : Person) is
30  begin
31    Put_Line ("Name: " & P.Name);
32    Put_Line ("Address: " & P.Address);
33    New_Line;
```

(continues on next page)

(continued from previous page)

```
32     end Display;
33
34 end Addresses;
```

Listing 34: show_address_addition.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Addresses;   use Addresses;
3
4 procedure Show_Address_Addition is
5     John : Person := "John" + "4 Main Street";
6     Jane : Person := "Jane" + "7 High Street";
7 begin
8     Display (John);
9     Display (Jane);
10    Put_Line ("-----");
11
12    Jane := Jane + John;
13    Display (Jane);
14 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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Addresses;   use Addresses;
3
4 procedure Show_Address_Addition is
5     John : constant Person :=
6         "John" + "4 Main Street";
7     Jane : constant Person :=
```

(continues on next page)

(continued from previous page)

```
8         "Jane" + "7 High Street";
9 begin
10    if Jane + John = John + Jane then
11        Put_Line ("It's commutative!");
12    else
13        Put_Line ("It's not commutative!");
14    end if;
15 end Show_Address_Addition;
```

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](#)¹⁴⁹

10.3 Expression functions

Usually, we implement Ada functions with a construct like this: **begin return X; end;**. In other words, we create a **begin ... end;** block and we have at least one **return** statement in that block. An expression function, in contrast, is a function that is implemented with a simple expression in parentheses, such as (X);. In this case, we don't use a **begin ... end;** block or a **return** statement.

As an example of an expression, let's say we want to implement a function named `Is_Zero` that checks if the value of the integer parameter `I` is zero. We can implement this function with the expression `I = 0`. In the usual approach, we would create the implementation by writing **is begin return I = 0; end Is_Zero;**. When using expression functions, however, we can simplify the implementation by just writing **is (I = 0);**. This is the complete code of `Is_Zero` using an expression function:

Listing 36: `expr_func.ads`

```
1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean is
5         (I = 0);
6
7 end Expr_Func;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_Expression_Function_1
MD5: 44779999566f764279e1c2f292226f95

¹⁴⁹ <http://www.ada-auth.org/standards/22rm/html/RM-6-1.html>

An expression function has the same effect as the usual version using a block. In fact, the code above is similar to this implementation of the `Is_Zero` function using a block:

Listing 37: `expr_func.ads`

```
1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean;
5
6 end Expr_Func;
```

Listing 38: `expr_func.adb`

```
1 package body Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean is
5     begin
6         return I = 0;
7     end Is_Zero;
8
9 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`

```
1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean is
5     begin
6         return I = 0;
7     end Is_Zero;
8
9 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

```
1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean;
5
6 end Expr_Func;
```

Listing 41: expr_func.adb

```
1 package body Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean is
5         (I = 0);
6
7 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

```
1 package My_Data is
2
3     type Data is private;
4
5     function Is_Valid (D : Data)
6         return Boolean;
7
8 private
9
10    type Data is record
11        Valid : Boolean;
12    end record;
13
14    function Is_Valid (D : Data)
15        return Boolean is
16        (D.Valid);
17
18 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

```
1 package My_Data is
2
3     type Data is private;
4
5     function Is_Valid (D : Data)
6         return Boolean;
7
8 private
9
10    type Data is record
11        Valid : Boolean;
12    end record;
13
14 end My_Data;
```

Listing 44: my_data.adb

```
1 package body My_Data is
2
3     function Is_Valid (D : Data)
4         return Boolean is
5         (D.Valid);
6
7 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](#)¹⁵⁰
-

10.4 Overloading

Note: This section was originally written by Robert A. Duff and published as [Gem #50: Overload Resolution](#)¹⁵¹.

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:

¹⁵⁰ <http://www.ada-auth.org/standards/22rm/html/RM-6-8.html>

¹⁵¹ <https://www.adacore.com/gems/gem-50>

Listing 45: show_overloading.adb

```
1 procedure Show_Overloading is
2
3   package Types is
4     type Sequence is null record;
5     type Set is null record;
6
7     procedure Print (S : Sequence) is null;
8     procedure Print (S : Set) is null;
9   end Types;
10
11   use Types;
12
13   X : Sequence;
14 begin
15   -- Compiler selects Print (S : Sequence)
16   Print (X);
17 end Show_Overloading;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
MD5: 020c4f04285c80c1050d8edbaaf2dbcae

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

```
1 procedure Show_Top_Down_Overloading is
2
3   package Types is
4     type Sequence is null record;
5     type Set is null record;
6
7     function Empty return Sequence is
8       ((others => <>));
9
10    function Empty return Set is
11      ((others => <>));
12
13    procedure Print_Sequence (S : Sequence) is
14      null;
15
16    procedure Print_Set (S : Set) is
17      null;
18  end Types;
19
20  use Types;
21
22  X : Sequence;
23 begin
24   -- Compiler selects function
25   -- Empty return Sequence
26   Print_Sequence (Empty);
27 end Show_Top_Down_Overloading;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
 MD5: 3b776a3efdee3d7e583ddb5159c9a1b

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

```

1 procedure Show_Overloading_Error is
2
3   package Types is
4     type Sequence is null record;
5     type Set is null record;
6
7     function Empty return Sequence is
8       ((others => <>));
9
10    function Empty return Set is
11      ((others => <>));
12
13    procedure Print (S : Sequence) is
14      null;
15
16    procedure Print (S : Set) is
17      null;
18  end Types;
19
20  use Types;
21
22  X : Sequence;
23 begin
24   Print (Empty); -- Illegal!
25 end Show_Overloading_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
 MD5: 5182c517a1aff4568ab2404ac66fda8

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`

```
1 procedure Show_Literal_Resolution is
2
3     type Apple_Count is range 0 .. 100;
4
5     procedure Peel (Count : Apple_Count) is null;
6 begin
7     Peel (20);
8 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`

```
1 procedure Show_Literal_Resolution_Error is
2
3     type Apple_Count is range 0 .. 100;
4     procedure Slice (Count : Apple_Count) is null;
5
6     type Orange_Count is range 0 .. 10_000;
7     procedure Slice (Count : Orange_Count) is null;
8 begin
9     Slice (Count => (10_000)); -- Illegal!
10 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: e3dd1fd0c403bcf672f4bab881b8ef9
```

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.

10.5 Operator Overloading

We've seen *previously* (page 370) that we can define custom operators for any type. We've also seen that subprograms can be *overloaded* (page 379). Since operators are functions, we're essentially talking about operator overloading, as we're defining the same operator (say `+` or `-`) for different types.

As another example of operator overloading, in the Ada standard library, operators are defined for the `Complex` type of the `Ada.Numerics.Generic_Complex_Types` package. This package contains not only the definition of the `+` operator for two objects of `Complex` type, but also for combination of `Complex` and other types. For instance, we can find these declarations:

```
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 Operators¹⁵²
 - G.1.1 Complex Types¹⁵³
-

10.6 Operator Overriding

We can also override operators of derived types. This allows for modifying the behavior of operators for the corresponding derived types.

To override an operator of a derived type, we simply implement a function for that operator. This is the same as how we implement custom operators (as we've seen previously).

As an example, when adding two fixed-point values, the result might be out of range, which causes an exception to be raised. A common strategy to avoid exceptions in this case is to saturate the resulting value. This strategy is typically employed in signal processing algorithms, for example.

In this example, we declare and use the 32-bit fixed-point type `TQ31`:

¹⁵² <http://www.ada-auth.org/standards/22rm/html/RM-6-6.html>

¹⁵³ <http://www.ada-auth.org/standards/22rm/html/RM-G-1-1.html>

Listing 51: fixed_point.ads

```

1 package Fixed_Point is
2
3   D : constant := 2.0 ** (-31);
4   type TQ31 is delta D range -1.0 .. 1.0 - D;
5
6 end Fixed_Point;
```

Listing 52: show_sat_op.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Fixed_Point; use Fixed_Point;
3
4 procedure Show_Sat_Op is
5   A, B, C : TQ31;
6 begin
7   A := TQ31'Last;
8   B := TQ31'Last;
9   C := A + B;
10
11   Put_Line (A'Image & " + "
12             & B'Image & " = "
13             & C'Image);
14
15   A := TQ31'First;
16   B := TQ31'First;
17   C := A + B;
18
19   Put_Line (A'Image & " + "
20             & B'Image & " = "
21             & C'Image);
22
23 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

```

1 package Fixed_Point is
2
3   D : constant := 2.0 ** (-31);
4   type TQ31 is delta D range -1.0 .. 1.0 - D;
5
6   function "+" (Left, Right : TQ31)
7               return TQ31;
8
9 end Fixed_Point;
```

Listing 54: fixed_point.adb

```
1 package body Fixed_Point is
2
3     function "+" (Left, Right : TQ31)
4         return TQ31
5     is
6         type TQ31_2 is
7             delta TQ31'Delta
8             range TQ31'First * 2.0 .. TQ31'Last * 2.0;
9
10        L : constant TQ31_2 := TQ31_2 (Left);
11        R : constant TQ31_2 := TQ31_2 (Right);
12        Res : TQ31_2;
13    begin
14        Res := L + R;
15
16        if Res > TQ31_2 (TQ31'Last) then
17            return TQ31'Last;
18        elsif Res < TQ31_2 (TQ31'First) then
19            return TQ31'First;
20        else
21            return TQ31 (Res);
22        end if;
23    end "+";
24
25 end Fixed_Point;
```

Listing 55: show_sat_op.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Fixed_Point; use Fixed_Point;
3
4 procedure Show_Sat_Op is
5     A, B, C : TQ31;
6 begin
7     A := TQ31'Last;
8     B := TQ31'Last;
9     C := A + B;
10
11     Put_Line (A'Image & " + "
12             & B'Image & " = "
13             & C'Image);
14
15     A := TQ31'First;
16     B := TQ31'First;
17     C := A + B;
18
19     Put_Line (A'Image & " + "
20             & B'Image & " = "
21             & C'Image);
22
23 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).

10.7 Nonreturning procedures

Usually, when calling a procedure P, we expect that it returns to the caller's *thread of control* after performing some action in the body of P. However, there are situations where a procedure never returns. We can indicate this fact by using the No_Return aspect in the subprogram declaration.

A typical example is that of a server that is designed to run forever until the process is killed or the machine where the server runs is switched off. This server can be implemented as an endless loop. For example:

Listing 56: servers.ads

```
1 package Servers is
2
3   procedure Run_Server
4     with No_Return;
5
6 end Servers;
```

Listing 57: servers.adb

```
1 package body Servers is
2
3   procedure Run_Server is
4     begin
5       pragma Warnings
6         (Off,
7          "implied return after this statement");
8       while True loop
9         -- Processing happens here...
10        null;
11      end loop;
12    end Run_Server;
13
14 end Servers;
```

Listing 58: show_endless_loop.adb

```
1 with Servers; use Servers;
2
3 procedure Show_Endless_Loop is
4 begin
```

(continues on next page)

(continued from previous page)

```
5   Run_Server;
6 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`

```
1 package Loggers is
2
3   Logged_Failure : exception;
4
5   procedure Log_And_Raise (Msg : String)
6     with No_Return;
7
8 end Loggers;
```

Listing 60: `loggers.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Loggers is
4
5   procedure Log_And_Raise (Msg : String) is
6   begin
7     Put_Line (Msg);
8     raise Logged_Failure;
9   end Log_And_Raise;
10
11 end Loggers;
```

Listing 61: `show_no_return_exception.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Loggers; use Loggers;
3
4 procedure Show_No_Return_Exception is
5   Check_Passed : constant Boolean := False;
6 begin
7   if not Check_Passed then
8     Log_And_Raise ("Check failed!");
9     Put_Line ("This line will not be reached!");
10  end if;
11 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`

```
1 package Loggers is
2
3   Logged_Failure : exception;
4
5   procedure Log_And_Raise (Msg : String)
6     with No_Return;
7
8 end Loggers;
```

Listing 63: `loggers.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Loggers is
4
5   procedure Log_And_Raise (Msg : String) is
6     begin
7       Put_Line (Msg);
8     end Log_And_Raise;
9
10 end Loggers;
```

Listing 64: `show_no_return_exception.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Loggers; use Loggers;
3
4 procedure Show_No_Return_Exception is
5   Check_Passed : constant Boolean := False;
6 begin
7   if not Check_Passed then
8     Log_And_Raise ("Check failed!");
9     Put_Line ("This line will not be reached!");
10  end if;
11 end Show_No_Return_Exception;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Nonreturning_Procedures.
↳ Erroneous_Log_Exception
MD5: e44fd8df0529dda5749e85b9e300a999

Build output

```
loggers.adb:7:07: warning: implied return after this statement will raise Program_
↳ Error [enabled by default]
loggers.adb:7:07: warning: procedure "Log_And_Raise" is marked as No_Return_
↳ [enabled by default]
```


Runtime output

```
Check failed!
```

```
raised PROGRAM_ERROR : loggers.adb:7 implicit return with No_Return
```

Here, `Program_Error` is raised when `Log_And_Raise` returns to the `Show_No_Return_Exception` procedure.

In the Ada Reference Manual

- [6.5.1 Nonreturning Subprograms](#)¹⁵⁴

10.8 Inline subprograms

[Inlining](#)¹⁵⁵ refers to a kind of optimization where the code of a subprogram is expanded at the point of the call in place of the call itself.

In modern compilers, inlining depends on the optimization level selected by the user. For example, if we select the higher optimization level, the compiler will perform automatic inlining aggressively.

In the GNAT toolchain

The highest optimization level (-O3) of GNAT performs aggressive automatic inlining. This could mean that this level inlines too much rather than not enough. As a result, the cache may become an issue and the overall performance may be worse than the one we would achieve by compiling the same code with optimization level 2 (-O2). Therefore, the general recommendation is to not *just* select -O3 for the optimized version of an application, but instead compare it the optimized version built with -O2.

It's important to highlight that the inlining we're referring above happens automatically, so the decision about which subprogram is inlined depends entirely on the compiler. However, in some cases, it's better to reduce the optimization level and perform manual inlining instead of automatic inlining. We do that by using the `Inline` aspect.

Let's look at this example:

Listing 65: float_arrays.ads

```
1 package Float_Arrays is
2
3     type Float_Array is
4         array (Positive range <>) of Float;
5
6     function Average (Data : Float_Array)
7         return Float
8         with Inline;
9
10 end Float_Arrays;
```

¹⁵⁴ <http://www.ada-auth.org/standards/22rm/html/RM-6-5-1.html>

¹⁵⁵ https://en.wikipedia.org/wiki/Inline_expansion

Listing 66: float_arrays.adb

```

1 package body Float_Arrays is
2
3     function Average (Data : Float_Array)
4         return Float
5     is
6         Total : Float := 0.0;
7     begin
8         for Value of Data loop
9             Total := Total + Value;
10        end loop;
11        return Total / Float (Data'Length);
12    end Average;
13
14 end Float_Arrays;

```

Listing 67: compute_average.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Float_Arrays; use Float_Arrays;
4
5 procedure Compute_Average is
6     Values      : constant Float_Array :=
7         (10.0, 11.0, 12.0, 13.0);
8     Average_Value : Float;
9 begin
10    Average_Value := Average (Values);
11    Put_Line ("Average = "
12            & Float'Image (Average_Value));
13 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](#)¹⁵⁶
-

10.9 Null Procedures

Null procedures are procedures that don't have any effect, as their body is empty. We declare a null procedure by simply writing `is null` in its declaration. For example:

Listing 68: null_procs.ads

```
1 package Null_Procs is
2
3     procedure Do_Nothing (Msg : String) is null;
4
5 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

```
1 with Null_Procs; use Null_Procs;
2
3 procedure Show_Null_Proc is
4 begin
5     Do_Nothing ("Hello");
6 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:

¹⁵⁶ <http://www.ada-auth.org/standards/22rm/html/RM-6-3-2.html>

Listing 70: null_procs.ads

```
1 package Null_Procs is
2
3     procedure Do_Nothing (Msg : String);
4
5 end Null_Procs;
```

Listing 71: null_procs.adb

```
1 package body Null_Procs is
2
3     procedure Do_Nothing (Msg : String) is
4     begin
5         null;
6     end Do_Nothing;
7
8 end Null_Procs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Null_Proc_1
MD5: d0c9dc9265ebbaa9603681182dee1d92
```

10.9.1 Null procedures and overriding

We can use null procedures as a way to simulate interfaces for non-tagged types — similar to what actual interfaces do for tagged types. For example, we may start by declaring a type and null procedures that operate on that type. For example, let's model a very simple API:

Listing 72: simple_storage.ads

```
1 package Simple_Storage is
2
3     type Storage_Model is null record;
4
5     procedure Set (S : in out Storage_Model;
6                 V : String) is null;
7     procedure Display (S : Storage_Model) is null;
8
9 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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Simple_Storage; use Simple_Storage;
3
4 procedure Show_Null_Proc is
```

(continues on next page)

(continued from previous page)

```
5   S : Storage_Model;
6   begin
7     Put_Line ("Setting 24...");
8     Set (S, "24");
9     Display (S);
10  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`

```
1 package Simple_Storage is
2
3   type Storage_Model is null record;
4
5   procedure Set (S : in out Storage_Model;
6                 V : String) is null;
7   procedure Display (S : Storage_Model) is null;
8
9   type Integer_Storage is private;
10
11  procedure Set (S : in out Integer_Storage;
12               V : String);
13  procedure Display (S : Integer_Storage);
14
15 private
16
17  type Integer_Storage is record
18    V : Integer := 0;
19  end record;
20
21 end Simple_Storage;
```

Listing 75: `simple_storage.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Simple_Storage is
4
5   procedure Set (S : in out Integer_Storage;
6                 V : String) is
7   begin
8     S.V := Integer'Value (V);
9   end Set;
10
11  procedure Display (S : Integer_Storage) is
12  begin
13    Put_Line ("Value: " & S.V'Image);
```

(continues on next page)

(continued from previous page)

```
14   end Display;
15
16 end Simple_Storage;
```

Listing 76: show_null_proc.adb

```
1 with Ada.Text_IO;   use Ada.Text_IO;
2 with Simple_Storage; use Simple_Storage;
3
4 procedure Show_Null_Proc is
5   S : Integer_Storage;
6 begin
7   Put_Line ("Setting 24...");
8   Set (S, "24");
9   Display (S);
10 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 151) contains an extended example that makes use of null procedures.

In the Ada Reference Manual

- [6.7 Null Procedures](#)¹⁵⁷
-

¹⁵⁷ <http://www.ada-auth.org/standards/22rm/html/RM-6-7.html>

EXCEPTIONS

11.1 Asserts

When we want to indicate a condition in the code that must always be valid, we can use the pragma `Assert`. As the name implies, when we use this pragma, we're *asserting* some truth about the source-code. (We can also use the procedural form, as we'll see later.)

Important

Another method to assert the truth about the source-code is to use pre and post-conditions.

A simple assert has this form:

Listing 1: show_pragma_assert.adb

```
1 procedure Show_Pragma_Assert is
2   I : constant Integer := 10;
3
4   pragma Assert (I = 10);
5 begin
6   null;
7 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 2: show_pragma_assert.adb

```
1 procedure Show_Pragma_Assert is
2   I : constant Integer := 11;
3
4   pragma Assert (I = 10, "I is not 10");
5 begin
6   null;
7 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 be implemented as follows:

Listing 3: show_procedure_assert.adb

```
1 with Ada.Assertions; use Ada.Assertions;
2
3 procedure Show_Procedure_Assert is
4   I : constant Integer := 11;
5
6 begin
7   Assert (I = 10, "I is not 10");
8 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 4: show_assertion_error.adb

```
1 with Ada.Assertions; use Ada.Assertions;
2
3 procedure Show_Assertion_Error is
4   I : constant Integer := 11;
5
6 begin
7   if I /= 10 then
8     raise Assertion_Error with "I is not 10";
9   end if;
10
11 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_Policy¹⁵⁸

11.2 Assertion policies

We can activate and deactivate assertions based on assertion policies. We can do that by using the pragma `Assertion_Policy`. As an argument to this pragma, we indicate whether a specific policy must be checked or ignored.

For example, we can deactivate assertion checks by specifying `Assert => Ignore`. Similarly, we can activate assertion checks by specifying `Assert => Check`. Let's see a code example:

Listing 5: `show_pragma_assertion_policy.adb`

```

1 procedure Show_Pragma_Assertion_Policy is
2   I : constant Integer := 11;
3
4   pragma Assertion_Policy (Assert => Ignore);
5 begin
6   pragma Assert (I = 10);
7 end Show_Pragma_Assertion_Policy;
```

Code block metadata

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	Description
<code>Assert</code>	Check assertions
<code>Static_Predicate</code>	Check static predicates
<code>Dynamic_Predicate</code>	Check dynamic predicates
<code>Pre</code>	Check pre-conditions
<code>Pre'Class</code>	Check pre-condition of classes of tagged types
<code>Post</code>	Check post-conditions
<code>Post'Class</code>	Check post-condition of classes of tagged types
<code>Type_Invariant</code>	Check type invariants
<code>Type_Invariant'Class</code>	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:

¹⁵⁸ <http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html>

- Assertions
- Assert_And_Cut
- Assume
- Contract_Cases
- Debug
- Ghost
- Initial_Condition
- Invariant
- Invariant'[Class](#)
- Loop_Invariant
- Loop_Variant
- Postcondition
- Precondition
- Predicate
- Refined_Post
- Statement_Assertions
- Subprogram_Variant

Also, in addition to Check and Ignore, GNAT allows you to set a policy to Disable and Suppressible.

You can read more about them in the [GNAT Reference Manual](#)¹⁵⁹.

You can specify multiple policies in a single call to `Assertion_Policy`. For example, you can activate all policies by writing:

Listing 6: `show_multiple_assertion_policies.adb`

```
1 procedure Show_Multiple_Assertion_Policies is
2   pragma Assertion_Policy
3     (Assert           => Check,
4      Static_Predicate => Check,
5      Dynamic_Predicate => Check,
6      Pre              => Check,
7      Pre'Class         => Check,
8      Post             => Check,
9      Post'Class       => Check,
10     Type_Invariant   => Check,
11     Type_Invariant'Class => Check);
12 begin
13   null;
14 end Show_Multiple_Assertion_Policies;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Multiple_
↳Assertion_Policies
MD5: 3abbf97160b755b84cc4f7e652ca5551
```

¹⁵⁹ https://gcc.gnu.org/onlinedocs/gnat_rm/Pragma-Assertion_005fPolicy.html

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](#)¹⁶⁰. 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 7: `show_assert_procedure_policy.adb`

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Ada.Assertions; use Ada.Assertions;
3
4 procedure Show_Assert_Procedure_Policy is
5   pragma Assertion_Policy (Assert => Ignore);
6
7   I : constant Integer := 1;
8 begin
9   Put_Line ("----- Pragma Assert -----");
10  pragma Assert (I = 0);
11
12  Put_Line ("---- Procedure Assert ----");
13  Assert (I = 0);
14
15  Put_Line ("Finished.");
16 end Show_Assert_Procedure_Policy;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Assert_Procedure_Policy
 ↪ 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](#)¹⁶¹

¹⁶⁰ https://gcc.gnu.org/onlinedocs/gnat_ugn/The-Configuration-Pragmas-Files.html#The-Configuration-Pragmas-Files

¹⁶¹ <http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html>

11.3 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
Storage_Check	Storage_Error

In addition, we can use `All_Checks` to refer to all those checks above at once.

Let's discuss each check and see code examples where those checks are performed. Note that all examples are erroneous, so please avoid reusing them elsewhere.

11.3.1 Access Check

As you know, an object of an access type might be null. It would be an error to dereference this object, as it doesn't indicate a valid position in memory. Therefore, the access check verifies that an access object is not null when dereferencing it. For example:

Listing 8: show_access_check.adb

```

1 procedure Show_Access_Check is
2
3     type Integer_Access is access Integer;
4
5     AI : Integer_Access;
6 begin
7     AI.all := 10;
8 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 554).) For example:

Listing 9: show_access_check.adb

```
1 procedure Show_Access_Check is
2
3   type Integer_Access is
4     access all Integer;
5
6   type Safe_Integer_Access is
7     not null access all Integer;
8
9   AI : Integer_Access;
10  SAI : Safe_Integer_Access := new Integer;
11
12 begin
13   SAI := Safe_Integer_Access (AI);
14 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**).

11.3.2 Discriminant Check

As we've seen earlier, a variant record is a record with discriminants that allows for changing its structure. In operations such as an assignment, it's important to ensure that the discriminants of the objects match — i.e. to ensure that the structure of the objects matches. The discriminant check verifies whether this is the case. For example:

Listing 10: show_discriminant_check.adb

```

1 procedure Show_Discriminant_Check is
2
3     type Rec (Valid : Boolean) is record
4         case Valid is
5             when True =>
6                 Counter : Integer;
7             when False =>
8                 null;
9         end case;
10    end record;
11
12    R : Rec (Valid => False);
13 begin
14     R := (Valid => True,
15         Counter => 10);
16 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 11: show_discriminant_check.adb

```

1 procedure Show_Discriminant_Check is
2
3     type Rec (Valid : Boolean) is record
4         case Valid is
5             when True =>
6                 Counter : Integer;
7             when False =>
8                 null;
9         end case;
10    end record;
```

(continues on next page)

(continued from previous page)

```

11
12     R : Rec (Valid => False);
13     I : Integer;
14 begin
15     I := R.Counter;
16 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**.

11.3.3 Division Check

The division check verifies that we're not trying to divide a value by zero when using the **/**, **rem** and **mod** operators. For example:

Listing 12: ops.ads

```

1 package Ops is
2     function Div_Op (A, B : Integer)
3         return Integer is
4         (A / B);
5
6     function Rem_Op (A, B : Integer)
7         return Integer is
8         (A rem B);
9
10    function Mod_Op (A, B : Integer)
11        return Integer is
12        (A mod B);
13 end Ops;

```

Listing 13: show_division_check.adb

```

1 with Ops; use Ops;
2
3 procedure Show_Division_Check is
4     I : Integer;
5 begin
6     I := Div_Op (10, 0);
7     I := Rem_Op (10, 0);

```

(continues on next page)

(continued from previous page)

```
8   I := Mod_Op (10, 0);
9 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.

11.3.4 Index Check

We use indices to access components of an array. An index check verifies that the index we're using to access a specific component is within the array's bounds. For example:

Listing 14: `show_index_check.adb`

```
1 procedure Show_Index_Check is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   function Value_Of (A : Integer_Array;
7                     I : Integer)
8     return Integer
9   is
10    type Half_Integer_Array is new
11      Integer_Array (A'First ..
12                    A'First + A'Length / 2);
13
14    A_2 : Half_Integer_Array := (others => 0);
15  begin
16    return A_2 (I);
17  end Value_Of;
18
19  Arr_1 : Integer_Array (1 .. 10) :=
20    (others => 1);
21
22  begin
23    Arr_1 (10) := Value_Of (Arr_1, 10);
24
25  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.

11.3.5 Length Check

In array assignments, both arrays must have the same length. To ensure that this is the case, a length check is performed. For example:

Listing 15: `show_length_check.adb`

```

1 procedure Show_Length_Check is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   procedure Assign (To : out Integer_Array;
7                   From : Integer_Array) is
8   begin
9     To := From;
10  end Assign;
11
12  Arr_1 : Integer_Array (1 .. 10);
13  Arr_2 : Integer_Array (1 .. 9) :=
14    (others => 1);
15
16 begin
17   Assign (Arr_1, Arr_2);
18 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.

11.3.6 Overflow Check

Operations on scalar objects might lead to overflow, which, if not checked, lead to wrong information being computed and stored. Therefore, an overflow check verifies that the value of a scalar object is within the base range of its type. For example:

Listing 16: `show_overflow_check.adb`

```

1 procedure Show_Overflow_Check is
2   A, B : Integer;
3 begin
```

(continues on next page)

(continued from previous page)

```
4   A := Integer'Last;
5   B := 1;
6
7   A := A + B;
8 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.

11.3.7 Range Check

The range check verifies that a scalar value is within a specific range — for instance, the range of a subtype. Let's see an example:

Listing 17: show_range_check.adb

```
1 procedure Show_Range_Check is
2
3     subtype Int_1_10 is Integer range 1 .. 10;
4
5     I : Int_1_10;
6
7 begin
8     I := 11;
9 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.

11.3.8 Tag Check

The tag check ensures that the tag of a tagged object matches the expected tag in a dispatching operation. For example:

Listing 18: p.ads

```
1 package P is
2
3     type T is tagged null record;
4     type T1 is new T with null record;
5     type T2 is new T with null record;
6
7 end P;
```

Listing 19: show_tag_check.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Tags;
3
4 with P;          use P;
5
6 procedure Show_Tag_Check is
7
8     A1 : T'Class := T1'(null record);
9     A2 : T'Class := T2'(null record);
10
11 begin
12     Put_Line ("A1'Tag: "
13             & Ada.Tags.Expanded_Name (A1'Tag));
14     Put_Line ("A2'Tag: "
15             & Ada.Tags.Expanded_Name (A2'Tag));
16
17     A2 := A1;
18
19 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.

11.3.9 Accessibility Check

The accessibility check verifies that the accessibility level of an entity matches the expected level. We discuss accessibility levels *in a later chapter* (page 534).

Let's look at an example that mixes access types and anonymous access types. Here, we use an anonymous access type in the declaration of A1 and a named access type in the declaration of A2:

Listing 20: p.ads

```
1 package P is
2
3     type T is tagged null record;
4     type T_Class is access all T'Class;
5
6 end P;
```

Listing 21: show_accessibility_check.adb

```
1 with P; use P;
2
3 procedure Show_Accessibility_Check is
4
5     A1 : access T'Class := new T;
6     A2 : T_Class;
7
8 begin
9     A2 := T_Class (A1);
10
11 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 22: show_accessibility_check.adb

```

1 with P; use P;
2
3 procedure Show_Accessibility_Check is
4
5     A : access T'Class := new T;
6
7     procedure P (A : T_Class) is null;
8
9 begin
10     P (T_Class (A));
11
12 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.

11.3.10 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-00280¹⁶²:

Listing 23: p.ads

```

1 with Ada.Finalization;
2 with Ada.Unchecked_Deallocation;
3
4 package P is
5     type T1 is new
6         Ada.Finalization.Controlled with null record;
7     procedure Finalize (X : in out T1);
8
9     type T2 is new
10        Ada.Finalization.Controlled with null record;
11    procedure Finalize (X : in out T2);
12
13    X1 : T1;
14
```

(continues on next page)

¹⁶² <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00280.txt?rev=1.12&raw=N>

(continued from previous page)

```
15  type T2_Ref is access T2;
16  procedure Free is new
17     Ada.Unchecked_Deallocation (T2, T2_Ref);
18  end P;
```

Listing 24: p.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body P is
4
5     procedure Finalize (X : in out T1) is
6         X2 : T2_Ref := new T2;
7     begin
8         Put_Line ("Finalizing T1...");
9         Free (X2);
10    end Finalize;
11
12    procedure Finalize (X : in out T2) is
13    begin
14        Put_Line ("Finalizing T2...");
15    end Finalize;
16
17  end P;
```

Listing 25: show_allocation_check.adb

```
1  with P; use P;
2
3  procedure Show_Allocation_Check is
4     X2 : T2_Ref := new T2;
5  begin
6     Free (X2);
7  end Show_Allocation_Check;
```

Code block metadata

```
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.

11.3.11 Elaboration Check

The elaboration check verifies that subprograms — or protected entries, or task activations — have been elaborated before being called.

This is an example adapted from AI-00064¹⁶³:

Listing 26: p.ads

```
1 function P return Integer;
```

Listing 27: p.adb

```
1 function P return Integer is
2 begin
3   return 1;
4 end P;
```

Listing 28: show_elaboration_check.adb

```
1 with P;
2
3 procedure Show_Elaboration_Check is
4
5   function F return Integer;
6
7   type Pointer_To_Func is
8     access function return Integer;
9
10  X : constant Pointer_To_Func := P'Access;
11
12  Y : constant Integer := F;
13  Z : constant Pointer_To_Func := X;
14
15  -- Renaming-as-body
16  function F return Integer renames Z.all;
17 begin
18   null;
19 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

¹⁶³ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00064.txt?rev=1.12&raw=N>

providing a body to F — this is called renaming-as-body. Consequently, the compiler doesn't complain that a body is missing for function F. (If you comment out the function renaming, you'll see that the compiler can then detect the missing body.) Therefore, at runtime, the elaboration check fails because the body of the first declaration of the F function is actually missing.

11.3.12 Storage Check

The storage check ensures that the storage pool has enough space when allocating memory. Let's revisit an example that we *discussed earlier* (page 85):

Listing 29: custom_types.ads

```
1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_Reserved_Access is access UInt_7
6         with Storage_Size => 8;
7
8 end Custom_Types;
```

Listing 30: show_storage_check.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Storage_Check is
6
7     RAV1, RAV2 : UInt_7_Reserved_Access;
8
9 begin
10    Put_Line ("Allocating RAV1...");
11    RAV1 := new UInt_7;
12
13    Put_Line ("Allocating RAV2...");
14    RAV2 := new UInt_7;
15
16    New_Line;
17 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.

- [11.5 Suppressing Checks](#)¹⁶⁴
-

11.4 Ada.Exceptions package

Note: Parts of this section were originally published as [Gem #142 : Exception-ally](#)¹⁶⁵

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](#)¹⁶⁶
-

11.4.1 Retrieving exception information

Exceptions also embed information set by the run-time itself that can be retrieved by calling the `Exception_Information` function. The function `Exception_Information` also displays the `Exception_Message`.

For example:

```
exception
  when E : others =>
    Put_Line
      (Ada.Exceptions.Exception_Information (E));
```

¹⁶⁴ <http://www.ada-auth.org/standards/22rm/html/RM-11-5.html>

¹⁶⁵ <https://www.adacore.com/gems/gem-142-exceptions>

¹⁶⁶ <http://www.ada-auth.org/standards/22rm/html/RM-11-4-1.html>

In the GNAT toolchain

In the case of GNAT, the information provided by an exception might include the source location where the exception was raised and a nonsymbolic traceback.

You can also retrieve this information individually. Here, you can use:

- the `Exception_Name` functions — and its derivatives `Wide_Exception_Name` and `Wide_Wide_Exception_Name` — to retrieve the name of an exception.
- the `Exception_Message` function to retrieve the message associated with an exception.

Let's see a complete example:

Listing 31: `show_exception_info.adb`

```
1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 procedure Show_Exception_Info is
5
6     Custom_Exception : exception;
7
8     procedure Nested is
9     begin
10         raise Custom_Exception
11         with "We got a problem";
12     end Nested;
13
14 begin
15     Nested;
16
17 exception
18     when E : others =>
19         Put_Line ("Exception info: "
20                 & Exception_Information (E));
21         Put_Line ("Exception name: "
22                 & Exception_Name (E));
23         Put_Line ("Exception msg: "
24                 & Exception_Message (E));
25 end Show_Exception_Info;
```

11.4.2 Collecting exceptions

Save_Occurrence

You can save an exception occurrence using the `Save_Occurrence` procedure. (Note that a `Save_Occurrence` function exists as well.)

For example, the following application collects exceptions into a list and displays them after running the `Test_Exceptions` procedure:

Listing 32: `exception_tests.ads`

```
1 with Ada.Exceptions; use Ada.Exceptions;
2
3 package Exception_Tests is
4
```

(continues on next page)

(continued from previous page)

```

5 Custom_Exception : exception;
6
7 type All_Exception_Occur is
8   array (Positive range <>) of
9     Exception_Occurrence;
10
11 procedure Test_Exceptions
12   (All_Occur : in out All_Exception_Occur;
13    Last_Occur : out Integer);
14
15 end Exception_Tests;

```

Listing 33: exception_tests.adb

```

1 package body Exception_Tests is
2
3   procedure Save_To_List
4     (E      : Exception_Occurrence;
5      All_Occur : in out All_Exception_Occur;
6      Last_Occur : in out Integer)
7   is
8     L : Integer renames Last_Occur;
9     O : All_Exception_Occur renames All_Occur;
10    begin
11      L := L + 1;
12      if L > O'Last then
13        raise Constraint_Error
14          with "Cannot save occurrence";
15      end if;
16
17      Save_Occurrence (Target => O (L),
18                     Source => E);
19    end Save_To_List;
20
21    procedure Test_Exceptions
22      (All_Occur : in out All_Exception_Occur;
23       Last_Occur : out Integer)
24    is
25
26      procedure Nested_1 is
27      begin
28        raise Custom_Exception
29          with "We got a problem";
30      exception
31        when E : others =>
32          Save_To_List (E,
33                     All_Occur,
34                     Last_Occur);
35      end Nested_1;
36
37      procedure Nested_2 is
38      begin
39        raise Constraint_Error
40          with "Constraint is not correct";
41      exception
42        when E : others =>
43          Save_To_List (E,
44                     All_Occur,
45                     Last_Occur);
46      end Nested_2;
47

```

(continues on next page)

(continued from previous page)

```
48   begin
49     Last_Occur := 0;
50
51     Nested_1;
52     Nested_2;
53   end Test_Exceptions;
54
55 end Exception_Tests;
```

Listing 34: show_exception_info.adb

```
1  with Ada.Text_IO;    use Ada.Text_IO;
2  with Ada.Exceptions; use Ada.Exceptions;
3
4  with Exception_Tests; use Exception_Tests;
5
6  procedure Show_Exception_Info is
7    L : Integer;
8    O : All_Exception_Occur (1 .. 10);
9  begin
10   Test_Exceptions (0, L);
11
12   for I in 0 'First .. L loop
13     Put_Line (Exception_Information (O (I)));
14   end loop;
15 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 35: exception_occurrence_stream.adb

```
1  with Ada.Text_IO;
2
3  with Ada.Streams.Stream_IO;
4  use  Ada.Streams.Stream_IO;
5
6  with Ada.Exceptions;
7  use  Ada.Exceptions;
8
```

(continues on next page)

(continued from previous page)

```

9  procedure Exception_Occurrence_Stream is
10
11     Custom_Exception : exception;
12
13     S : Stream_Access;
14
15     procedure Nested_1 is
16     begin
17         raise Custom_Exception
18             with "We got a problem";
19     exception
20         when E : others =>
21             Exception_Occurrence'Write (S, E);
22     end Nested_1;
23
24     procedure Nested_2 is
25     begin
26         raise Constraint_Error
27             with "Constraint is not correct";
28     exception
29         when E : others =>
30             Exception_Occurrence'Write (S, E);
31     end Nested_2;
32
33     F      : File_Type;
34     File_Name : constant String :=
35         "exceptions_file.bin";
36 begin
37     Create (F, Out_File, File_Name);
38     S := Stream (F);
39
40     Nested_1;
41     Nested_2;
42
43     Close (F);
44
45     Read_Exceptions : declare
46         E : Exception_Occurrence;
47     begin
48         Open (F, In_File, File_Name);
49         S := Stream (F);
50
51         while not End_Of_File (F) loop
52             Exception_Occurrence'Read (S, E);
53
54             Ada.Text_IO.Put_Line
55                 (Exception_Information (E));
56         end loop;
57         Close (F);
58     end Read_Exceptions;
59
60 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 occurrence in the file. In the `Read_Exceptions` block, we read the exceptions from the file by calling `Exception_Occurrence'Read`.

11.4.3 Debugging exceptions in the GNAT toolchain

Here is a typical exception handler that catches all unexpected exceptions in the application:

Listing 36: main.adb

```
1 with Ada.Exceptions;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 procedure Main is
5
6     procedure Nested is
7     begin
8         raise Constraint_Error
9             with "some message";
10    end Nested;
11
12 begin
13     Nested;
14
15 exception
16     when E : others =>
17         Put_Line
18             (Ada.Exceptions.Exception_Information (E));
19 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
```

(continues on next page)

(continued from previous page)

```
(gdb) run
Catchpoint 1, CONSTRAINT_ERROR at 0x000000000402860 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  0x00000000040426f in ada.exceptions.complete_occurrence (x=x@entry=0x637050)
at a-except.adb:934
#2  0x00000000040427b in ada.exceptions.complete_and_propagate_occurrence (
x=x@entry=0x637050) at a-except.adb:943
#3  0x0000000004042d0 in <__gnat_raise_exception> (e=0x62ec40 <constraint_error>,
message=...) at a-except.adb:982
#4  0x000000000402860 in main.nested ()
#5  0x00000000040287c 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 37: 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.

11.5 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 38: show_exception_renaming.adb

```

1 procedure Show_Exception_Renaming is
2   CE : exception renames Constraint_Error;
3 begin
4   raise CE;
5 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 39: internal_exceptions.ads

```

1 package Internal_Exceptions is
2
3   Int_E : exception;
4
5 end Internal_Exceptions;
```

Listing 40: test_constraints.ads

```

1 with Internal_Exceptions;
2
3 package Test_Constraints is
4
5   Ext_E : exception renames
6         Internal_Exceptions.Int_E;
7
8 end Test_Constraints;
```

Listing 41: show_exception_renaming_view.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 with Test_Constraints; use Test_Constraints;
5
6 procedure Show_Exception_Renaming_View is
7 begin
8   raise Ext_E;
9 exception
10  when E : others =>
11    Put_Line
12      (Ada.Exceptions.Exception_Information (E));
13 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 **withing** the `Internal_Exceptions` package. However, it is indirectly visible in that procedure via the renaming (`Ext_E`) in the `Test_Constraints` package.

In the Ada Reference Manual

- [8.5.2 Exception Renaming Declarations](#)¹⁶⁷
-

11.6 Out and Uninitialized

Note: This section was originally written by Robert Dewar and published as [Gem #150: Out and Uninitialized](#)¹⁶⁸

Perhaps surprisingly, the Ada standard indicates cases where objects passed to **out** and **in out** parameters might not be updated when a procedure terminates due to an exception. Let's take an example:

Listing 42: `show_out_uninitialized.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 procedure Show_Out_Uninitialized is
3
4     procedure Local (A      : in out Integer;
5                     Error : Boolean) is
6     begin
7         A := 1;
8
9         if Error then
10            raise Program_Error;
11        end if;
12    end Local;
13
14    B : Integer := 0;
15
16 begin
17     Local (B, Error => True);
18 exception
19     when Program_Error =>
20         Put_Line ("Value for B is"
21                 & Integer'Image (B)); -- "0"
22 end Show_Out_Uninitialized;
```

¹⁶⁷ <http://www.ada-auth.org/standards/22rm/html/RM-8-5-2.html>

¹⁶⁸ <https://www.adacore.com/gems/gem-150out-and-uninitialized>

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_Uninitialized_1
 ↪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 43: show_out_uninitialized_warnings.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Out_Uninitialized_Warnings is
4
5     procedure Local (A : in out Integer) is
6     begin
7         A := 1;
8         raise Program_Error;
9     end Local;
10
11     B : Integer := 0;
12
13 begin
14     Local (B);
15 exception
16     when others =>
17         Put_Line ("Value for B is"
18                 & Integer'Image (B));
19 end Show_Out_Uninitialized_Warnings;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_Uninitialized_2
 ↪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 44: show_out_initialized_rec.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Out_Initialized_Rec is
4
5     type Rec is tagged record
6         Field : Integer;
7     end record;
8
9     procedure Local (A : in out Rec) is
10        begin
11            A.Field := 1;
12            raise Program_Error;
13        end Local;
14
15        V : Rec;
16
17    begin
18        V.Field := 0;
19        Local (V);
20    exception
21        when others =>
22            Put_Line ("Value of Field is"
23                & V.Field'Img); -- "1"
24    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 45: exported_procedures.ads

```

1 package Exported_Procedures is
2
3   procedure Local (A      : in out Integer;
4                   Error : Boolean);
5   pragma Export_Procedure
6     (Local,
7      Mechanism => (A => Reference));
8
9 end Exported_Procedures;
```

Listing 46: exported_procedures.adb

```

1 package body Exported_Procedures is
2
3   procedure Local (A      : in out Integer;
4                   Error : Boolean) is
5     begin A := 1;
6     if Error then
7       raise Program_Error;
8     end if;
9   end Local;
10
11 end Exported_Procedures;
```

Listing 47: show_out_reference.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Exported_Procedures;
4 use Exported_Procedures;
5
6 procedure Show_Out_Reference is
7   B : Integer := 0;
8 begin
9   Local (B, Error => True);
10 exception
11   when Program_Error =>
12     Put_Line ("Value for B is"
13             & Integer'Image (B)); -- "1"
14 end Show_Out_Reference;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
↳Uninitialized_4
MD5: aed2788be2b3ceec19b28421c53fc66

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](http://www.ada-auth.org/standards/22rm/html/RM-11-6.html)¹⁶⁹). For instance, the following code makes an incorrect assumption:

¹⁶⁹ <http://www.ada-auth.org/standards/22rm/html/RM-11-6.html>

```
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](#)¹⁷⁰

11.7 Suppressing checks

11.7.1 pragma Suppress

Note: This section was originally written by Gary Dismukes and published as [Gem #63: The Effect of Pragma Suppress](#)¹⁷¹.

One of Ada's key strengths has always been its strong typing. The language imposes stringent checking of type and subtype properties to help prevent accidental violations of the type system that are a common source of program bugs in other less-strict languages such as C. This is done using a combination of compile-time restrictions (legality rules), that prohibit mixing values of different types, together with run-time checks to catch violations of various dynamic properties. Examples are checking values against subtype constraints and preventing dereferences of null access values.

At the same time, Ada does provide certain "loophole" features, such as `Unchecked_Conversion`, that allow selective bypassing of the normal safety features, which is sometimes necessary when interfacing with hardware or code written in other languages.

Ada also permits explicit suppression of the run-time checks that are there to ensure that various properties of objects are not violated. This suppression can be done using `pragma Suppress`, as well as by using a compile-time switch on most implementations — in the case of GNAT, with the `-gnatp` switch.

In addition to allowing all checks to be suppressed, `pragma Suppress` supports suppression of specific forms of check, such as `Index_Check` for array indexing, `Range_Check` for scalar bounds checking, and `Access_Check` for dereferencing of access values. (See section 11.5 of the Ada Reference Manual for further details.)

Here's a simple example of suppressing index checks within a specific subprogram:

```
procedure Main is
  procedure Sort_Array (A : in out Some_Array) is
    pragma Suppress (Index_Check);
    -- ~~~~~
    -- eliminate check overhead
  begin
    ...
  end Sort_Array;
end Main;
```

¹⁷⁰ <http://www.ada-auth.org/standards/22rm/html/RM-11-6.html>

¹⁷¹ <https://www.adacore.com/gems/gem-63>

Unlike a feature such as `Unchecked_Conversion`, however, the purpose of check suppression is not to enable programs to subvert the type system, though many programmers seem to have that misconception.

What's important to understand about `pragma Suppress` is that it only gives permission to the implementation to remove checks, but doesn't require such elimination. The intention of `Suppress` is not to allow bypassing of Ada semantics, but rather to improve efficiency, and the Ada Reference Manual has a clear statement to that effect in the note in RM-11.5, paragraph 29:

There is no guarantee that a suppressed check is actually removed; hence a `pragma Suppress` should be used only for efficiency reasons.

There is associated Implementation Advice that recommends that implementations should minimize the code executed for checks that have been suppressed, but it's still the responsibility of the programmer to ensure that the correct functioning of the program doesn't depend on checks not being performed.

There are various reasons why a compiler might choose not to remove a check. On some hardware, certain checks may be essentially free, such as null pointer checks or arithmetic overflow, and it might be impractical or add extra cost to suppress the check. Another example where it wouldn't make sense to remove checks is for an operation implemented by a call to a run-time routine, where the check might be only a small part of a more expensive operation done out of line.

Furthermore, in many cases GNAT can determine at compile time that a given run-time check is guaranteed to be violated. In such situations, it gives a warning that an exception will be raised, and generates code specifically to raise the exception. Here's an example:

```
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:

```
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.

11.7.2 pragma Unsuppress

We can use `pragma Unsuppress` to reverse the effect of a `pragma Suppress`. While `pragma Suppress` gives permission to the compiler to remove a specific check, `pragma Unsuppress` revokes that permission.

Let's see an example:

Listing 48: show_index_check.adb

```
1 procedure Show_Index_Check is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   pragma Suppress (Index_Check);
7   -- from now on, the compiler may
8   -- eliminate index checks...
9
10  function Unchecked_Value_Of
11    (A : Integer_Array;
12     I : Integer)
13    return Integer
14  is
15    type Half_Integer_Array is new
16      Integer_Array (A'First ..
17                    A'First + A'Length / 2);
18
19    A_2 : Half_Integer_Array := (others => 0);
20  begin
21    return A_2 (I);
22  end Unchecked_Value_Of;
23
24  pragma Unsuppress (Index_Check);
25  -- from now on, index checks are
26  -- typically performed...
27
28  function Value_Of
29    (A : Integer_Array;
30     I : Integer)
31    return Integer
32  is
33    type Half_Integer_Array is new
34      Integer_Array (A'First ..
35                    A'First + A'Length / 2);
36
37    A_2 : Half_Integer_Array := (others => 0);
```

(continues on next page)

(continued from previous page)

```
38   begin
39     return A_2 (I);
40   end Value_Of;
41
42   Arr_1 : Integer_Array (1 .. 10) :=
43     (others => 1);
44
45   begin
46     Arr_1 (10) := Unchecked_Value_Of (Arr_1, 10);
47     Arr_1 (10) := Value_Of (Arr_1, 10);
48
49   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 Checks¹⁷²](#)

¹⁷² <http://www.ada-auth.org/standards/22rm/html/RM-11-5.html>

Part III

Modular programming

PACKAGES

12.1 Package renaming

We've seen in the Introduction to Ada course that we can [rename packages](#)¹⁷³.

In the Ada Reference Manual

- [10.1.1 Compilation Units - Library Units](#)¹⁷⁴
-

12.1.1 Grouping packages

A use-case that we haven't mentioned in that course is that we can apply package renaming to group individual packages into a common hierarchy. For example:

Listing 1: driver_m1.ads

```
1 package Driver_M1 is
2
3 end Driver_M1;
```

Listing 2: driver_m2.ads

```
1 package Driver_M2 is
2
3 end Driver_M2;
```

Listing 3: drivers.ads

```
1 package Drivers
2   with Pure is
3
4 end Drivers;
```

Listing 4: drivers-m1.ads

```
1 with Driver_M1;
2
3 package Drivers.M1 renames Driver_M1;
```

¹⁷³ https://learn.adacore.com/courses/intro-to-ada/chapters/modular_programming.html#intro-ada-package-renaming

¹⁷⁴ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html>

Listing 5: drivers-m2.ads

```
1 with Driver_M2;
2
3 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

```
1 package Driver_M1 is
2
3     Counter_2 : Integer := Drivers.Counter;
4
5 end Driver_M1;
```

Listing 7: drivers.ads

```
1 package Drivers is
2
3     Counter : Integer := 0;
4
5 end Drivers;
```

Listing 8: drivers-m1.ads

```
1 with Driver_M1;
2
3 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.

12.1.2 Child of renamed package

Note that we cannot create a child package using a parent package name that was introduced by a renaming. For example, let's say we want to create a child package `Ext` for the `Drivers.M1` package we've seen earlier. We cannot just declare a `Drivers.M1.Ext` package like this:

```
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

```
1 package Driver_M1.Ext is
2
3 end Driver_M1.Ext;
```

Listing 10: dummy.adb

```
1 -- A package called Drivers.M1.Ext is
2 -- automatically available!
3
4 with Drivers.M1.Ext;
5
6 procedure Dummy is
7 begin
8     null;
9 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.)

12.1.3 Backwards-compatibility via renaming

We can also use renaming to ensure backwards-compatibility when changing the package hierarchy. For example, we could adapt the previous source-code by:

- converting `Driver_M1` and `Driver_M2` to child packages of `Drivers`, and
- using package renaming to *mimic* the original names (`Driver_M1` and `Driver_M2`).

This is the adapted code:

Listing 11: drivers.ads

```
1 package Drivers
2     with Pure is
3
4 end Drivers;
```


Listing 12: drivers-m1.ads

```
1 -- We've converted Driver_M1 to
2 -- Drivers.M1:
3
4 package Drivers.M1 is
5
6 end Drivers.M1;
```

Listing 13: drivers-m2.ads

```
1 -- We've converted Driver_M2 to
2 -- Drivers.M2:
3
4 package Drivers.M2 is
5
6 end Drivers.M2;
```

Listing 14: driver_m1.ads

```
1 -- Original Driver_M1 package still
2 -- available via package renaming:
3
4 with Drivers.M1;
5
6 package Driver_M1 renames Drivers.M1;
```

Listing 15: driver_m2.ads

```
1 -- Original Driver_M2 package still
2 -- available via package renaming:
3
4 with Drivers.M2;
5
6 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.

12.2 Private packages

In this section, we discuss the concept of private packages. However, before we proceed with the discussion, let's recapitulate some important ideas that we've seen earlier.

In the [Introduction to Ada course](#)¹⁷⁵, we've seen that encapsulation plays an important role in modular programming. By using the private part of a package specification, we can disclose some information, but, at the same time, prevent that this information gets accessed where it shouldn't be used directly. Similarly, we've seen that we can use the

¹⁷⁵ <https://learn.adacore.com/courses/intro-to-ada/chapters/privacy.html#intro-ada-course-privacy>

private part of a package to distinguish between the *partial and full view* (page 35) of a data type.

The main application of private packages is to create private child packages, whose purpose is to serve as internal implementation packages within a package hierarchy. By doing so, we can expose the internals to other public child packages, but prevent that external clients can directly access them.

As we'll see next, there are many rules that ensure that internal visibility is enforced for those private child packages. At the same time, the same rules ensure that private packages aren't visible outside of the package hierarchy.

12.2.1 Declaration and usage

We declare private packages by using the **private** keyword. For example, let's say we have a package named `Data_Processing`:

Listing 16: `data_processing.ads`

```
1 package Data_Processing is
2
3 -- ...
4
5 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`

```
1 private package Data_Processing.Calculations is
2
3 -- ...
4
5 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`

```
1 package Data_Processing is
2
3     type Data is private;
4
5     procedure Process (D : in out Data);
6
7 private
8
9     type Data is null record;
```

(continues on next page)

(continued from previous page)

```
10
11 end Data_Processing;
```

Listing 19: data_processing-calculations.ads

```
1 private package Data_Processing.Calculations is
2
3     procedure Calculate (D : in out Data);
4
5 end Data_Processing.Calculations;
```

Listing 20: data_processing.adb

```
1 with Data_Processing.Calculations;
2 use Data_Processing.Calculations;
3
4 package body Data_Processing is
5
6     procedure Process (D : in out Data) is
7     begin
8         Calculate (D);
9     end Process;
10
11 end Data_Processing;
```

Listing 21: data_processing-calculations.adb

```
1 package body Data_Processing.Calculations is
2
3     procedure Calculate (D : in out Data) is
4     begin
5         -- Dummy implementation...
6         null;
7     end Calculate;
8
9 end Data_Processing.Calculations;
```

Listing 22: test_data_processing.adb

```
1 with Data_Processing; use Data_Processing;
2
3 procedure Test_Data_Processing is
4     D : Data;
5 begin
6     Process (D);
7 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.

12.2.2 Private sibling packages

We can introduce another private package `Advanced_Calculations` as a child of `Data_Processing` and refer to the `Calculations` package in its specification:

Listing 23: `data_processing.ads`

```

1 package Data_Processing is
2     type Data is private;
3     procedure Process (D : in out Data);
4
5 private
6     type Data is null record;
7
8 end Data_Processing;
```

Listing 24: `data_processing-calculations.ads`

```

1 private package Data_Processing.Calculations is
2     procedure Calculate (D : in out Data);
3
4 end Data_Processing.Calculations;
```

Listing 25: `data_processing-advanced_calculations.ads`

```

1 with Data_Processing.Calculations;
2 use Data_Processing.Calculations;
3
4 private
5 package Data_Processing.Advanced_Calculations is
6     procedure Advanced_Calculate (D : in out Data)
7         renames Calculate;
8
9 end Data_Processing.Advanced_Calculations;
```

Listing 26: `data_processing.adb`

```

1 with Data_Processing.Advanced_Calculations;
2 use Data_Processing.Advanced_Calculations;
3
4 package body Data_Processing is
5     procedure Process (D : in out Data) is
6     begin
7         Advanced_Calculate (D);
8     end Process;
9
10 end Data_Processing;
```

Listing 27: `data_processing-calculations.adb`

```

1 package body Data_Processing.Calculations is
2     procedure Calculate (D : in out Data) is
3     begin
4         -- Dummy implementation...
5
```

(continues on next page)

(continued from previous page)

```
6     null;
7     end Calculate;
8
9 end Data_Processing.Calculations;
```

Listing 28: test_data_processing.adb

```
1 with Data_Processing; use Data_Processing;
2
3 procedure Test_Data_Processing is
4     D : Data;
5 begin
6     Process (D);
7 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

```
1 with Data_Processing.Calculations;
2 use Data_Processing.Calculations;
3
4 package Data_Processing.Advanced_Calculations is
5
6     procedure Advanced_Calculate (D : in out Data)
7         renames Calculate;
8
9 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

```
1 package Data_Processing.Advanced_Calculations is
2
3     procedure Advanced_Calculate (D : in out Data);
```

(continues on next page)

(continued from previous page)

```
4
5 end Data_Processing.Advanced_Calculations;
```

Listing 31: data_processing-advanced_calculations.adb

```
1 with Data_Processing.Calculations;
2 use Data_Processing.Calculations;
3
4 package body Data_Processing.Advanced_Calculations
5 is
6
7     procedure Advanced_Calculate (D : in out Data)
8     is
9     begin
10        Calculate (D);
11    end Advanced_Calculate;
12
13 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.

12.2.3 Outside the package tree

While we can use a with-clause of a private child package in the body of the `Data_Processing` package, we cannot do the same outside the package tree. For example, we cannot refer to it in the `Test_Data_Processing` procedure:

Listing 32: test_data_processing.adb

```
1 with Data_Processing; use Data_Processing;
2
3 with Data_Processing.Calculations;
4 use Data_Processing.Calculations;
5
6 procedure Test_Data_Processing is
7     D : Data;
8     begin
9         Calculate (D);
10    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`

```
1 package Data_Processing.Calculations.Child is
2
3   procedure Process (D : in out Data);
4
5 end Data_Processing.Calculations.Child;
```

Listing 34: `data_processing-calculations-child.adb`

```
1 package body Data_Processing.Calculations.Child is
2
3   procedure Process (D : in out Data) is
4     begin
5       Calculate (D);
6   end Process;
7
8 end Data_Processing.Calculations.Child;
```

Listing 35: `test_data_processing.adb`

```
1 with Data_Processing; use Data_Processing;
2
3 with Data_Processing.Calculations.Child;
4 use Data_Processing.Calculations.Child;
5
6 procedure Test_Data_Processing is
7   D : Data;
8 begin
9   Calculate (D);
10 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 454).

Note that subprograms can also be declared private. We'll see this *in another section* (page 473).

Important

We've discussed package renaming *in a previous section* (page 435). We can rename a package as a private package, too. For example:

Listing 36: driver_m1.ads

```
1 package Driver_M1 is
2
3 end Driver_M1;
```

Listing 37: drivers.ads

```
1 package Drivers
2   with Pure is
3
4 end Drivers;
```

Listing 38: drivers-m1.ads

```
1 with Driver_M1;
2
3 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

```
1 with Driver_M1;
2 with Drivers.M1;
3
4 procedure Test_Driver is
5 begin
6   null;
7 end Test_Driver;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↳Package_Renaming
MD5: 55415978604ccea4eeab02df13cd2f4
```

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 Units¹⁷⁶
-

12.3 Private with clauses

12.3.1 Definition and usage

A private with clause allows us to refer to a package in the private part of another package. For example, if we want to refer to package P in the private part of Data, we can write **private with P**:

Listing 40: p.ads

```
1 package P is
2
3     type T is null record;
4
5 end P;
```

Listing 41: data.ads

```
1 private with P;
2
3 package Data is
4
5     type T2 is private;
6
7 private
8
9     -- Information from P is
10    -- visible here
11    type T2 is new P.T;
12
13 end Data;
```

Listing 42: main.adb

```
1 with Data; use Data;
2
3 procedure Main is
4     A : T2;
5 begin
6     null;
7 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:

¹⁷⁶ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html>

Listing 43: data.ads

```
1 private with P;  
2  
3 package Data is  
4     -- ERROR: information from P  
5     -- isn't visible here  
6  
7     type T2 is new P.T;  
8  
9  
10 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

```
1 package P is  
2  
3     type T is null record;  
4  
5     procedure Process (A : T) is null;  
6  
7 end P;
```

Listing 45: data.ads

```
1 private with P;  
2  
3 package Data is  
4  
5     type T2 is private;  
6  
7     procedure Process (A : T2);  
8  
9 private  
10  
11     -- Information from P is  
12     -- visible here  
13     type T2 is new P.T;  
14  
15 end Data;
```

Listing 46: data.adb

```
1 package body Data is  
2  
3     procedure Process (A : T2) is
```

(continues on next page)

(continued from previous page)

```
4   begin
5     P.Process (P.T (A));
6   end Process;
7
8 end Data;
```

Listing 47: main.adb

```
1 with Data; use Data;
2
3 procedure Main is
4   A : T2;
5   begin
6     null;
7   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.

12.3.2 Referring to private child package

There's one case where using a private with clause is the only way to refer to a package: when we want to refer to a private child package in another child package. For example, here we have a package P and its two child packages: **Private_Child** and Public_Child:

Listing 48: p.ads

```
1 package P is
2
3 end P;
```

Listing 49: p-private_child.ads

```
1 private package P.Private_Child is
2
3   type T is null record;
4
5 end P.Private_Child;
```

Listing 50: p-public_child.ads

```
1 private with P.Private_Child;
2
3 package P.Public_Child is
4
5   type T2 is private;
6
7 private
8
9   type T2 is new P.Private_Child.T;
10
11 end P.Public_Child;
```

Listing 51: test_parent_child.adb

```
1 with P.Public_Child; use P.Public_Child;
2
3 procedure Test_Parent_Child is
4   A : T2;
5 begin
6   null;
7 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

```
1 with P.Private_Child;
2
3 package P.Public_Child is
4
5   type T2 is private;
6
7 private
8
9   type T2 is new P.Private_Child.T;
10
11 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](#)¹⁷⁷

¹⁷⁷ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html>

12.4 Limited Visibility

Sometimes, we might face the situation where two packages depend on information from each other. Let's consider a package A that depends on a package B, and vice-versa:

Listing 53: a.ads

```
1 with B; use B;
2
3 package A is
4     type T1 is record
5         Value : T2;
6     end record;
7
8
9 end A;
```

Listing 54: b.ads

```
1 with A; use A;
2
3 package B is
4     type T2 is record
5         Value : T1;
6     end record;
7
8
9 end B;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Circular_
↳Dependency
MD5: ae64f33706f1c58603aff2c33b02c910
```

Build output

```
a.ads:1:06: error: circular unit dependency
a.ads:1:06: error: "A (spec)" depends on "B (spec)"
a.ads:1:06: error: "B (spec)" depends on "A (spec)"
a.ads:1:06: error: "A (spec)" depends on "A (spec)"
gprbuild: *** compilation phase failed
```

Here, we have two *mutually dependent types* (page 149) T1 and T2, which are declared in two packages A and B that refer to each other. These with clauses constitute a circular dependency, so the compiler cannot compile either of those packages.

One way to solve this problem is by transforming this circular dependency into a partial dependency. We do this by limiting the visibility — using a limited with clause. To use a limited with clause for a package P, we simply write **limited with P**.

If a package A has limited visibility to a package B, then all types from package B are visible as if they had been declared as *incomplete types* (page 34). For the specific case of the previous source-code example, this would be the limited visibility to package B from package A's perspective:

```
package B is
    -- Incomplete type
    type T2;
```

(continues on next page)

(continued from previous page)

```
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 149).

Keeping this information in mind, we can now correct the previous code by using limited with clauses for package A and declaring the component of the T1 record using an anonymous access type:

Listing 55: a.ads

```
1 limited with B;  
2  
3 package A is  
4  
5     type T1 is record  
6         Ref : access B.T2;  
7     end record;  
8  
9 end A;
```

Listing 56: b.ads

```
1 with A; use A;  
2  
3 package B is  
4  
5     type T2 is record  
6         Value : T1;  
7     end record;  
8  
9 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

```
1 limited with B;  
2  
3 package A is  
4  
5     type T1 is record  
6         Ref : access B.T2;  
7     end record;  
8  
9 end A;
```

Listing 58: b.ads

```
1 limited with A;
2
3 package B is
4
5     type T2 is record
6         Ref : access A.T1;
7     end record;
8
9 end B;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↳Visibility_2
MD5: 3884086e89400245346acfbff0691906
```

Now, both packages A and B have limited visibility to each other.

In the Ada Reference Manual

- [10.1.2 Context Clauses - With Clauses](#)¹⁷⁸
-

12.4.1 Limited visibility and private with clauses

We can limit the visibility and use *private with clauses* (page 446) 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

```
1 limited private with B;
2
3 package A is
4
5     type T1 is private;
6
7 private
8
9     -- Here, we have limited visibility
10    -- of package B
11
12    type T1 is record
13        Ref : access B.T2;
14    end record;
15
16 end A;
```

Listing 60: b.ads

```
1 private with A;
2
3 package B is
4
```

(continues on next page)

¹⁷⁸ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html>

(continued from previous page)

```

5   type T2 is private;
6
7   private
8
9   use A;
10
11  -- Here, we have full visibility
12  -- of package A
13
14  type T2 is record
15    Value : T1;
16  end record;
17
18 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.

12.4.2 Limited visibility and other elements

It's important to mention that the limited visibility we've been discussing so far is restricted to type declarations — which are seen as incomplete types. In fact, when we use a limited with clause, all other declarations have no visibility at all! For example, let's say we have a package Info that declares a constant Zero_Const and a function Zero_Func:

Listing 61: info.ads

```

1 package Info is
2
3   function Zero_Func return Integer is (0);
4
5   Zero_Const : constant := 0;
6
7 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

```

1 limited private with Info;
2
3 package A is
4
5   type T1 is private;
6
7 private
```

(continues on next page)

(continued from previous page)

```
8
9  type T1 is record
10     V : Integer := Info.Zero_Const;
11     W : Integer := Info.Zero_Func;
12 end record;
13
14 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
```

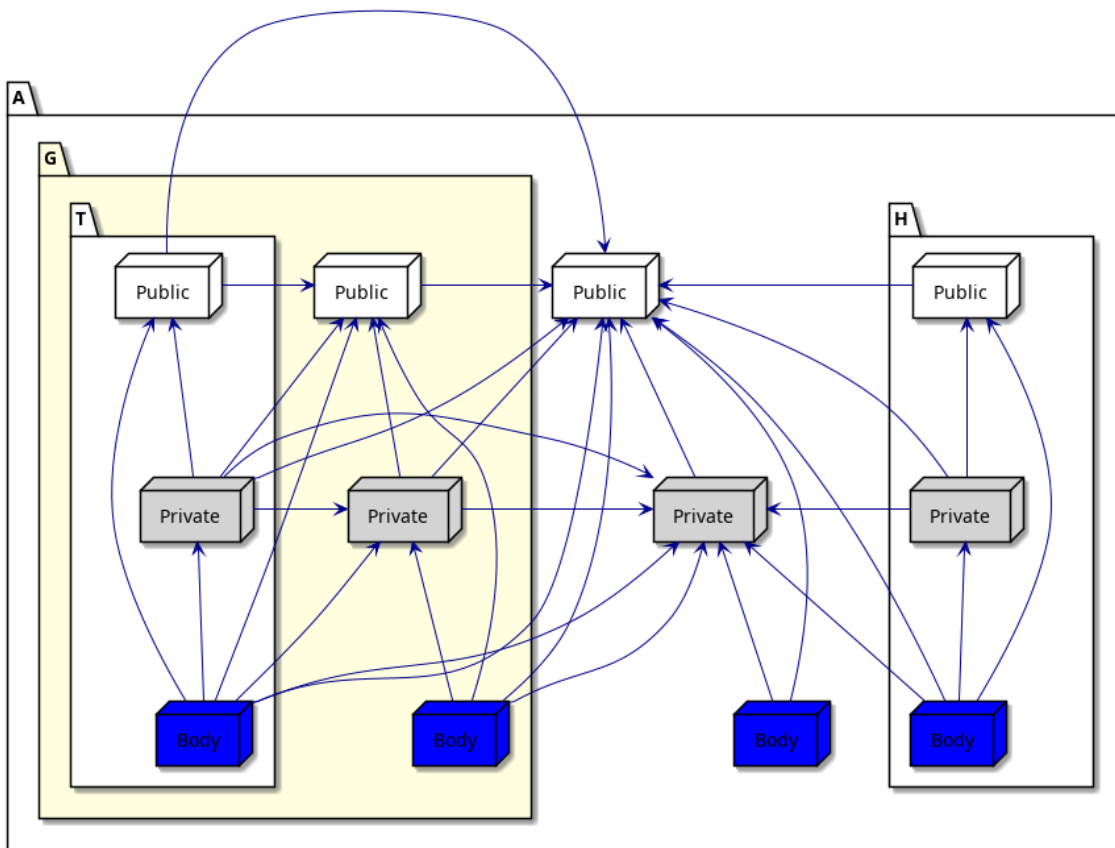
As expected, compilation fails because of the limited visibility — as `Zero_Const` and `Zero_Func` from the `Info` package are not visible in the private part of `A`. (Of course, if we revert to full visibility by simply removing the `limited` keyword from the example, the code compiles just fine.)

12.5 Visibility

In the previous sections, we already discussed visibility from various angles. However, it can be interesting to recapitulate this information with the help of diagrams that illustrate the different parts of a package and its relation with other units.

12.5.1 Automatic visibility

First, let's consider we have a package `A`, its children (`A.G` and `A.H`), and the grandchild `A.G.T`. As we've seen before, information of a parent package is automatically visible in its children. The following diagrams illustrates this:

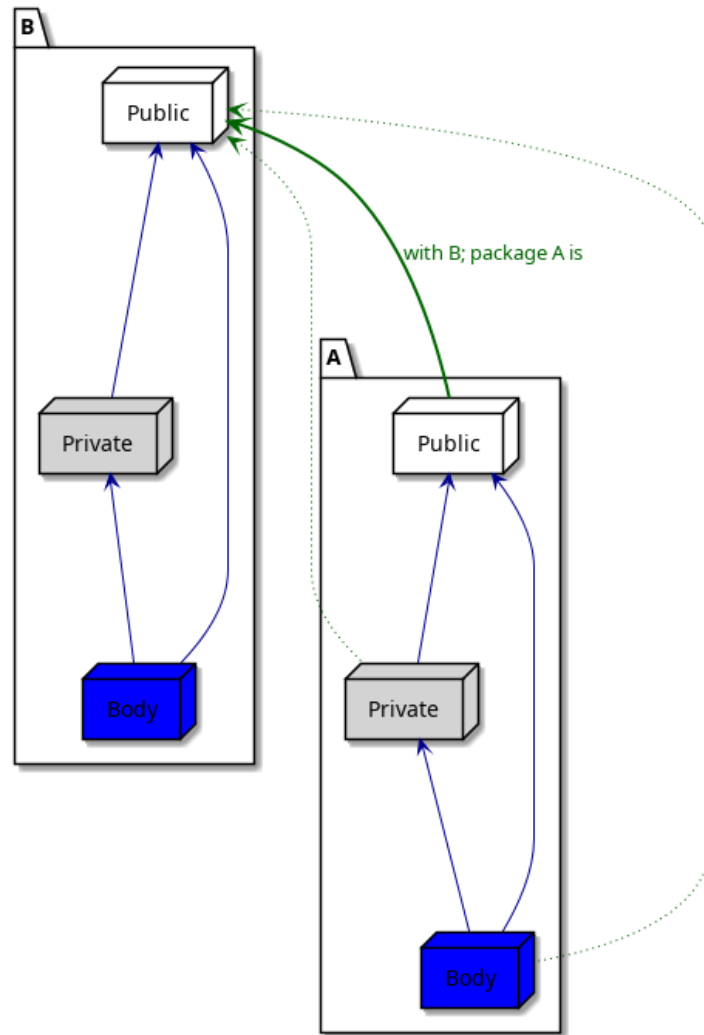


Because of this automatic visibility, many with clauses would be redundant in child packages. For example, we don't have to write `with A; package A.G is`, since the specification of package A is already visible in its child packages.

If we focus on package A.G (highlighted in the figure above), we see that it only has automatic visibility to its parent A, but not its child A.G.T. Also, it doesn't have visibility to its sibling A.H.

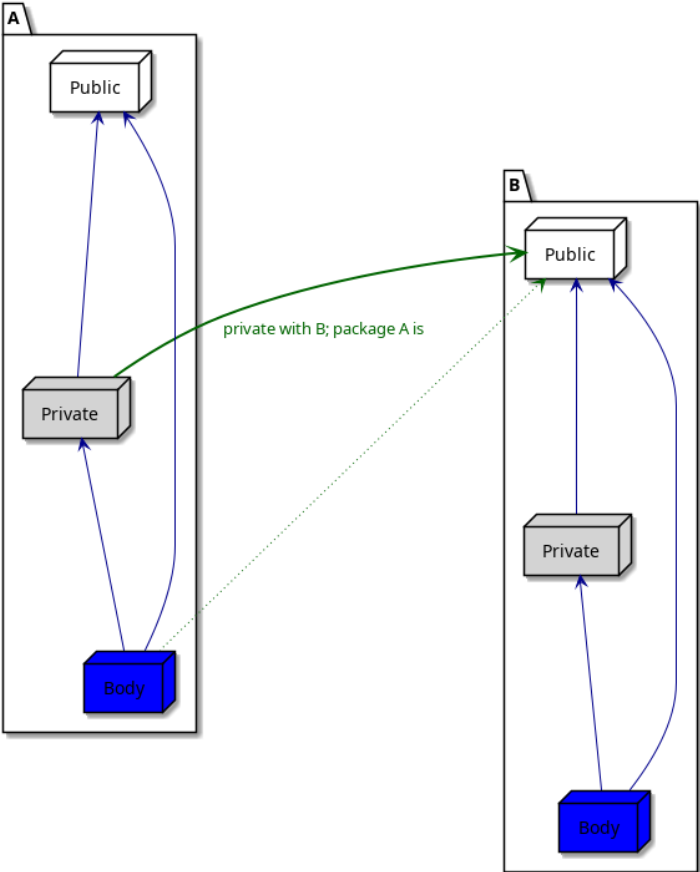
12.5.2 With clauses and visibility

In the rest of this section, we discuss all the situations where using with clauses is necessary to access the information of a package. Let's consider this example where we refer to a package B in the specification of a package A (using `with B`):

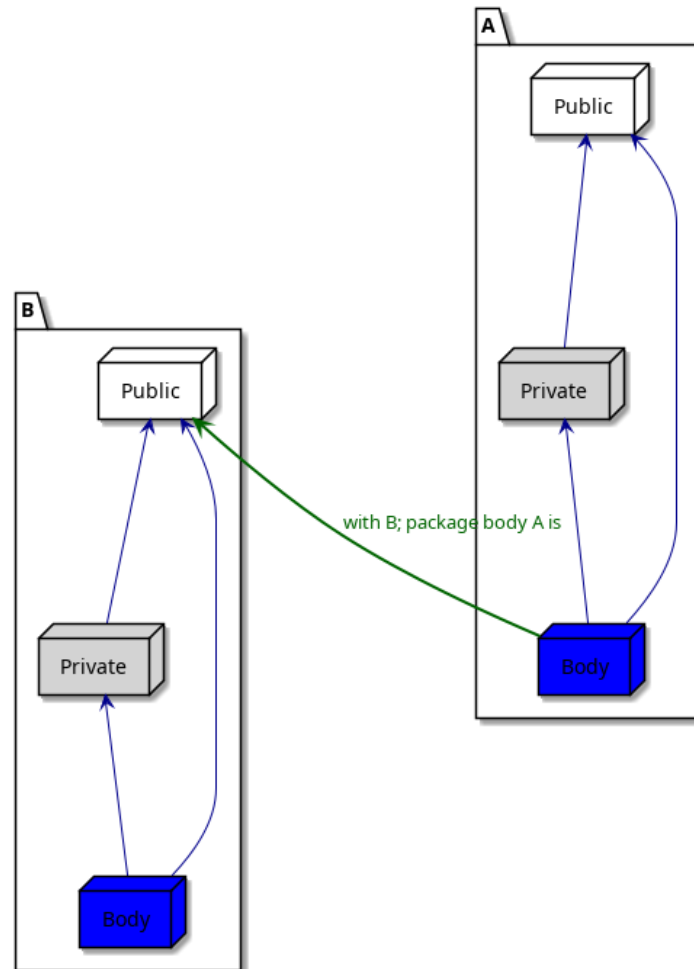


As we already know, the information from the public part of package B is visible in the public part of package A. In addition to that, it's also visible in the private part and in the body of package A. This is indicated by the dotted green arrows in the figure above.

Now, let's see the case where we refer to package B in the private part of package A (using **private with B**):



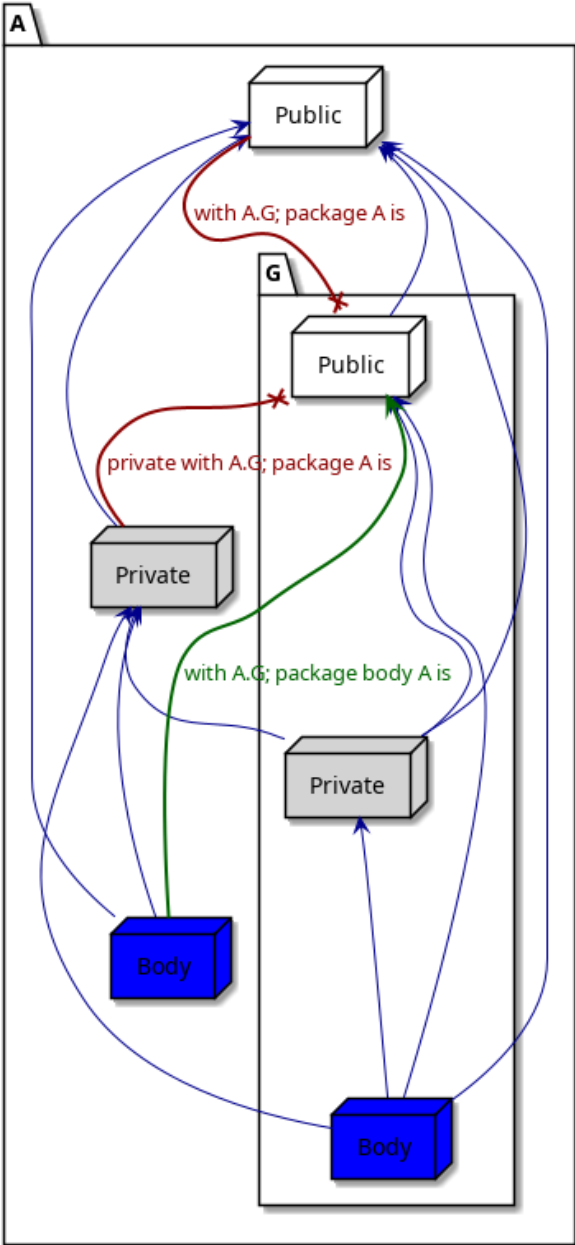
Here, the information is visible in the private part of package A, as well as in its body. Finally, let's see the case where we refer to package B in the body of package A:



Here, the information is only visible in the body of package A.

12.5.3 Circular dependency

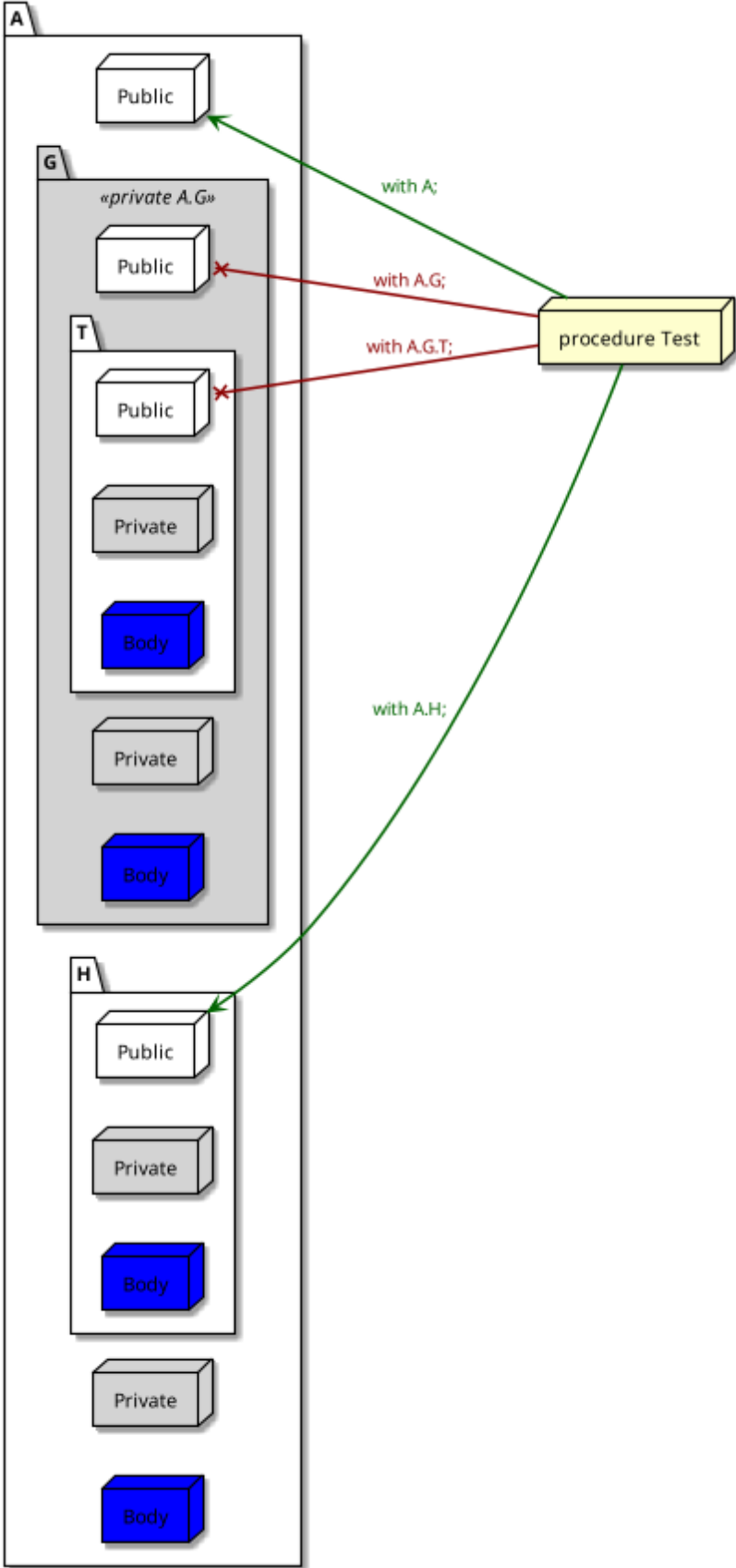
Let's return to package A and its descendants. As we've seen in previous sections, we cannot refer to a child package in the specification of its parent package because that would constitute circular dependency. (For example, we cannot write **with A.G; package A is.**) This situation — which causes a compilation error — is indicated by the red arrows in the figure below:



Note that referring to the child package A.G in the body of its parent is perfectly fine.

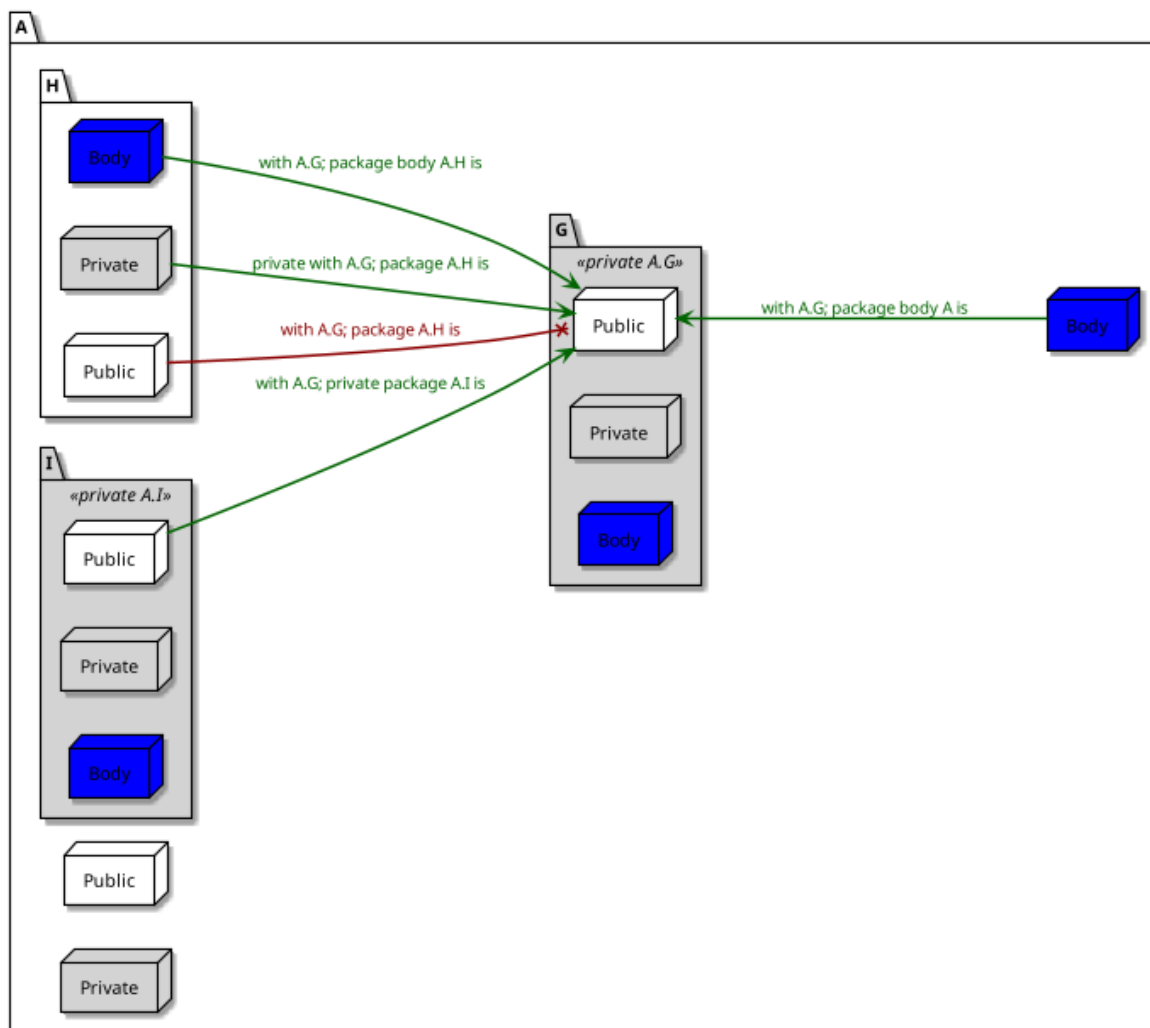
12.5.4 Private packages

The previous examples of this section only showed public packages. As we've seen before, we cannot refer to private packages outside of a package hierarchy, as we can see in the following example where we try to refer to package A and its descendants in the Test procedure:



As indicated by the red arrows, we cannot refer to the private child packages of A in the Test procedure, only the public child packages. Within the package hierarchy itself, we

cannot refer to the private package A.G in public sibling packages. For example:



Here, we cannot refer to the private package A.G in the public package A.H — as indicated by the red arrow. However, we can refer to the private package A.G in other private packages, such as A.I — as indicated by the green arrows.

12.6 Use type clause

Back in the [Introduction to Ada course](#)¹⁷⁹, 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

```

1 with Ada.Text_IO;
2
3 procedure Display_Message is
4 begin

```

(continues on next page)

¹⁷⁹ https://learn.adacore.com/courses/intro-to-ada/chapters/modular_programming.html#intro-ada-use-clause

(continued from previous page)

```
5 Ada.Text_IO.Put_Line ("Hello World!");
6 end Display_Message;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.No_Use_Clause
MD5: 4c6ff19809c13ebd2fdafa482914e5f8
```

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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Message is
4 begin
5     Put_Line ("Hello World!");
6 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](#)¹⁸⁰
-

12.6.1 Another use clause example

Let's now consider a simple package called `Points`, which contains the declaration of the `Point` type and two primitive: an `Init` function and an addition operator.

Listing 65: points.ads

```
1 package Points is
2
3     type Point is private;
4
5     function Init return Point;
6
7     function "+" (P : Point;
8                 I : Integer) return Point;
```

(continues on next page)

¹⁸⁰ <http://www.ada-auth.org/standards/22rm/html/RM-8-4.html>

(continued from previous page)

```
9
10 private
11
12     type Point is record
13         X, Y : Integer;
14     end record;
15
16     function Init return Point is (0, 0);
17
18     function "+" (P : Point;
19                 I : Integer) return Point is
20         (P.X + I, P.Y + I);
21
22 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

```
1 with Points; use Points;
2
3 procedure Show_Point is
4     P : Point;
5 begin
6     P := Init;
7     P := P + 1;
8 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.

12.6.2 Visibility and Readability

In certain situations, however, we might want to avoid the use clause. If that's the case, we can rewrite the previous implementation by removing the use clause and specifying the Points package in the prefixed form:

Listing 67: show_point.adb

```
1 with Points;
2
3 procedure Show_Point is
4     P : Points.Point;
5 begin
6     P := Points.Init;
7     P := Points."+" (P, 1);
8 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).

12.6.3 use type

As a compromise, we can have direct visibility to the operators of a certain type. We do this by using a use clause in the form `use type`. This allows us to simplify the previous example:

Listing 68: show_point.adb

```
1 with Points;
2
3 procedure Show_Point is
4     use type Points.Point;
5
6     P : Points.Point;
7 begin
8     P := Points.Init;
9     P := P + 1;
10 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.

12.6.4 use all type

If we want to have direct visibility to all primitives of a certain type (and not just its operators), we need to write a use clause in the form `use all type`. This allows us to simplify the previous example even further:

Listing 69: show_point.adb

```
1 with Points;
2
3 procedure Show_Point is
4     use all type Points.Point;
5
6     P : Points.Point;
7 begin
8     P := Init;
9     P := P + 1;
10 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.

12.7 Use clauses and naming conflicts

Visibility issues may arise when we have multiple use clauses. For instance, we might have types with the same name declared in multiple packages. This constitutes a naming conflict; in this case, the types become hidden — so they're not directly visible anymore, even if we have a use clause.

In the Ada Reference Manual

- 8.4 Use Clauses¹⁸¹
-

12.7.1 Code example

Let's start with a code example. First, we declare and implement a generic procedure that shows the value of a Complex object:

Listing 70: show_any_complex.ads

```
1 with Ada.Numerics.Generic_Complex_Types;
2
3 generic
4   with package Complex_Types is new
5     Ada.Numerics.Generic_Complex_Types (<>);
6 procedure Show_Any_Complex
7   (Msg : String;
8    Val : Complex_Types.Complex);
```

Listing 71: show_any_complex.adb

```
1 with Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO;
3
4 procedure Show_Any_Complex
5   (Msg : String;
6    Val : Complex_Types.Complex)
7 is
8   package Complex_Float_Types_IO is new
9     Ada.Text_IO.Complex_IO (Complex_Types);
10  use Complex_Float_Types_IO;
11
12  use Ada.Text_IO;
13 begin
14   Put (Msg & " ");
15   Put (Val);
16   New_Line;
17 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
```

¹⁸¹ <http://www.ada-auth.org/standards/22rm/html/RM-8-4.html>

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

```

1  with Ada.Numerics; use Ada.Numerics;
2
3  with Ada.Numerics.Generic_Complex_Types;
4
5  with Show_Any_Complex;
6
7  procedure Show_Use is
8      package Complex_Float_Types is new
9          Ada.Numerics.Generic_Complex_Types
10             (Real => Float);
11      use Complex_Float_Types;
12
13      procedure Show_Complex_Float is new
14          Show_Any_Complex (Complex_Float_Types);
15
16      C, D, X : Complex;
17  begin
18      C := Compose_From_Polar (3.0, Pi / 2.0);
19      D := Compose_From_Polar (5.0, Pi / 2.0);
20      X := C + D;
21
22      Show_Complex_Float ("C:", C);
23      Show_Complex_Float ("D:", D);
24      Show_Complex_Float ("X:", X);
25  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.

12.7.2 Naming conflict

Now, let's add the declaration of the `Complex_Long_Float_Types` package — a second instantiation of the `Generic_Complex_Types` package — to the code example:

Listing 73: show_use.adb

```

1  with Ada.Numerics; use Ada.Numerics;
2
3  with Ada.Numerics.Generic_Complex_Types;
4
5  with Show_Any_Complex;

```

(continues on next page)

(continued from previous page)

```

6
7 procedure Show_Use is
8   package Complex_Float_Types is new
9     Ada.Numerics.Generic_Complex_Types
10      (Real => Float);
11   use Complex_Float_Types;
12
13   package Complex_Long_Float_Types is new
14     Ada.Numerics.Generic_Complex_Types
15      (Real => Long_Float);
16   use Complex_Long_Float_Types;
17
18   procedure Show_Complex_Float is new
19     Show_Any_Complex (Complex_Float_Types);
20
21   C, D, X : Complex;
22   --      ^ ERROR: type is hidden!
23 begin
24   C := Compose_From_Polar (3.0, Pi / 2.0);
25   D := Compose_From_Polar (5.0, Pi / 2.0);
26   X := C + D;
27
28   Show_Complex_Float ("C:", C);
29   Show_Complex_Float ("D:", D);
30   Show_Complex_Float ("X:", X);
31 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.

12.7.3 Circumventing naming conflicts

As we know, a simple fix for this compilation error is to add the package prefix in the variable declaration:

Listing 74: `show_use.adb`

```

1 with Ada.Numerics; use Ada.Numerics;
2
3 with Ada.Numerics.Generic_Complex_Types;
4

```

(continues on next page)

(continued from previous page)

```

5 with Show_Any_Complex;
6
7 procedure Show_Use is
8   package Complex_Float_Types is new
9     Ada.Numerics.Generic_Complex_Types
10      (Real => Float);
11   use Complex_Float_Types;
12
13   package Complex_Long_Float_Types is new
14     Ada.Numerics.Generic_Complex_Types
15      (Real => Long_Float);
16   use Complex_Long_Float_Types;
17
18   procedure Show_Complex_Float is new
19     Show_Any_Complex (Complex_Float_Types);
20
21   C, D, X : Complex_Float_Types.Complex;
22   --      ^ SOLVED: package is now specified.
23 begin
24   C := Compose_From_Polar (3.0, Pi / 2.0);
25   D := Compose_From_Polar (5.0, Pi / 2.0);
26   X := C + D;
27
28   Show_Complex_Float ("C:", C);
29   Show_Complex_Float ("D:", D);
30   Show_Complex_Float ("X:", X);
31 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

```

1 with Ada.Numerics; use Ada.Numerics;
2
3 with Ada.Numerics.Generic_Complex_Types;
4
5 with Show_Any_Complex;
6
7 procedure Show_Use is
8   package Complex_Float_Types is new
9     Ada.Numerics.Generic_Complex_Types
10      (Real => Float);
11   use all type Complex_Float_Types.Complex;
12
13   package Complex_Long_Float_Types is new
14     Ada.Numerics.Generic_Complex_Types
15      (Real => Long_Float);
16   use all type Complex_Long_Float_Types.Complex;
17

```

(continues on next page)

(continued from previous page)

```

18  procedure Show_Complex_Float is new
19      Show_Any_Complex (Complex_Float_Types);
20
21  C, D, X : Complex_Float_Types.Complex;
22  begin
23      C := Compose_From_Polar (3.0, Pi / 2.0);
24      D := Compose_From_Polar (5.0, Pi / 2.0);
25      X := C + D;
26
27      Show_Complex_Float ("C:", C);
28      Show_Complex_Float ("D:", D);
29      Show_Complex_Float ("X:", X);
30  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

```

1  with Ada.Numerics; use Ada.Numerics;
2
3  with Ada.Numerics.Generic_Complex_Types;
4
5  with Show_Any_Complex;
6
7  procedure Show_Use is
8      package Complex_Float_Types is new
9          Ada.Numerics.Generic_Complex_Types
10             (Real => Float);
11      use all type Complex_Float_Types.Complex;
12
13      package Complex_Long_Float_Types is new
14          Ada.Numerics.Generic_Complex_Types
15             (Real => Long_Float);
16      use all type Complex_Long_Float_Types.Complex;
17
18      procedure Show_Complex_Float is new
19          Show_Any_Complex (Complex_Float_Types);
20
21      procedure Show_Complex_Long_Float is new
22          Show_Any_Complex (Complex_Long_Float_Types);
23
24      C, D, X : Complex_Float_Types.Complex;
25      E, F, Y : Complex_Long_Float_Types.Complex;
26  begin
27      C := Compose_From_Polar (3.0, Pi / 2.0);
28      D := Compose_From_Polar (5.0, Pi / 2.0);
29      X := C + D;
30
31      Show_Complex_Float ("C:", C);

```

(continues on next page)

(continued from previous page)

```
32 Show_Complex_Float ("D:", D);
33 Show_Complex_Float ("X:", X);
34
35 E := Compose_From_Polar (3.0, Pi / 2.0);
36 F := Compose_From_Polar (5.0, Pi / 2.0);
37 Y := E + F;
38
39 Show_Complex_Long_Float ("E:", E);
40 Show_Complex_Long_Float ("F:", F);
41 Show_Complex_Long_Float ("Y:", Y);
42 end Show_Use;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use-Clause_Naming_Conflicts.
↳ Use_Type-Clause_Complex_Types
MD5: 48f31250116f107d3143703debb3107d
```

Runtime output

```
C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)
E: ( 1.83697019872103E-16, 3.000000000000000E+00)
F: ( 3.06161699786838E-16, 5.000000000000000E+00)
Y: ( 4.89858719658941E-16, 8.000000000000000E+00)
```

As expected, the code compiles correctly.

SUBPROGRAMS AND MODULARITY

13.1 Private subprograms

We've seen *previously* (page 438) that we can declare private packages. Because packages and subprograms can both be library units, we can declare private subprograms as well. We do this by using the **private** keyword. For example:

Listing 1: test.ads

```
1 private procedure Test;
```

Listing 2: test.adb

```
1 procedure Test is
2 begin
3   null;
4 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

```
1 with Test;
2
3 procedure Show_Test is
4 begin
5   Test;
6 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 Units¹⁸²
 - 10.1.2 Context Clauses - With Clauses¹⁸³
-

13.1.1 Private subprograms of a package

A more useful example is to declare private subprograms of a package. For example:

Listing 4: `data_processing.ads`

```
1 package Data_Processing is
2
3     type Data is private;
4
5     procedure Process (D : in out Data);
6
7 private
8
9     type Data is record
10        F : Float;
11    end record;
12
13 end Data_Processing;
```

Listing 5: `data_processing.adb`

```
1 with Data_Processing.Calculate;
2
3 package body Data_Processing is
4
5     procedure Process (D : in out Data) is
6     begin
7         Calculate (D);
8     end Process;
9
10 end Data_Processing;
```

Listing 6: `data_processing-calculate.ads`

```
1 private
2 procedure Data_Processing.Calculate
3 (D : in out Data);
```

Listing 7: `data_processing-calculate.adb`

```
1 procedure Data_Processing.Calculate
2 (D : in out Data)
3 is
4 begin
5     -- Dummy implementation...
```

(continues on next page)

¹⁸² <http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html>

¹⁸³ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html>

(continued from previous page)

```
6   D.F := 0.0;
7   end Data_Processing.Calculate;
```

Listing 8: test_data_processing.adb

```
1   with Data_Processing; use Data_Processing;
2
3   procedure Test_Data_Processing is
4     D : Data;
5   begin
6     Process (D);
7   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.

13.1.2 Private subprograms and private packages

We can also use private subprograms to test private packages. As we know, in most cases, we cannot access private packages in external clients — such as external subprograms. However, by declaring a subprogram private, we're allowed to access private packages. This can be very useful to create applications that we can use to test private packages. (Note that these applications must be library-level parameterless subprograms, because only those can be main programs.)

Let's see an example:

Listing 9: private_data_processing.ads

```
1   private package Private_Data_Processing is
2
3     type Data is private;
4
5     procedure Process (D : in out Data);
6
7   private
8
9     type Data is record
10      F : Float;
11    end record;
12
13   end Private_Data_Processing;
```

Listing 10: private_data_processing.adb

```
1   package body Private_Data_Processing is
2
3     procedure Process (D : in out Data) is
4     begin
```

(continues on next page)

(continued from previous page)

```
5     D.F := 0.0;
6     end Process;
7
8 end Private_Data_Processing;
```

Listing 11: test_private_data_processing.ads

```
1 private procedure Test_Private_Data_Processing;
```

Listing 12: test_private_data_processing.adb

```
1 with Private_Data_Processing;
2 use Private_Data_Processing;
3
4 procedure Test_Private_Data_Processing is
5     D : Data;
6 begin
7     Process (D);
8 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

```
1 private package Private_Data_Processing is
2
3     type Data is private;
4
5     procedure Process (D : in out Data);
6
7 private
8
9     type Data is record
10        F : Float;
11    end record;
12
13 end Private_Data_Processing;
```

Listing 14: private_data_processing.adb

```
1 package body Private_Data_Processing is
2
3     procedure Process (D : in out Data) is
4     begin
5         null;
6     end Process;
7
8 end Private_Data_Processing;
```

Listing 15: private_data_processing-test.ads

```
1 procedure Private_Data_Processing.Test;
```

Listing 16: private_data_processing-test.adb

```
1 procedure Private_Data_Processing.Test is
2     D : Data := (F => 0.0);
3 begin
4     Process (D);
5 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

ACCESS TYPES

We discussed access types back in the [Introduction to Ada course](#)¹⁸⁴. In this chapter, we discuss further details about access types and techniques when using them. Before we dig into details, however, we're going to make sure we understand the terminology.

14.1 Access types: Terminology

In this section, we discuss some of the terminology associated with access types. Usually, the terms used in Ada when discussing references and dynamic memory allocation are different than the ones you might encounter in other languages, so it's necessary you understand what each term means.

14.1.1 Access type, designated subtype and profile

The first term we encounter is (obviously) *access type*, which is a type that provides us access to an object or a subprogram. We declare access types by using the **access** keyword:

Listing 1: show_access_type_declaration.ads

```
1 package Show_Access_Type_Declaration is
2
3   --
4   --   Declaring access types:
5   --
6
7   --   Access-to-object type
8   type Integer_Access is access Integer;
9
10  --   Access-to-subprogram type
11  type Init_Integer_Access is access
12     function return Integer;
13
14 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 566)).

¹⁸⁴ https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-types-overview

In the declaration of an access type, we always specify — after the **access** keyword — the kind of thing we want to designate. In the case of an access-to-object type declaration, we declare a subtype we want to access, which is known as the *designated subtype* of an access type. In the case of an access-to-subprogram type declaration, the subprogram prototype is known as the *designated profile*.

In our previous code example, **Integer** is the designated subtype of the `Integer_Access` type, and **function return Integer** is the designated profile of the `Init_Integer_Access` type.

Important

In contrast to other programming languages, an access type is not a pointer, and it doesn't just indicate an address in memory. We discuss more about *addresses* (page 595) later on.

14.1.2 Access object and designated object

We use an access-to-object type by first declaring a variable (or constant) of an access type and then allocating an object. (This is actually just one way of using access types; we discuss other methods later in this chapter.) The actual variable or constant of an access type is called *access object*, while the object we allocate (via **new**) is the *designated object*.

For example:

Listing 2: `show_simple_allocation.adb`

```
1 procedure Show_Simple_Allocation is
2
3   -- Access-to-object type
4   type Integer_Access is access Integer;
5
6   -- Access object
7   I1 : Integer_Access;
8
9 begin
10  I1 := new Integer;
11  --      ^^^^^^^^^^^^^ allocating an object,
12  --                    which becomes the designated
13  --                    object for I1
14
15 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.

14.1.3 Access value and designated value

An access object and a designated (allocated) object, both store values. The value of an access object is the *access value* and the value of a designated object is the *designated value*. For example:

Listing 3: show_values.adb

```

1 procedure Show_Values is
2
3   -- Access-to-object type
4   type Integer_Access is access Integer;
5
6   I1, I2, I3 : Integer_Access;
7
8 begin
9   I1 := new Integer;
10  I3 := new Integer;
11
12  -- Copying the access value of I1 to I2
13  I2 := I1;
14
15  -- Copying the designated value of I1
16  I3.all := I1.all;
17
18 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 512).)

In the Ada Reference Manual

- 3.10 Access Types¹⁸⁵

14.2 Access types: Allocation

Ada makes the distinction between pool-specific and general access types, as we'll discuss in this section. Before doing so, however, let's talk about memory allocation.

In general terms, memory can be allocated dynamically on the heap or statically on the stack. (Strictly speaking, both are dynamic allocations, in that they occur at run-time with amounts not previously specified.) For example:

Listing 4: show_simple_allocation.adb

```

1 procedure Show_Simple_Allocation is
2
3   -- Declaring access type:
4   type Integer_Access is access Integer;
```

(continues on next page)

¹⁸⁵ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

5
6  -- Declaring access object:
7  A1 : Integer_Access;
8
9  begin
10  -- Allocating an Integer object on the heap
11  A1 := new Integer;
12
13  declare
14  -- Allocating an Integer object on the
15  -- stack
16  I : Integer;
17  begin
18  null;
19  end;
20
21 end Show_Simple_Allocation;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↳Allocation.Simple_Allocation
MD5: 4144feb99e6e0b1a0749fce0b20370a1

```

Build output

```

show_simple_allocation.adb:16:07: warning: variable "I" is never read and never
↳assigned [-gnatwv]

```

When we allocate an object on the heap via **new**, the allocation happens in a memory pool that is associated with the access type. In our code example, there's a memory pool associated with the `Integer_Access` type, and each **new Integer** allocates a new integer object in that pool. Therefore, access types of this kind are called pool-specific access types. (We discuss *more about these types* (page 486) later.)

It is also possible to access objects that were allocated on the stack. To do that, however, we cannot use pool-specific access types because — as the name suggests — they're only allowed to access objects that were allocated in the specific pool associated with the type. Instead, we have to use general access types in this case:

Listing 5: show_general_access_type.adb

```

1  procedure Show_General_Access_Type is
2
3  -- Declaring general access type:
4  type Integer_Access is access all Integer;
5
6  -- Declaring access object:
7  A1 : Integer_Access;
8
9  -- Allocating an Integer object on the
10  -- stack:
11  I : aliased Integer;
12
13  begin
14  -- Getting access to an Integer object that
15  -- was allocated on the stack
16  A1 := I'Access;
17
18  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 525) 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`

```
1 procedure Show_Simple_Allocation is
2
3   -- Declaring general access type:
4   type Integer_Access is access all Integer;
5
6   -- Declaring access object:
7   A1 : Integer_Access;
8
9 begin
10  --
11  -- Allocating an Integer object on the heap
12  -- and initializing an access object of
13  -- the general access type Integer_Access.
14  --
15  A1 := new Integer;
16
17 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](#)¹⁸⁶

¹⁸⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

14.2.1 Pool-specific access types

We've already discussed many aspects about pool-specific access types. In this section, we recapitulate some of those aspects, and discuss some new details that haven't seen yet.

As we know, we cannot directly assign an object `Distance_Miles` of type `Miles` to an object `Distance_Meters` of type `Meters`, even if both share a common **Float** type ancestor. The assignment is only possible if we perform a type conversion from `Miles` to `Meters`, or vice-versa — e.g.: `Distance_Meters := Meters (Distance_Miles) * Miles_To_Meters_Factor`.

Similarly, in the case of pool-specific access types, a direct assignment between objects of different access types isn't possible. However, even if both access types have the same designated subtype (let's say, they are both declared using **is access Integer**), it's still not possible to perform a type conversion between those access types. The only situation when an access type conversion is allowed is when both types have a common ancestor.

Let's see an example:

Listing 7: `show_simple_allocation.adb`

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Simple_Allocation is
6
7      -- Declaring pool-specific access type:
8      type Integer_Access_1 is access Integer;
9      type Integer_Access_2 is access Integer;
10     type Integer_Access_2B is new Integer_Access_2;
11
12     -- Declaring access object:
13     A1 : Integer_Access_1;
14     A2 : Integer_Access_2;
15     A2B : Integer_Access_2B;
16
17 begin
18     A1 := new Integer;
19     Put_Line ("A1 : " & A1'Image);
20     Put_Line ("Pool: " & A1'Storage_Pool'Image);
21
22     A2 := new Integer;
23     Put_Line ("A2:  " & A2'Image);
24     Put_Line ("Pool: " & A2'Storage_Pool'Image);
25
26     -- ERROR: Cannot directly assign access values
27     --         for objects of unrelated access
28     --         types; also, cannot convert between
29     --         these types.
30     --
31     -- A1 := A2;
32     -- A1 := Integer_Access_1 (A2);
33
34     A2B := Integer_Access_2B (A2);
35     Put_Line ("A2B: " & A2B'Image);
36     Put_Line ("Pool: " & A2B'Storage_Pool'Image);
37
38 end Show_Simple_Allocation;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_Allocation.Pool_Specific_Access_Types
 MD5: 8984cb9cb9083f09b9b4096da812f805

Runtime output

```
A1 : (access ec12a0)
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL' {SYSTEM.STORAGE_POOLS.TROOT_
↳STORAGE_POOLC object}
A2: (access ec1360)
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL' {SYSTEM.STORAGE_POOLS.TROOT_
↳STORAGE_POOLC object}
A2B: (access ec1360)
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](#)¹⁸⁷:
 - an access type can be unconstrained, but the actual object allocation must be constrained;
 - we can use a *qualified expression* (page 61) to allocate an object.
- We can use the `Storage_Size` attribute to limit the size of the memory pool associated with an access type, as discussed previously in the [section about storage size](#) (page 84).
- When running out of memory while allocating via `new`, we get a `Storage_Error` exception because of the *storage check* (page 414).

For example:

Listing 8: `show_array_allocation.adb`

```
1 pragma Ada_2022;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 procedure Show_Array_Allocation is
6
7     -- Unconstrained array type:
8     type Integer_Array is
9         array (Positive range <>) of Integer;
10
11     -- Access type with unconstrained
12     -- designated subtype and limited storage
13     -- size.
14     type Integer_Array_Access is
15         access Integer_Array
```

(continues on next page)

¹⁸⁷ https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-type-allocation-constraints

(continued from previous page)

```

16     with Storage_Size => 128;
17
18     -- An access object:
19     A1 : Integer_Array_Access;
20
21     procedure Show_Info
22     (IAA : Integer_Array_Access) is
23     begin
24         Put_Line ("Allocated: " & IAA'Image);
25         Put_Line ("Length: "
26                 & IAA.all'Length'Image);
27         Put_Line ("Values: "
28                 & IAA.all'Image);
29     end Show_Info;
30
31 begin
32     -- Allocating an integer array with
33     -- constrained range on the heap:
34     A1 := new Integer_Array (1 .. 3);
35     A1.all := [others => 42];
36     Show_Info (A1);
37
38     -- Allocating an integer array on the
39     -- heap using a qualified expression:
40     A1 := new Integer_Array'(5, 10);
41     Show_Info (A1);
42
43     -- A third allocation fails at run time
44     -- because of the constrained storage
45     -- size:
46     A1 := new Integer_Array (1 .. 100);
47     Show_Info (A1);
48
49 exception
50     when Storage_Error =>
51         Put_Line ("Out of memory!");
52
53 end Show_Array_Allocation;

```

14.2.2 Multiple allocation

Up to now, we have seen examples of allocating a single object on the heap. It's possible to allocate multiple objects *at once* as well — i.e. syntactic sugar is available to simplify the code that performs this allocation. For example:

Listing 9: show_access_array_allocation.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Access_Array_Allocation is
6
7      type Integer_Access is access Integer;
8
9      type Integer_Access_Array is
10         array (Positive range <>) of Integer_Access;
11

```

(continues on next page)

(continued from previous page)

```

12  -- An array of access objects:
13  Arr : Integer_Access_Array (1 .. 10);
14
15  begin
16  --
17  -- Allocating 10 access objects and
18  -- initializing the corresponding designated
19  -- object with zero:
20  --
21  Arr := (others => new Integer'(0));
22
23  -- Same as:
24  for I in Arr'Range loop
25    Arr (I) := new Integer'(0);
26  end loop;
27
28  Put_Line ("Arr: " & Arr'Image);
29
30  Put_Line ("Arr (designated values): ");
31  for E of Arr loop
32    Put (E.all'Image);
33  end loop;
34
35  end Show_Access_Array_Allocation;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
Allocation.Integer_Access_Array
MD5: c32af182dc35879d76df989a689ee35c

```

Runtime output

```

Arr:
[(access 12453e0), (access 1245400), (access 1245420), (access 1245440),
 (access 1245460), (access 1245480), (access 12454a0), (access 12454c0),
 (access 12454e0), (access 1245500)]
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 182) is syntactic sugar for a loop over `Arr` that allocates each component. (Note that, by writing `Arr := (others => new Integer'(0))`, we're also initializing the designated objects with zero.)

Let's see another code example, this time with task types:

Listing 10: workers.ads

```

1  package Workers is
2
3    task type Worker is
4      entry Start (Id : Positive);
5      entry Stop;
6    end Worker;
7
8    type Worker_Access is access Worker;
9

```

(continues on next page)

(continued from previous page)

```
10  type Worker_Array is
11      array (Positive range <>) of Worker_Access;
12
13  end Workers;
```

Listing 11: workers.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Workers is
4
5      task body Worker is
6          Id : Positive;
7      begin
8          accept Start (Id : Positive) do
9              Worker.Id := Id;
10             end Start;
11             Put_Line ("Started Worker #"
12                 & Id'Image);
13
14             accept Stop;
15
16             Put_Line ("Stopped Worker #"
17                 & Id'Image);
18         end Worker;
19
20     end Workers;
```

Listing 12: show_workers.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Workers; use Workers;
4
5  procedure Show_Workers is
6      Worker_Arr : Worker_Array (1 .. 20);
7  begin
8      --
9      --  Allocating 20 workers at once:
10     --
11     Worker_Arr := (others => new Worker);
12
13     for I in Worker_Arr'Range loop
14         Worker_Arr (I).Start (I);
15     end loop;
16
17     Put_Line ("Some processing...");
18     delay 1.0;
19
20     for W of Worker_Arr loop
21         W.Stop;
22     end loop;
23
24 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 # 4
Started Worker # 2
Started Worker # 3
Started Worker # 5
Started Worker # 6
Started Worker # 7
Started Worker # 8
Started Worker # 9
Started Worker # 10
Started Worker # 11
Started Worker # 12
Started Worker # 13
Started Worker # 14
Started Worker # 15
Started Worker # 16
Started Worker # 17
Started Worker # 18
Started Worker # 19
Started Worker # 20
Some processing...
Stopped Worker # 5
Stopped Worker # 2
Stopped Worker # 1
Stopped Worker # 3
Stopped Worker # 4
Stopped Worker # 6
Stopped Worker # 7
Stopped Worker # 15
Stopped Worker # 16
Stopped Worker # 8
Stopped Worker # 17
Stopped Worker # 9
Stopped Worker # 12
Stopped Worker # 10
Stopped Worker # 14
Stopped Worker # 13
Stopped Worker # 19
Stopped Worker # 11
Stopped Worker # 18
Stopped Worker # 20
```

In this example, we declare the task type `Worker`, the access type `Worker_Access` and an array of access to tasks `Worker_Array`. Using this approach, a task is only created when we allocate an individual component of an array of `Worker_Array` type. Thus, when we declare the `Worker_Arr` array in this example, we're only preparing a *container* of 20 workers, but we don't have any actual tasks yet. We bring the 20 tasks into existence by writing `Worker_Arr := (others => new Worker)`.

14.3 Discriminants as Access Values

We can use access types when declaring discriminants. Let's see an example:

Listing 13: custom_rec.ads

```

1 package Custom_Recs is
2
3   -- Declaring an access type:
4   type Integer_Access is access Integer;
5
6   -- Declaring a discriminant with this
7   -- access type:
8   type Rec (IA : Integer_Access) is record
9
10      I : Integer := IA.all;
11      -- ^^^^^^^^^
12      -- Setting I's default to use the
13      -- designated value of IA:
14   end record;
15
16   procedure Show (R : Rec);
17
18 end Custom_Recs;
```

Listing 14: custom_rec.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Recs is
4
5   procedure Show (R : Rec) is
6   begin
7     Put_Line ("R.IA = "
8              & Integer'Image (R.IA.all));
9     Put_Line ("R.I = "
10             & Integer'Image (R.I));
11   end Show;
12
13 end Custom_Recs;
```

Listing 15: show_discriminants_as_access_values.adb

```

1 with Custom_Recs; use Custom_Recs;
2
3 procedure Show_Discriminants_As_Access_Values is
4
5   IA : constant Integer_Access :=
6       new Integer'(10);
7   R : Rec (IA);
8
9   begin
10    Show (R);
11
12    IA.all := 20;
13    R.I := 30;
14    Show (R);
15
16    -- As expected, we cannot change the
17    -- discriminant. The following line is
18    -- triggers a compilation error:
```

(continues on next page)

(continued from previous page)

```

19  --
20  -- R.IA := new Integer;
21
22  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

```

1  package Custom_Recs is
2
3     type Integer_Access is access Integer;
4
5     type Rec (IA : Integer_Access
6               := new Integer'(0)) is
7         --
8         --
9     record
10        I : Integer := IA.all;
11    end record;
12
13    procedure Show (R : Rec);
14
15  end Custom_Recs;

```

Listing 17: show_discriminants_as_access_values.adb

```

1  with Custom_Recs; use Custom_Recs;
2
3  procedure Show_Discriminants_As_Access_Values is
4
5     R1 : Rec;
6     --
7     -- no discriminant: use default
8
9     R2 : Rec (new Integer'(20));
10    --
11    -- allocating an unnamed integer object
12
13  begin
14    Show (R1);

```

(continues on next page)

(continued from previous page)

```
15 Show (R2);
16 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](#)¹⁸⁸
- [3.7.1 Discriminant Constraints](#)¹⁸⁹

14.3.1 Unconstrained type as designated subtype

Notice that we were using a scalar type as the designated subtype of the Integer_Access type. We could have used an unconstrained type as well. In fact, this is often used for the sake of having the effect of an unconstrained discriminant type.

Let's see an example:

Listing 18: persons.ads

```
1 package Persons is
2
3   -- Declaring an access type whose
4   -- designated subtype is unconstrained:
5   type String_Access is access String;
6
7   -- Declaring a discriminant with this
8   -- access type:
9   type Person (Name : String_Access) is record
10    Age : Integer;
11  end record;
12
13  procedure Show (P : Person);
14
15 end Persons;
```

¹⁸⁸ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

¹⁸⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html>

Listing 19: persons.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Persons is
4
5     procedure Show (P : Person) is
6     begin
7         Put_Line ("Name = "
8                 & P.Name.all);
9         Put_Line ("Age = "
10                & Integer'Image (P.Age));
11     end Show;
12
13 end Persons;
```

Listing 20: show_person.adb

```

1 with Persons; use Persons;
2
3 procedure Show_Person is
4     P : Person (new String'("John"));
5 begin
6     P.Age := 30;
7     Show (P);
8 end Show_Person;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
 ↪ Access_Values.Persons
 MD5: 9b1109d076b6f06632c8685a41616210

Runtime output

```
Name = John
Age  = 30
```

In this example, the discriminant of the Person type has an unconstrained designated type. In the Show_Person procedure, we declare the P object and specify the constraints of the allocated string object — in this case, a four-character string initialized with the name "John".

For further reading...

In the previous code example, we used an array — actually, a string — to demonstrate the advantage of using discriminants as access values, for we can use an unconstrained type as the designated subtype. In fact, as we discussed *earlier in another chapter* (page 24), we can only use discrete types (or access types) as discriminants. Therefore, you wouldn't be able to use a string, for example, directly as a discriminant without using access types:

Listing 21: persons.ads

```

1 package Persons is
2
3     -- ERROR: Declaring a discriminant with an
4     --           unconstrained type:
5     type Person (Name : String) is record
6         Age : Integer;
7     end record;
8
```

(continues on next page)

(continued from previous page)

```
9 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 generalize this to:

Listing 22: gen_custom_rec.ads

```
1 generic
2   type T (<>); -- any type
3   type T_Access is access T;
4 package Gen_Custom_Recs is
5   -- Declare a type whose discriminant D can
6   -- access any type:
7   type T_Rec (D : T_Access) is null record;
8 end Gen_Custom_Recs;
```

Listing 23: custom_rec.ads

```
1 with Gen_Custom_Recs;
2
3 package Custom_Recs is
4
5   type Incomp;
6   -- Incomplete type declaration!
7
8   type Incomp_Access is access Incomp;
9
10  -- Instantiating package using
11  -- incomplete type Incomp:
12  package Inst is new
13    Gen_Custom_Recs
14    (T => Incomp,
15     T_Access => Incomp_Access);
16  subtype Rec is Inst.T_Rec;
17
18  -- At this point, Rec (Inst.T_Rec) uses
19  -- an incomplete type as the designated
20  -- subtype of its discriminant type
21
22  procedure Show (R : Rec) is null;
23
24  -- Now, we complete the Incomp type:
25  type Incomp (B : Boolean := True) is private;
26
27 private
28  -- Finally, we have the full view of the
```

(continues on next page)

(continued from previous page)

```

29  -- Incomp type:
30  type Incomp (B : Boolean := True) is
31      null record;
32
33  end Custom_Recs;

```

Listing 24: show_rec.adb

```

1  with Custom_Recs; use Custom_Recs;
2
3  procedure Show_Rec is
4      R : Rec (new Incomp);
5  begin
6      Show (R);
7  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 615), which provide some advantages in terms of *accessibility rules* (page 534).

14.3.2 Whole object assignments

As expected, we cannot change the discriminant value in whole object assignments. If we do that, the Constraint_Error exception is raised at runtime:

Listing 25: show_person.adb

```

1  with Persons; use Persons;
2
3  procedure Show_Person is
4      S1 : String_Access := new String ("John");
5      S2 : String_Access := new String ("Mark");
6      P : Person := (Name => S1,
7                    Age  => 30);
8
9  begin
10     P := (Name => S1, Age => 31);
11     --      ^^ OK: we didn't change the
12     --      discriminant.
13     Show (P);
14
15     -- We can just repeat the discriminant:
16     P := (Name => P.Name, Age => 32);
17     --      ^^^^^^ OK: we didn't change the
18     --      discriminant.
19     Show (P);
20
21     -- Of course, we can change the string itself:
22     S1.all := "Mark";

```

(continues on next page)

(continued from previous page)

```
22 Show (P);
23
24 P := (Name => S2, Age => 40);
25 --      ^^ ERROR: we changed the
26 --      discriminant!
27 Show (P);
28 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.

14.4 Parameters as Access Values

In addition to *using discriminants as access values* (page 492), we can use access types for subprogram formal parameters. For example, the N parameter of the Show procedure below has an access type:

Listing 26: names.ads

```
1 package Names is
2
3   type Name is access String;
4
5   procedure Show (N : Name);
6
7 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

```
1 package Names is
2
3   type Name is access String;
```

(continues on next page)

(continued from previous page)

```
4
5  procedure Show (N : Name);
6
7  end Names;
```

Listing 28: names.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Names is
4
5      procedure Show (N : Name) is
6      begin
7          Put_Line ("Name: " & N.all);
8      end Show;
9
10 end Names;
```

Listing 29: show_names.adb

```
1  with Names; use Names;
2
3  procedure Show_Names is
4      N : Name := new String'("John");
5  begin
6      Show (N);
7  end Show_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↳ Access_Values.Names
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

```
1  package Names is
2
3      type Name is access String;
4
5      procedure Show (N : String);
6
7  end Names;
```

Listing 31: names.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Names is
4
5      procedure Show (N : String) is
6      begin
```

(continues on next page)

(continued from previous page)

```
7     Put_Line ("Name: " & N);
8     end Show;
9
10    end Names;
```

Listing 32: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4     N : Name := new String ("John");
5     begin
6         Show (N.all);
7     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.

14.4.1 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

```
1 package Names is
2
3     type Name is access String;
4
5     procedure Lower_And_Show (N : Name);
6
7 end Names;
```

Listing 34: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Characters.Handling;
4 use Ada.Characters.Handling;
5
```

(continues on next page)

(continued from previous page)

```

6 package body Names is
7
8   procedure Lower_And_Show (N : Name) is
9   begin
10    for I in N'Range loop
11      N (I) := To_Lower (N (I));
12    end loop;
13    Put_Line ("Name: " & N.all);
14  end Lower_And_Show;
15
16 end Names;
```

Listing 35: show_changed_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Changed_Names is
4   N : Name := new String ("John");
5 begin
6   Lower_And_Show (N);
7 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

```

1 package Names is
2
3   type Name is access String;
4
5   procedure Lower_And_Show (N : in out String);
6
7 end Names;
```

Listing 37: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Characters.Handling;
4 use Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Lower_And_Show (N : in out String) is
9   begin
10    for I in N'Range loop
11      N (I) := To_Lower (N (I));
12    end loop;
13    Put_Line ("Name: " & N);
```

(continues on next page)

(continued from previous page)

```
14   end Lower_And_Show;
15
16 end Names;
```

Listing 38: show_changed_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Changed_Names is
4   N : Name := new String'("John");
5 begin
6   Lower_And_Show (N.all);
7 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

14.4.2 Replace the access value

Instead of changing the object in the `Lower_And_Show` procedure, we could replace the access value by another one — for example, by allocating a new string inside the procedure. In this case, we have to pass the access value by reference using the `in out` parameter mode:

Listing 39: names.ads

```
1 package Names is
2
3   type Name is access String;
4
5   procedure Lower_And_Show (N : in out Name);
6
7 end Names;
```

Listing 40: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Characters.Handling;
4 use Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Lower_And_Show (N : in out Name) is
9   begin
10    N := new String'(To_Lower (N.all));
11    Put_Line ("Name: " & N.all);
12   end Lower_And_Show;
13
14 end Names;
```

Listing 41: show_changed_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Changed_Names is
4   N : Name := new String("John");
5 begin
6   Lower_And_Show (N);
7 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 546), as explained later on.

14.4.3 Side-effects on designated objects

In previous code examples from this section, we've seen that passing a parameter by reference using the **in** or **in out** parameter modes is an alternative to using access values as parameters. Let's focus on the subprogram declarations of those code examples and their parameter modes:

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 526), which we discuss later on. In this case, we're basically saying that the object we're accessing in Show is constant, so we cannot possibly change it:

Listing 42: names.ads

```
1 package Names is
2
3     type Name is access String;
4
5     type Constant_Name is access constant String;
6
7     procedure Show (N : Constant_Name);
8
9 end Names;
```

Listing 43: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Show (N : Constant_Name) is
9     begin
10         -- for I in N'Range loop
11         --     N (I) := To_Lower (N (I));
12         -- end loop;
13         Put_Line ("Name: " & N.all);
14     end Show;
15
16 end Names;
```

Listing 44: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4     N : Name := new String'("John");
5 begin
6     Show (Constant_Name (N));
7 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

```

1 package Names is
2
3   type Name is access String;
4
5   procedure Show (N : Name)
6     with Post => N.all'Old = N.all;
7     ~~~~~
8   --   we promise that we won't change
9   --   the object
10
11 end Names;
```

Listing 46: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Show (N : Name) is
9     begin
10      -- for I in N'Range loop
11      --   N (I) := To_Lower (N (I));
12      -- end loop;
13      Put_Line ("Name: " & N.all);
14     end Show;
15
16 end Names;
```

Listing 47: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4   N : Name := new String ("John");
5 begin
6   Show (N);
7 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 sub-

program call. Unfortunately, we cannot use this attribute when dealing with *limited private types* (page 677) — or limited types in general. For example, let's change the declaration of `Name` and have it as a limited private type instead:

Listing 48: names.ads

```
1 package Names is
2
3     type Name is limited private;
4
5     function Init (S : String) return Name;
6
7     function Equal (N1, N2 : Name)
8         return Boolean;
9
10    procedure Show (N : Name)
11        with Post => Equal (N'Old = N);
12
13 private
14
15    type Name is access String;
16
17    function Init (S : String) return Name is
18        (new String'(S));
19
20    function Equal (N1, N2 : Name)
21        return Boolean is
22        (N1.all = N2.all);
23
24 end Names;
```

Listing 49: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Show (N : Name) is
9     begin
10        -- for I in N'Range loop
11        --     N (I) := To_Lower (N (I));
12        -- end loop;
13        Put_Line ("Name: " & N.all);
14    end Show;
15
16 end Names;
```

Listing 50: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4     N : Name := Init ("John");
5 begin
6     Show (N);
7 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

```

1 package Names is
2
3     type Name is limited private;
4
5     type Constant_Name is limited private;
6
7     function Init (S : String) return Name;
8
9     function To_Constant_Name
10        (N : Name)
11        return Constant_Name;
12
13     procedure Show (N : Constant_Name);
14
15 private
16
17     type Name is
18         access String;
19
20     type Constant_Name is
21         access constant String;
22
23     function Init (S : String) return Name is
24         (new String'(S));

```

(continues on next page)

(continued from previous page)

```
25
26   function To_Constant_Name
27     (N : Name)
28     return Constant_Name is
29     (Constant_Name (N));
30
31 end Names;
```

Listing 52: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Show (N : Constant_Name) is
9   begin
10    -- for I in N'Range loop
11    --   N (I) := To_Lower (N (I));
12    -- end loop;
13    Put_Line ("Name: " & N.all);
14 end Show;
15
16 end Names;
```

Listing 53: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4   N : Name := Init ("John");
5 begin
6   Show (To_Constant_Name (N));
7 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 509).

14.5 Self-reference

As we've discussed in the section about incomplete types <Adv_Ada_Incomplete_Types>, we can use incomplete types to create a recursive, self-referencing type. Let's revisit a code example from that section:

Listing 54: linked_list_example.ads

```

1 package Linked_List_Example is
2
3     type Integer_List;
4
5     type Next is access Integer_List;
6
7     type Integer_List is record
8         I : Integer;
9         N : Next;
10    end record;
11
12 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

```

1 generic
2     type T is private;
3 package Linked_Lists is
4
5     type List is limited private;
6
7     procedure Append_Front
8         (L : in out List;
9          E : T);
10
11    procedure Append_Rear
12        (L : in out List;
13         E : T);
14
15    procedure Show (L : List);
16
17 private
18
19    -- Incomplete type declaration:
20    type Component;
21
22    -- Using incomplete type:
```

(continues on next page)

(continued from previous page)

```

23  type List is access Component;
24
25  type Component is record
26      Value : T;
27      Next  : List;
28      --      ^^^^
29      --      Self-reference via access type
30  end record;
31
32  end Linked_Lists;

```

Listing 56: linked_lists.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  package body Linked_Lists is
6
7      procedure Append_Front
8          (L : in out List;
9           E :          T)
10     is
11         New_First : constant List := new
12             Component'(Value => E,
13                       Next  => L);
14     begin
15         L := New_First;
16     end Append_Front;
17
18     procedure Append_Rear
19         (L : in out List;
20          E :          T)
21     is
22         New_Last : constant List := new
23             Component'(Value => E,
24                       Next  => null);
25     begin
26         if L = null then
27             L := New_Last;
28         else
29             declare
30                 Last : List := L;
31             begin
32                 while Last.Next /= null loop
33                     Last := Last.Next;
34                 end loop;
35                 Last.Next := New_Last;
36             end;
37         end if;
38     end Append_Rear;
39
40     procedure Show (L : List) is
41         Curr : List := L;
42     begin
43         if L = null then
44             Put_Line ("[ ]");
45         else
46             Put ("[");
47             loop
48                 Put (Curr.Value'Image);

```

(continues on next page)

(continued from previous page)

```

49         Put (" ");
50         exit when Curr.Next = null;
51         Curr := Curr.Next;
52     end loop;
53     Put_Line ("]");
54 end if;
55 end Show;
56
57 end Linked_Lists;

```

Listing 57: test_linked_list.adb

```

1  with Linked_Lists;
2
3  procedure Test_Linked_List is
4      package Integer_Lists is new
5          Linked_Lists (T => Integer);
6      use Integer_Lists;
7
8      L : List;
9  begin
10     Append_Front (L, 3);
11     Append_Rear (L, 4);
12     Append_Rear (L, 5);
13     Append_Front (L, 2);
14     Append_Front (L, 1);
15     Append_Rear (L, 6);
16     Append_Rear (L, 7);
17
18     Show (L);
19 end Test_Linked_List;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Self_Reference.
↳Linked_List_Example
MD5: 8af1ff7bbda44362802ba4f93b9c5741

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 498) for the Append_Front, Append_Rear and Show procedures.

In the Ada Reference Manual

- [3.10.1 Incomplete Type Declarations](#)¹⁹⁰

¹⁹⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html>

14.6 Mutually dependent types using access types

In the section on *mutually dependent types* (page 149), we've seen a code example where each type depends on the other one. We could rewrite that code example using access types:

Listing 58: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T2;
4     type T2_Access is access T2;
5
6     type T1 is record
7         B : T2_Access;
8     end record;
9
10    type T1_Access is access T1;
11
12    type T2 is record
13        A : T1_Access;
14    end record;
15
16 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.

14.7 Dereferencing

In the [Introduction to Ada course](#)¹⁹¹, we discussed the `.all` syntax to dereference access values:

Listing 59: show_dereferencing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Dereferencing is
4
5     -- Declaring access type:
6     type Integer_Access is access Integer;
7
8     -- Declaring access object:
9     A1 : Integer_Access;
10
11 begin
12     A1 := new Integer;
13
14     -- Dereferencing access value:
```

(continues on next page)

¹⁹¹ https://learn.adacore.com/courses/intro-to-ada/chapters/access_types.html#intro-ada-access-dereferencing

(continued from previous page)

```

15   A1.all := 22;
16
17   Put_Line ("A1: " & Integer'Image (A1.all));
18 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

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Dereferencing is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     type Integer_Array_Access is
9         access Integer_Array;
10
11     Arr : constant Integer_Array_Access :=
12         new Integer_Array (1 .. 6);
13 begin
14     Arr.all := (1, 2, 3, 5, 8, 13);
15
16     for I in Arr'Range loop
17         Put_Line ("Arr (: "
18             & Integer'Image (I) & "): "
19             & Integer'Image (Arr.all (I)));
20     end loop;
21 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
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 Names¹⁹²
-

14.7.1 Implicit Dereferencing

Implicit dereferencing allows us to omit the `.all` suffix without getting a compilation error. In this case, the compiler *knows* that the dereferenced object is implied, not the access value.

Ada supports implicit dereferencing in these use cases:

- when accessing components of a record or an array — including array slices.
- when accessing subprograms that have at least one parameter (we discuss this topic later in this chapter);
- when accessing some attributes — such as some array and task attributes.

Arrays

Let's start by looking into an example of implicit dereferencing of arrays. We can take the previous code example and replace `Arr.all (I)` by `Arr (I)`:

Listing 61: show_dereferencing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Dereferencing is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     type Integer_Array_Access is
9         access Integer_Array;
10
11     Arr : constant Integer_Array_Access :=
12         new Integer_Array (1 .. 6);
13 begin
14     Arr.all := (1, 2, 3, 5, 8, 13);
15
16     Arr (1 .. 6) := (1, 2, 3, 5, 8, 13);
17
18     for I in Arr'Range loop
19         Put_Line
20             ("Arr (:"
21              & Integer'Image (I) & "): "
22              & Integer'Image (Arr (I)));
23         -- ^ .all is implicit.
24     end loop;
25 end Show_Dereferencing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_Implicit_Dereferencing
MD5: ade602a9e6976018e0c00f930a2399f1

¹⁹² <http://www.ada-auth.org/standards/22rm/html/RM-4-1.html>

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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Array_Assignments is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     type Integer_Array_Access is
9         access Integer_Array;
10
11     procedure Show_Array
12         (Name : String;
13          Arr  : Integer_Array_Access) is
14     begin
15         Put (Name);
16         for E of Arr.all loop
17             Put (Integer'Image (E));
18         end loop;
19         New_Line;
20     end Show_Array;
21
22     Arr_1 : constant Integer_Array_Access :=
23             new Integer_Array (1 .. 6);
24     Arr_2 : Integer_Array_Access :=
25             new Integer_Array (1 .. 6);
26 begin
27     Arr_1.all := (1, 2, 3, 5, 8, 13);
28     Arr_2.all := (21, 34, 55, 89, 144, 233);
29
30     -- Array assignment
31     Arr_2.all := Arr_1.all;
32
33     Show_Array ("Arr_2", Arr_2);
34
```

(continues on next page)

(continued from previous page)

```
35  -- Access value assignment
36  Arr_2 := Arr_1;
37
38  Arr_1.all := (377, 610, 987, 1597, 2584, 4181);
39
40  Show_Array ("Arr_2", Arr_2);
41  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

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Dereferencing is
4
5      type Rec is record
6          I : Integer;
7          F : Float;
8      end record;
9
10     type Rec_Access is access Rec;
11
12     R : constant Rec_Access := new Rec;
13  begin
14     R.all := (I => 1, F => 5.0);
15
16     Put_Line ("R.I: "
17              & Integer'Image (R.I));
18     Put_Line ("R.F: "
19              & Float'Image (R.F));
20  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

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Dereferencing is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     type Integer_Array_Access is
9         access Integer_Array;
10
11     Arr : constant Integer_Array_Access :=
12         new Integer_Array (1 .. 6);
13 begin
14     Put_Line
15         ("Arr'First: "
16          & Integer'Image (Arr'First));
17     Put_Line
18         ("Arr'Last: "
19          & Integer'Image (Arr'Last));
20
21     Put_Line
22         ("Arr'Component_Size: "
23          & Integer'Image (Arr'Component_Size));
24     Put_Line
25         ("Arr.all'Component_Size: "
26          & Integer'Image (Arr.all'Component_Size));
27
28     Put_Line
29         ("Arr'Size: "
30          & Integer'Image (Arr'Size));
31     Put_Line
32         ("Arr.all'Size: "
33          & Integer'Image (Arr.all'Size));
34 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	<code>Arr.all (I)</code>	<code>Arr (I)</code>
Array slices	<code>Arr.all (F .. L)</code>	<code>Arr (F .. L)</code>
Record components	<code>Rec.all.C</code>	<code>Rec.C</code>
Array attributes	<code>Arr.all'First</code>	<code>Arr'First</code>
	<code>Arr.all'First (N)</code>	<code>Arr'First (N)</code>
	<code>Arr.all'Last</code>	<code>Arr'Last</code>
	<code>Arr.all'Last (N)</code>	<code>Arr'Last (N)</code>
	<code>Arr.all'Range</code>	<code>Arr'Range</code>
	<code>Arr.all'Range (N)</code>	<code>Arr'Range (N)</code>
	<code>Arr.all'Length</code>	<code>Arr'Length</code>
	<code>Arr.all'Length (N)</code>	<code>Arr'Length (N)</code>
	<code>Arr.all'Component_Size</code>	<code>Arr'Component_Size</code>
	Task attributes	<code>T.all'Identity</code>
<code>T.all'Storage_Size</code>		<code>T'Storage_Size</code>
<code>T.all'Terminated</code>		<code>T'Terminated</code>
<code>T.all'Callable</code>		<code>T'Callable</code>
Tagged type attributes	<code>X.all'Tag</code>	<code>X'Tag</code>
Other attributes	<code>X.all'Valid</code>	<code>X'Valid</code>
	<code>X.all'Old</code>	<code>X'Old</code>
	<code>A.all'Constrained</code>	<code>A'Constrained</code>

In the Ada Reference Manual

- [4.1 Names](#)¹⁹³
- [4.1.1 Indexed Components](#)¹⁹⁴
- [4.1.2 Slices](#)¹⁹⁵

¹⁹³ <http://www.ada-auth.org/standards/22rm/html/RM-4-1.html>

¹⁹⁴ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-1.html>

¹⁹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-2.html>

- 4.1.3 Selected Components¹⁹⁶
- 4.1.4 Attributes¹⁹⁷

14.8 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.

14.8.1 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`

```

1 package Data_Processing is
2
3   type Quality_Level is
4     (Simplified, Better, Best);
5
6 private
7
8   Calc_Table : constant array
9     (Quality_Level, 1 .. 5) of Float :=
10     (Simplified =>
11       (0.15, 0.00, 0.00, 0.00, 0.00),
12       Better   =>
13         (0.02, 0.16, 0.27, 0.00, 0.00),
14       Best     =>
15         (0.01, 0.08, 0.12, 0.20, 0.34));
16
17   Last : constant array
18     (Quality_Level) of Positive :=
19     (Simplified => 1,
20      Better    => 3,
21      Best      => 5);
22
23 end Data_Processing;
```

Code block metadata

¹⁹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-3.html>

¹⁹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-4.html>

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

```
1 package Data_Processing is
2
3   type Quality_Level is
4     (Simplified, Better, Best);
5
6   type Data is
7     array (Positive range <>) of Float;
8
9 private
10
11   type Table_Coefficient is record
12     Last : Positive;
13     Coef : Data (1 .. 5);
14   end record;
15
16   Calc_Table : constant array
17     (Quality_Level) of Table_Coefficient :=
18     (Simplified =>
19       (1, (0.15, 0.00, 0.00, 0.00, 0.00)),
20      Better    =>
21       (3, (0.02, 0.16, 0.27, 0.00, 0.00)),
22      Best      =>
23       (5, (0.01, 0.08, 0.12, 0.20, 0.34)));
24
25 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

```
1 package Data_Processing.Operations is
2
3   procedure Process (D : in out Data;
4                     Q :           Quality_Level);
5
6 end Data_Processing.Operations;
```

Listing 68: data_processing-operations.adb

```
1 package body Data_Processing.Operations is
2
```

(continues on next page)

(continued from previous page)

```

3  procedure Process (D : in out Data;
4                      Q :           Quality_Level) is
5  begin
6      for I in D'Range loop
7          for J in 1 .. Calc_Table (Q).Last loop
8              -- ... * Calc_Table (Q).Coef (J)
9              null;
10             end loop;
11             -- D (I) := ...
12             null;
13         end loop;
14     end Process;
15
16 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.

14.8.2 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

```

1  package Data_Processing is
2
3      type Integer_Array is
4          array (Positive range <>) of Integer;
5
6  private
7
8      type Integer_Array_Access is
9          access constant Integer_Array;
10
11     Table : constant array (1 .. 3) of
12         Integer_Array_Access :=
13         (1 => new Integer_Array'(1 => 15),
14          2 => new Integer_Array'(1 => 12,
15                                 2 => 15,
16                                 3 => 20),
17          3 => new Integer_Array'(1 => 12,
18                                 2 => 15,

```

(continues on next page)

(continued from previous page)

```
19         3 => 20,  
20         4 => 20,  
21         5 => 25,  
22         6 => 30));  
23  
24 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

```
1 package Data_Processing is  
2  
3   type Quality_Level is  
4     (Simplified, Better, Best);  
5  
6   type Data is  
7     array (Positive range <>) of Float;  
8  
9 private  
10  
11   type Coefficients is access constant Data;  
12  
13   Calc_Table : constant array (Quality_Level) of  
14     Coefficients :=  
15     (Simplified =>  
16       new Data'(1 => 0.15),  
17     Better      =>  
18       new Data'(0.02, 0.16, 0.27),  
19     Best        =>  
20       new Data'(0.01, 0.08, 0.12,  
21                 0.20, 0.34));  
22  
23 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 526) in more details later on.

This is the adapted Process procedure:

Listing 71: data_processing-operations.ads

```

1 package Data_Processing.Operations is
2
3   procedure Process (D : in out Data;
4                     Q :           Quality_Level);
5
6 end Data_Processing.Operations;
```

Listing 72: data_processing-operations.adb

```

1 package body Data_Processing.Operations is
2
3   procedure Process (D : in out Data;
4                     Q :           Quality_Level) is
5   begin
6     for I in D'Range loop
7       for J in Calc_Table (Q)'Range loop
8         -- ... * Calc_Table (Q).Coef (J)
9         null;
10      end loop;
11      -- D (I) := ...
12      null;
13    end loop;
14  end Process;
15
16 end Data_Processing.Operations;
```

Now, we can simply loop over the coefficients by writing `for J in Calc_Table (Q)'Range loop`, as each element of `Calc_Table` automatically has the correct range.

14.9 Aliasing

The term *aliasing*¹⁹⁸ 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

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliasing is
4   type Integer_Access is access Integer;
5
6   A1, A2 : Integer_Access;
7 begin
8   A1 := new Integer;
9   A2 := A1;
10
11  A1.all := 22;
12  Put_Line ("A1: " & Integer'Image (A1.all));
13  Put_Line ("A2: " & Integer'Image (A2.all));
14
15  A2.all := 24;
16  Put_Line ("A1: " & Integer'Image (A1.all));
17  Put_Line ("A2: " & Integer'Image (A2.all));
18 end Show_Aliasing;
```

¹⁹⁸ [https://en.wikipedia.org/wiki/Aliasing_\(computing\)](https://en.wikipedia.org/wiki/Aliasing_(computing))

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliasing_
↳Via_Access
MD5: 2fde6073ce9823a1a9d93aec82384e1
```

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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Renaming is
4   A1 : Integer;
5   A2 : Integer renames A1;
6 begin
7   A1 := 22;
8   Put_Line ("A1: " & Integer'Image (A1));
9   Put_Line ("A2: " & Integer'Image (A2));
10
11  A2 := 24;
12  Put_Line ("A1: " & Integer'Image (A1));
13  Put_Line ("A2: " & Integer'Image (A2));
14 end Show_Renaming;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Renaming
MD5: 99a47d02000b91f7464df994fd8ee6
```

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](#)¹⁹⁹

¹⁹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

14.9.1 Aliased objects

As we discussed *previously* (page 483), we use **new** to create aliased objects on the heap. We can also use general access types to access objects that were created on the stack.

By default, objects created on the stack aren't aliased. Therefore, we have to indicate that an object is aliased by using the **aliased** keyword in the object's declaration: `Obj : aliased Integer;`

Let's see an example:

Listing 75: show_aliased_obj.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Obj is
4   type Integer_Access is access all Integer;
5
6   I_Var : aliased Integer;
7   A1    : Integer_Access;
8 begin
9   A1 := I_Var'Access;
10
11  A1.all := 22;
12  Put_Line ("A1: " & Integer'Image (A1.all));
13 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

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Obj is
4   type Integer_Access is access all Integer;
5
6   I_Var : aliased Integer;
7   A1, A2 : Integer_Access;
8 begin
9   A1 := I_Var'Access;
10  A2 := A1;
11
12  A1.all := 22;
13  Put_Line ("A1: " & Integer'Image (A1.all));
14  Put_Line ("A2: " & Integer'Image (A2.all));
15
16  A2.all := 24;
17  Put_Line ("A1: " & Integer'Image (A1.all));
18  Put_Line ("A2: " & Integer'Image (A2.all));
19
20 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](#)²⁰⁰
- [3.10 Access Types](#)²⁰¹

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	<code>type T_Acc is access all T</code>
Access-to-constant	<code>type T_Acc is access constant T</code>

Let's look at an example:

Listing 77: integer_access_types.ads

```
1 package Integer_Access_Types is
2
3     type Integer_Access is
4       access Integer;
5
6     type Integer_Access_All is
7       access all Integer;
8
9     type Integer_Access_Const is
10      access constant Integer;
11
12 end Integer_Access_Types;
```

Code block metadata

²⁰⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-3-1.html>

²⁰¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

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

```

1 package Integer_Access_Types is
2
3     type Integer_Access is
4         access Integer;
5
6     type Integer_Access_All is
7         access all Integer;
8
9     type Integer_Access_Const is
10        access constant Integer;
11
12    procedure Show;
13
14 end Integer_Access_Types;
```

Listing 79: integer_access_types.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2
3 package body Integer_Access_Types is
4
5     I_Var : aliased Integer := 0;
6     Fact  : aliased constant Integer := 42;
7
8     Dyn_Ptr   : constant Integer_Access
9                 := new Integer'(30);
10    I_Var_Ptr  : constant Integer_Access_All
11                 := I_Var'Access;
12    I_Var_C_Ptr : constant Integer_Access_Const
13                 := I_Var'Access;
14    Fact_Ptr   : constant Integer_Access_Const
15                 := Fact'Access;
16
17    procedure Show is
18    begin
19        Put_Line ("Dyn_Ptr:      "
20                & Integer'Image (Dyn_Ptr.all));
21        Put_Line ("I_Var_Ptr:   "
22                & Integer'Image (I_Var_Ptr.all));
23        Put_Line ("I_Var_C_Ptr:  "
```

(continues on next page)

(continued from previous page)

```
24         & Integer'Image
25           (I_Var_C_Ptr.all));
26     Put_Line ("Fact_Ptr: "
27             & Integer'Image (Fact_Ptr.all));
28   end Show;
29
30 end Integer_Access_Types;
```

Listing 80: show_access_modifiers.adb

```
1 with Integer_Access_Types;
2
3 procedure Show_Access_Modifiers is
4 begin
5   Integer_Access_Types.Show;
6 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](#)²⁰²
-

Non-aliased objects

As mentioned earlier, by default, declared objects — which are allocated on the stack — aren't aliased. Therefore, we cannot get a reference to those objects. For example:

Listing 81: show_access_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Access_Error is
4   type Integer_Access is access all Integer;
5   I_Var : Integer;
6   A1    : Integer_Access;
7 begin
```

(continues on next page)

²⁰² <http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html>

(continued from previous page)

```

8   A1 := I_Var'Access;
9
10  A1.all := 22;
11  Put_Line ("A1: " & Integer'Image (A1.all));
12 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 519). For example, we can rewrite a previous program using aliased constant objects:

Listing 82: data_processing.ads

```

1 package Data_Processing is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6 private
7
8   type Integer_Array_Access is
9     access constant Integer_Array;
10
11  Tab_1 : aliased constant Integer_Array
12         := (1 => 15);
13  Tab_2 : aliased constant Integer_Array
14         := (12, 15, 20);
15  Tab_3 : aliased constant Integer_Array
16         := (12, 15, 20,
17            20, 25, 30);
18
19  Table : constant array (1 .. 3) of
20         Integer_Array_Access :=
21         (1 => Tab_1'Access,
22          2 => Tab_2'Access,
23          3 => Tab_3'Access);
24
25 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Ragged_
↳Array_Aliased_Obj
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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Access_Obj is
4
5     type Integer_Access is
6         access all Integer;
7     type Integer_Access_Access is
8         access all Integer_Access;
9
10    I_Var : aliased Integer;
11    A      : aliased Integer_Access;
12    B      : Integer_Access_Access;
13 begin
14    A := I_Var'Access;
15    B := A'Access;
16
17    B.all.all := 22;
18    Put_Line ("A: " & Integer'Image (A.all));
19    Put_Line ("B: " & Integer'Image (B.all.all));
20 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.

14.9.2 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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Components is
4
5     type Integer_Access is access all Integer;
6
7     type Rec is record
```

(continues on next page)

(continued from previous page)

```

8     I_Var_1 : Integer;
9     I_Var_2 : aliased Integer;
10    end record;
11
12    type Integer_Array is
13        array (Positive range <>) of aliased Integer;
14
15    R : Rec := (22, 24);
16    Arr : Integer_Array (1 .. 3) := (others => 42);
17    A : Integer_Access;
18 begin
19    -- A := R.I_Var_1'Access;
20    --           ^ ERROR: cannot access
21    --           non-aliased
22    --           component
23
24    A := R.I_Var_2'Access;
25    Put_Line ("A: " & Integer'Image (A.all));
26
27    A := Arr (2)'Access;
28    Put_Line ("A: " & Integer'Image (A.all));
29 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 give us a compilation error, as this component is not aliased.) Similarly, we get access to the second component of array `Arr`.

Declaring components with the **aliased** keyword allows us to specify that those are accessible via other paths besides the component name. Therefore, the compiler won't store them in registers. This can be essential when doing low-level programming — for example, when accessing memory-mapped registers. In this case, we want to ensure that the compiler uses the memory address we're specifying (instead of assigning registers for those components).

In the Ada Reference Manual

- [3.6 Array Types](#)²⁰³

²⁰³ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

14.9.3 Aliased parameters

In addition to aliased objects and components, we can declare *aliased parameters* (page 365), as we already discussed in an earlier chapter. As we mentioned there, aliased parameters are always passed by reference, independently of the type we're using.

The parameter mode indicates which type we must use for the access type:

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

```

1 package Data_Processing is
2
3   procedure Proc (I : aliased in out Integer);
4
5 end Data_Processing;
```

Listing 86: data_processing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Data_Processing is
4
5   procedure Show (I : aliased Integer) is
6     --      ^ equivalent to
7     --      "aliased in Integer"
8
9     type Integer_Constant_Access is
10      access constant Integer;
11
12     A : constant Integer_Constant_Access
13        := I'Access;
14   begin
15     Put_Line ("Value : I "
16              & Integer'Image (A.all));
17   end Show;
18
19   procedure Set_One (I : aliased out Integer) is
20
21     type Integer_Access is access all Integer;
22
23     procedure Local_Set_One (A : Integer_Access)
24       is
25       begin
26         A.all := 1;
27       end Local_Set_One;
28
29   begin
30     Local_Set_One (I'Access);
31   end Set_One;
32
33   procedure Proc (I : aliased in out Integer) is
34
```

(continues on next page)

(continued from previous page)

```

35     type Integer_Access is access all Integer;
36
37     procedure Add_One (A : Integer_Access) is
38     begin
39         A.all := A.all + 1;
40     end Add_One;
41
42     begin
43         Show (I);
44         Add_One (I'Access);
45         Show (I);
46     end Proc;
47
48 end Data_Processing;

```

Listing 87: show_aliased_param.adb

```

1  with Data_Processing; use Data_Processing;
2
3  procedure Show_Aliased_Param is
4      I : aliased Integer := 22;
5  begin
6      Proc (I);
7  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](#)²⁰⁴
- [6.2 Formal Parameter Modes](#)²⁰⁵
- [6.4.1 Parameter Associations](#)²⁰⁶

²⁰⁴ <http://www.ada-auth.org/standards/22rm/html/RM-6-1.html>

²⁰⁵ <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

²⁰⁶ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

14.10 Accessibility Levels and Rules: An Introduction

This section provides an introduction to accessibility levels and accessibility rules. This topic can be very complicated, and by no means do we intend to cover all the details here. (In fact, discussing all the details about accessibility levels and rules could be a long chapter on its own. If you're interested in them, please refer to the Ada Reference Manual.) In any case, the goal of this section is to present the intention behind the accessibility rules and build intuition on how to best use access types in your code.

In the Ada Reference Manual

- [3.10.2 Operations of Access Types](#)²⁰⁷

14.10.1 Lifetime of objects

First, let's talk a bit about [lifetime of objects](#)²⁰⁸. 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

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Lifetime is
4      I_Var_1 : Integer := 22;
5  begin
6
7      Inner_Block : declare
8          I_Var_2 : Integer := 42;
9      begin
10         Put_Line ("I_Var_1: "
11                 & Integer'Image (I_Var_1));
12         Put_Line ("I_Var_2: "
13                 & Integer'Image (I_Var_2));
14
15         -- I_Var_2 will get out of scope
16         -- when the block finishes.
17     end Inner_Block;
18
19     -- I_Var_2 is now out of scope...
20
21     Put_Line ("I_Var_1: "
22             & Integer'Image (I_Var_1));
23     Put_Line ("I_Var_2: "
24             & Integer'Image (I_Var_2));
25     --
26     -- ERROR: lifetime of I_Var_2 has ended!
27 end Show_Lifetime;
```

Code block metadata

²⁰⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html>

²⁰⁸ [https://en.wikipedia.org/wiki/Variable_\(computer_science\)#Scope_and_extent](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.)

14.10.2 Accessibility Levels

In basic terms, accessibility levels are a mechanism to assess the lifetime of objects (as we've just discussed). The starting point is the library level: this is the base level, and no level can be deeper than that. We start "moving" to deeper levels when we use a library in a subprogram or call other subprograms for example.

Suppose we have a procedure `Proc` that makes use of a package `Pkg`, and there's a block in the `Proc` procedure:

```
package Pkg is
    -- Library level
end Pkg;
with Pkg; use Pkg;
procedure Proc is
    -- One level deeper than
    -- library level
begin
    declare
        -- Two levels deeper than
        -- library level
    begin
        null;
    end;
end Proc;
```

For this code, we can say that:

- the specification of `Pkg` is at library level;
- the declarative part of `Proc` is one level deeper than the library level; and
- the block is two levels deeper than the library level.

(Note that this is still a very simplified overview of accessibility levels. Things start getting more complicated when we use information from `Pkg` in `Proc`. Those details will become more clear in the next sections.)

The levels themselves are not visible to the programmer. For example, there's no `Access_Level` attribute that returns an integer value indicating the level. Also, you cannot write a user message that displays the level at a certain point. In this sense, accessibility levels are assessed relatively to each other: we can only say that a specific operation is at the same or at a deeper level than another one.

14.10.3 Accessibility Rules

The accessibility rules determine whether a specific use of access types or objects is legal (or not). Actually, accessibility rules exist to prevent *dangling references* (page 541), which we discuss later. Also, they are based on the *accessibility levels* (page 535) we discussed earlier.

Code example

As mentioned earlier, the accessibility level at a specific point isn't visible to the programmer. However, to illustrate which level we have at each point in the following code example, we use a prefix (L0, L1, and L2) to indicate whether we're at the library level (L0) or at a deeper level.

Let's now look at the complete code example:

Listing 89: library_level.ads

```
1 package Library_Level is
2
3   type L0_Integer_Access is
4     access all Integer;
5
6   L0_IA : L0_Integer_Access;
7
8   L0_Var : aliased Integer;
9
10 end Library_Level;
```

Listing 90: show_library_level.adb

```
1 with Library_Level; use Library_Level;
2
3 procedure Show_Library_Level is
4   type L1_Integer_Access is
5     access all Integer;
6
7   L0_IA_2 : L0_Integer_Access;
8   L1_IA   : L1_Integer_Access;
9
10  L1_Var : aliased Integer;
11
12  procedure Test is
13    type L2_Integer_Access is
14      access all Integer;
15
16    L2_IA : L2_Integer_Access;
17
18    L2_Var : aliased Integer;
19  begin
20    L1_IA := L2_Var'Access;
21    --    ^^^^^^
```

(continues on next page)

(continued from previous page)

```

22     --      ILLEGAL: L2 object to
23     --      L1 access object
24
25     L2_IA := L2_Var'Access;
26     --      ^^^^^^
27     --      LEGAL: L2 object to
28     --      L2 access object
29 end Test;
30
31 begin
32     L0_IA := new Integer'(22);
33     --      ^^^^^^^^^^^
34     --      LEGAL: L0 object to
35     --      L0 access object
36
37     L0_IA_2 := new Integer'(22);
38     --      ^^^^^^^^^^^
39     --      LEGAL: L0 object to
40     --      L0 access object
41
42     L0_IA := L1_Var'Access;
43     --      ^^^^^^
44     --      ILLEGAL: L1 object to
45     --      L0 access object
46
47     L0_IA_2 := L1_Var'Access;
48     --      ^^^^^^
49     --      ILLEGAL: L1 object to
50     --      L0 access object
51
52     L1_IA := L0_Var'Access;
53     --      ^^^^^^
54     --      LEGAL: L0 object to
55     --      L1 access object
56
57     L1_IA := L1_Var'Access;
58     --      ^^^^^^
59     --      LEGAL: L1 object to
60     --      L1 access object
61
62     L0_IA := L1_IA;
63     --      ^^^^^^
64     --      ILLEGAL: type mismatch
65
66     L0_IA := L0_Integer_Access (L1_IA);
67     --      ^^^^^^^^^^^^^^^^^^^
68     --      ILLEGAL: cannot convert
69     --      L1 access object to
70     --      L0 access object
71
72     Test;
73 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
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

```

1  procedure Show_Same_Level_Conversion is
2      type L1_Integer_Access is
3          access all Integer;
4
5      type L1_B_Integer_Access is
6          access all Integer;
7
8      L1_IA    : L1_Integer_Access;
9      L1_B_IA : L1_B_Integer_Access;
10
11     L1_Var   : aliased Integer;
12 begin
13     L1_IA := L1_Var'Access;
14
15     L1_B_IA := L1_B_Integer_Access (L1_IA);
16     --      ~~~~~
17     --      LEGAL: conversion from
18     --              L1 access object to
19     --              L1 access object
20 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.

14.10.4 Accessibility rules on parameters

Note that the accessibility rules also apply to access values as subprogram parameters. For example, compilation fails for this example:

Listing 92: names.ads

```
1 package Names is
2
3     type Name is access all String;
4
5     type Constant_Name is
6       access constant String;
7
8     procedure Show (N : Constant_Name);
9
10 end Names;
```

Listing 93: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Show (N : Constant_Name) is
9     begin
10        -- for I in N'Range loop
11          N (I) := To_Lower (N (I));
12        -- end loop;
13        Put_Line ("Name: " & N.all);
14    end Show;
15
16 end Names;
```

Listing 94: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4   S : aliased String := "John";
5 begin
6   Show (S'Access);
7 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

```

1 with Names; use Names;
2
3 procedure Show_Names is
4   S : Name := new String("John");
5 begin
6   Show (Constant_Name (S));
7 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.

14.10.5 Dangling References

An access value that points to a non-existent object is called a dangling reference. Later on, we'll discuss how dangling references may occur using *unchecked deallocation* (page 549).

Dangling references are created when we have an access value pointing to an object whose lifetime has ended, so it becomes a non-existent object. This could occur, for example, when an access value still points to an object X that has gone out of scope.

As mentioned in the previous section, the accessibility rules of the Ada language ensure that such situations never happen! In fact, whenever possible, the compiler applies those rules to detect potential dangling references at compile time. When this detection isn't possible at compile time, the compiler introduces an *accessibility check* (page 410). If this check fails at runtime, it raises a Program_Error exception — thereby preventing that a dangling reference gets used.

Let's see an example of how dangling references could occur:

Listing 96: show_dangling_reference.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Dangling_Reference is
4
5   type Integer_Access is
6     access all Integer;
7
8   I_Var_1 : aliased Integer := 22;
9
10  A1      : Integer_Access;
11 begin
12  A1 := I_Var_1'Access;
13  Put_Line ("A1.all: "
14           & Integer'Image (A1.all));
15
16  Put_Line ("Inner_Block will start now!");
17
18  Inner_Block : declare
```

(continues on next page)

(continued from previous page)

```

19  --
20  --  I_Var_2 only exists in Inner_Block
21  --
22  I_Var_2 : aliased Integer := 42;
23
24  --
25  --  A2 only exists in Inner_Block
26  --
27  A2      : Integer_Access;
28  begin
29  A2 := I_Var_1'Access;
30  Put_Line ("A2.all: "
31           & Integer'Image (A2.all));
32
33  A1 := I_Var_2'Access;
34  --  PROBLEM: A1 and Integer_Access type
35  --            have longer lifetime than
36  --            I_Var_2
37
38  Put_Line ("A1.all: "
39           & Integer'Image (A1.all));
40
41  A2 := I_Var_2'Access;
42  --  PROBLEM: A2 has the same lifetime as
43  --            I_Var_2, but Integer_Access
44  --            type has a longer lifetime.
45
46  Put_Line ("A2.all: "
47           & Integer'Image (A2.all));
48  end Inner_Block;
49
50  Put_Line ("Inner_Block has ended!");
51  Put_Line ("A1.all: "
52           & Integer'Image (A1.all));
53
54  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;
--   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 544).

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 551).

14.11 Unchecked Access

In this section, we discuss the `Unchecked_Access` attribute, which we can use to circumvent accessibility issues for objects in specific cases. (Note that this attribute only exists for objects, not for subprograms.)

We've seen *previously* (page 534) that the accessibility levels verify the lifetime of access types. Let's see a simplified version of a code example from that section:

Listing 97: integers.ads

```
1 package Integers is
2
3     type Integer_Access is access all Integer;
4
5 end Integers;
```

Listing 98: show_access_issue.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Integers; use Integers;
4
5 procedure Show_Access_Issue is
6     I_Var : aliased Integer := 42;
7
8     A      : Integer_Access;
9 begin
10    A := I_Var'Access;
11    -- PROBLEM: A has the same lifetime as I_Var,
12    --           but Integer_Access type has a
13    --           longer lifetime.
14
15    Put_Line ("A.all: " & Integer'Image (A.all));
16 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 use the `Unchecked_Access` attribute. For instance, we can use this attribute to

circumvent the compilation error in the previous code example, since we know that the assignment is actually safe:

Listing 99: integers.ads

```
1 package Integers is
2
3     type Integer_Access is access all Integer;
4
5 end Integers;
```

Listing 100: show_access_issue.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Integers; use Integers;
4
5 procedure Show_Access_Issue is
6     I_Var : aliased Integer := 42;
7
8     A      : Integer_Access;
9 begin
10    A := I_Var'Unchecked_Access;
11    -- OK: assignment is now accepted.
12
13    Put_Line ("A.all: " & Integer'Image (A.all));
14 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](#)²⁰⁹

²⁰⁹ <http://www.ada-auth.org/standards/22rm/html/RM-13-10.html>

14.12 Unchecked Deallocation

So far, we've seen multiple examples of using `new` to allocate objects. In this section, we discuss how to manually deallocate objects.

Our starting point to manually deallocate an object is the generic `Ada.Unchecked_Deallocation` procedure. We first instantiate this procedure for an access type whose objects we want to be able to deallocate. For example, let's instantiate it for the `Integer_Access` type:

Listing 101: integer_types.ads

```

1 with Ada.Unchecked_Deallocation;
2
3 package Integer_Types is
4
5     type Integer_Access is access Integer;
6
7     --
8     -- Instantiation of Ada.Unchecked_Deallocation
9     -- for the Integer_Access type:
10    --
11    procedure Free is
12        new Ada.Unchecked_Deallocation
13            (Object => Integer,
14             Name   => Integer_Access);
15 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

```

1 with Ada.Unchecked_Deallocation;
2
3 package Integer_Types is
4
5     type Integer_Access is access Integer;
6
7     procedure Free is
```

(continues on next page)

(continued from previous page)

```

8     new Ada.Unchecked_Deallocation
9       (Object => Integer,
10        Name  => Integer_Access);
11
12    type Another_Integer_Access is access Integer;
13
14    procedure Free is
15      new Ada.Unchecked_Deallocation
16        (Object => Integer,
17         Name  => Another_Integer_Access);
18  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

```

1  with Ada.Unchecked_Deallocation;
2
3  package Integer_Types is
4
5     type Integer_Access is access Integer;
6
7     procedure Free is
8       new Ada.Unchecked_Deallocation
9         (Object => Integer,
10         Name  => Integer_Access);
11
12    procedure Show_Is_Null (I : Integer_Access);
13
14  end Integer_Types;

```

Listing 104: integer_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Integer_Types is
4
5     procedure Show_Is_Null (I : Integer_Access) is
6     begin
7       if I = null then
8         Put_Line ("access value is null.");
9       else
10        Put_Line ("access value is NOT null.");
11      end if;
12    end Show_Is_Null;
13
14  end Integer_Types;

```

Listing 105: show_unchecked_deallocation.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Integer_Types; use Integer_Types;
3
4 procedure Show_Unchecked_Deallocation is
5
6     I : Integer_Access;
7
8 begin
9     Put ("We haven't called new yet... ");
10    Show_Is_Null (I);
11
12    Put ("Calling new... ");
13    I := new Integer;
14    Show_Is_Null (I);
15
16    Put ("Calling Free... ");
17    Free (I);
18    Show_Is_Null (I);
19 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

```
1 with Integer_Types; use Integer_Types;
2
3 procedure Show_Unchecked_Deallocation is
4
5     I : Integer_Access;
6
7 begin
8     I := new Integer;
9
10    Free (I);
11    Free (I);
12    Free (I);
13 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 Allocators²¹⁰](#)
 - [13.11.2 Unchecked Storage Deallocation²¹¹](#)
-

14.12.1 Unchecked Deallocation and Dangling References

We've discussed *dangling references* (page 541) before. In this section, we discuss how unchecked deallocation can create dangling references and the issues of having them in an application.

Let's reuse the last example and introduce `I_2`, which will point to the same object as `I`:

Listing 107: `show_unchecked_deallocation.adb`

```
1 with Integer_Types; use Integer_Types;
2
3 procedure Show_Unchecked_Deallocation is
4
5     I, I_2 : Integer_Access;
6
7 begin
8     I := new Integer;
9
10    I_2 := I;
11
12    -- NOTE: I_2 points to the same
13    --       object as I.
14
15    --
16    -- Use I and I_2...
17    --
18    -- ... then deallocate memory...
19    --
20
21    Free (I);
22
23    -- NOTE: at this point, I_2 is a
24    --       dangling reference!
25
26    -- Further calls to Free (I)
27    -- are OK!
28
29    Free (I);
30    Free (I);
```

(continues on next page)

²¹⁰ <http://www.ada-auth.org/standards/22rm/html/RM-4-8.html>

²¹¹ <http://www.ada-auth.org/standards/22rm/html/RM-13-11-2.html>

(continued from previous page)

```
31
32  -- A call to Free (I_2) is
33  -- NOT OK:
34
35  Free (I_2);
36 end Show_Unchecked_Deallocation;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↳Deallocation.Unchecked_Deallocation
MD5: ee5c20209a113a6c1bc7895b8ebdb174
```

Runtime output

```
free(): double free detected in tcache 2

raised PROGRAM_ERROR : unhandled signal
```

As we've seen before, we can have multiple calls to `Free (I)`. However, the call to `Free (I_2)` is bad because `I_2` is not null. In fact, it is a dangling reference — i.e. `I_2` points to an object that doesn't exist anymore. Also, the first call to `Free (I)` will reclaim the storage that was allocated for the object that `I` originally referred to. The call to `Free (I_2)` will then try to reclaim the previously-reclaimed object, but it'll fail in an undefined manner.

Because of these potential errors, you should be very careful when using unchecked deallocation: it is the programmer's responsibility to avoid creating dangling references!

For the example we've just seen, we could avoid creating a dangling reference by explicitly assigning `null` to `I_2` to indicate that it doesn't point to any specific object:

Listing 108: show_unchecked_deallocation.adb

```
1 with Integer_Types; use Integer_Types;
2
3 procedure Show_Unchecked_Deallocation is
4
5     I, I_2 : Integer_Access;
6
7 begin
8     I := new Integer;
9
10    I_2 := I;
11
12    -- NOTE: I_2 points to the same
13    --       object as I.
14
15    --
16    -- Use I and I_2...
17    --
18    -- ... then deallocate memory...
19    --
20
21    I_2 := null;
22
23    -- NOTE: now, I_2 doesn't point to
24    --       any object, so calling
25    --       Free (I_2) is OK.
26
27    Free (I);
28    Free (I_2);
29 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 558) later on.

14.12.2 Dereferencing dangling references

Of course, you shouldn't try to dereference a dangling reference because your program becomes erroneous, as we discuss in this section. Let's see an example:

Listing 109: show_unchecked_deallocation.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Integer_Types; use Integer_Types;
3
4 procedure Show_Unchecked_Deallocation is
5
6     I_1, I_2 : Integer_Access;
7
8 begin
9     I_1 := new Integer'(42);
10    I_2 := I_1;
11
12    Put_Line ("I_1.all = "
13             & Integer'Image (I_1.all));
14    Put_Line ("I_2.all = "
15             & Integer'Image (I_2.all));
16
17    Put_Line ("Freeing I_1");
18    Free (I_1);
19
20    if I_1 /= null then
21        Put_Line ("I_1.all = "
22                 & Integer'Image (I_1.all));
23    end if;
24
25    if I_2 /= null then
26        Put_Line ("I_2.all = "
27                 & Integer'Image (I_2.all));
28    end if;
29 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 = 7670
```

In this example, we allocate an object for `I_1` and make `I_2` point to the same object. Then, we call `Free (I)`, which has the following consequences:

- The call to `Free (I_1)` will try to reclaim the storage for the original object (`I_1.all`), so it may be reused for other allocations.
- `I_1 = null` after the call to `Free (I_1)`.
- `I_2` becomes a dangling reference by the call to `Free (I_1)`.
 - In other words, `I_2` is still non-null, and what it points to is now undefined.

In principle, we could check for `null` before trying to dereference the access value. (Remember that when deallocating an object via a call to `Free`, the corresponding access value is set to `null`.) In fact, this strategy works fine for `I_1`, but it doesn't work for `I_2` because the access value is not `null`. As a consequence, the application tries to dereference `I_2`.

Dereferencing a dangling reference is erroneous: the behavior is undefined in this case. For the example we've just seen,

- `I_2.all` might make the application crash;
- `I_2.all` might give us a different value than before;
- `I_2.all` might even give us the same value as before (42) if the original object is still available.

Because the effect is unpredictable, it might be really difficult to debug the application and identify the cause.

Having dangling pointers in an application should be avoided at all costs! Again, it is the programmer's responsibility to be very careful when using unchecked deallocation: avoid creating dangling references!

In the Ada Reference Manual

- [13.9.1 Data Validity](#)²¹²
 - [13.11.2 Unchecked Storage Deallocation](#)²¹³
-

14.12.3 Restrictions for `Ada.Unchecked_Deallocation`

There are two unsurprising restrictions for `Ada.Unchecked_Deallocation`:

1. It cannot be instantiated for access-to-constant types; and
2. It cannot be used when the `Storage_Size` aspect of a type is zero (i.e. when its storage pool is empty).

(Note that this last restriction also applies to the allocation via `new`.)

Let's see an example of these restrictions:

²¹² <http://www.ada-auth.org/standards/22rm/html/RM-13-9-1.html>

²¹³ <http://www.ada-auth.org/standards/22rm/html/RM-13-11-2.html>

Listing 110: show_unchecked_deallocation_errors.adb

```

1 with Ada.Unchecked_Deallocation;
2
3 procedure Show_Unchecked_Deallocation_Errors is
4
5     type Integer_Access_Zero is access Integer
6         with Storage_Size => 0;
7
8     procedure Free is
9         new Ada.Unchecked_Deallocation
10            (Object => Integer,
11             Name   => Integer_Access_Zero);
12
13     type Constant_Integer_Access is
14         access constant Integer;
15
16     -- ERROR: Cannot use access-to-constant type
17     --         for Name
18     procedure Free is
19         new Ada.Unchecked_Deallocation
20            (Object => Integer,
21             Name   => Constant_Integer_Access);
22
23     I : Integer_Access_Zero;
24
25 begin
26     -- ERROR: Cannot allocate objects from
27     --         empty storage pool
28     I := new Integer;
29
30     -- ERROR: Cannot deallocate objects from
31     --         empty storage pool
32     Free (I);
33 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.

14.13 Null & Not Null Access

Note: This section was originally written by Robert A. Duff and published as [Gem #23: Null Considered Harmful](#)²¹⁴ and [Gem #24](#)²¹⁵.

Ada, like many languages, defines a special **null** value for access types. All values of an access type designate some object of the designated type, except for **null**, which does not designate any object. The null value can be used as a special flag. For example, a singly-linked list can be null-terminated. A Lookup function can return **null** to mean "not found", presuming the result is of an access type:

Listing 111: show_null_return.ads

```
1 package Show_Null_Return is
2
3     type Ref_Element is access all Element;
4
5     Not_Found : constant Ref_Element := null;
6
7     function Lookup (T : Table) return Ref_Element;
8     -- Returns Not_Found if not found.
9 end Show_Null_Return;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_Access.Null_Return
↪ Access.Null_Return
MD5: 6c4eed750d42685198ec9495805e3e23

An alternative design for Lookup would be to raise an exception:

Listing 112: show_not_found_exception.ads

```
1 package Show_Not_Found_Exception is
2     Not_Found : exception;
3
4     function Lookup (T : Table) return Ref_Element;
5     -- Raises Not_Found if not found.
6     -- Never returns null.
7 end Show_Not_Found_Exception;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_Access.Not_Found_Exception
↪ 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:

²¹⁴ <https://www.adacore.com/gems/ada-gem-23>

²¹⁵ <https://www.adacore.com/gems/ada-gem-24>

Listing 113: show_not_null_return.ads

```

1 package Show_Not_Null_Return is
2     type Ref_Element is access all Element;
3
4     Not_Found : constant Ref_Element := null;
5
6     function Lookup (T : Table)
7         return not null Ref_Element;
8     -- Possible since Ada 2005.
9 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

```

1 package Example is
2
3     type Element is limited private;
4     type Ref_Element is access all Element;
5
6     type Table is limited private;
7
8     Not_Found : constant Ref_Element := null;
9     function Lookup (T : Table)
10         return Ref_Element;
11     -- Returns Not_Found if not found.
12
13     Not_Found_2 : exception;
14     function Lookup_2 (T : Table)
15         return not null Ref_Element;
16     -- Raises Not_Found_2 if not found.
17
18     procedure P (X : not null Ref_Element);
19
20     procedure Q (X : not null Ref_Element);
21
22 private
23     type Element is limited
24         record
25             Component : Integer;
26         end record;
27     type Table is limited null record;
28 end Example;
```

Listing 115: example.adb

```

1 package body Example is
2
3     An_Element : aliased Element;
4
5     function Lookup (T : Table)
6         return Ref_Element is
7         pragma Unreferenced (T);
8     begin
```

(continues on next page)

(continued from previous page)

```

9      -- ...
10     return Not_Found;
11 end Lookup;
12
13 function Lookup_2 (T : Table)
14                 return not null Ref_Element
15 is
16 begin
17     -- ...
18     raise Not_Found_2;
19
20     return An_Element'Access;
21     -- suppress error: 'missing "return"
22     -- statement in function body'
23 end Lookup_2;
24
25 procedure P (X : not null Ref_Element) is
26 begin
27     X.all.Component := X.all.Component + 1;
28 end P;
29
30 procedure Q (X : not null Ref_Element) is
31 begin
32     for I in 1 .. 1000 loop
33         P (X);
34     end loop;
35 end Q;
36
37 procedure R is
38 begin
39     Q (An_Element'Access);
40 end R;
41
42 pragma Unreferenced (R);
43
44 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](https://en.wikipedia.org/wiki/Standard_ML)²¹⁶ works, for example — you don't get any special null-like value unless you ask for it. Of course, because Ada 2005 needs to be compatible with previous versions of the language, **not null** cannot be the default for Ada.

One word of caution: access objects are default-initialized to **null**, so if you have a **not null** object (or component) you had better initialize it explicitly, or you will get `Constraint_Error`. **not null** is more often useful on parameters and function results, for this reason.

²¹⁶ https://en.wikipedia.org/wiki/Standard_ML

Another advantage of **not null** over comments is for efficiency. Consider procedures P and Q in this example:

Listing 116: example-processing.ads

```

1 package Example.Processing is
2
3   procedure P (X : not null Ref_Element);
4
5   procedure Q (X : not null Ref_Element);
6
7 end Example.Processing;
```

Listing 117: example-processing.adb

```

1 package body Example.Processing is
2
3   procedure P (X : not null Ref_Element) is
4   begin
5     X.all.Component := X.all.Component + 1;
6   end P;
7
8   procedure Q (X : not null Ref_Element) is
9   begin
10    for I in 1 .. 1000 loop
11      P (X);
12    end loop;
13  end Q;
14
15 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 $X \neq \text{null}$, 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

```

1 package Show_Process_Array is
2
3   type My_Index is range 1 .. 10;
4   type My_Array is array (My_Index) of Integer;
5
6   procedure Process_Array
7     (X      : in out My_Array;
8      Index  :          My_Index);
9
10 end Show_Process_Array;
```


Listing 119: show_process_array.adb

```
1 package body Show_Process_Array is
2
3     procedure Process_Array
4         (X      : in out My_Array;
5          Index  :           My_Index) is
6     begin
7         X (Index) := X (Index) + 1;
8     end Process_Array;
9
10 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.

14.14 Design strategies for access types

Previously, we learned about *dangling references* (page 541) and discussed the effects of *dereferencing them* (page 551). Also, we've seen the relationship between *unchecked deallocation and dangling references* (page 549). Ensuring that all calls to `Free` for a specific access type will never cause dangling references can become an arduous task — if not impossible — if those calls are located in different parts of the source code.

Although we used access types directly in the main application in many of the previous code examples from this chapter, this approach was in fact selected just for illustration purposes — i.e. to make the code look simpler. In general, however, we should avoid this approach. Instead, our recommendation is to encapsulate the access types in some form of abstraction. In this section, we discuss design strategies for access types that take this recommendation into account.

14.14.1 Abstract data type for access types

The simplest form of abstraction is of course an abstract data type. For example, we could declare a limited private type, which allows us to hide the access type and to avoid copies of references that could potentially become dangling references. (We discuss limited private types later *in another chapter* (page 677).)

Let's see an example:

Listing 120: access_type_abstraction.ads

```
1 package Access_Type_Abstraction is
2
3     type Info is limited private;
4
5     function To_Info (S : String) return Info;
6
7     function To_String (Obj : Info)
8         return String;
```

(continues on next page)

(continued from previous page)

```

9
10  function Copy (Obj : Info) return Info;
11
12  procedure Copy (To   : in out Info;
13                From :      Info);
14
15  procedure Append (Obj : in out Info;
16                  S   : String);
17
18  procedure Reset (Obj : in out Info);
19
20  procedure Destroy (Obj : in out Info);
21
22  private
23
24    type Info is access String;
25
26  end Access_Type_Abstraction;

```

Listing 121: access_type_abstraction.adb

```

1  with Ada.Unchecked_Deallocation;
2
3  package body Access_Type_Abstraction is
4
5    function To_Info (S : String) return Info is
6      (new String'(S));
7
8    function To_String (Obj : Info)
9      return String is
10     (if Obj /= null then Obj.all else "");
11
12    function Copy (Obj : Info) return Info is
13     (To_Info (Obj.all));
14
15    procedure Copy (To   : in out Info;
16                  From :      Info) is
17    begin
18     Destroy (To);
19     To := To_Info (From.all);
20    end Copy;
21
22    procedure Append (Obj : in out Info;
23                    S   : String) is
24     New_Info : constant Info :=
25     To_Info (To_String (Obj) & S);
26    begin
27     Destroy (Obj);
28     Obj := New_Info;
29    end Append;
30
31    procedure Reset (Obj : in out Info) is
32    begin
33     Destroy (Obj);
34    end Reset;
35
36    procedure Destroy (Obj : in out Info) is
37     procedure Free is
38     new Ada.Unchecked_Deallocation
39     (Object => String,
40      Name   => Info);

```

(continues on next page)

(continued from previous page)

```

41  begin
42      Free (Obj);
43  end Destroy;
44
45  end Access_Type_Abstraction;

```

Listing 122: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Access_Type_Abstraction;
4  use Access_Type_Abstraction;
5
6  procedure Main is
7      Obj_1 : Info := To_Info ("hello");
8      Obj_2 : Info := Copy (Obj_1);
9  begin
10     Put_Line ("TO_INFO / COPY");
11     Put_Line ("Obj_1 : "
12             & To_String (Obj_1));
13     Put_Line ("Obj_2 : "
14             & To_String (Obj_2));
15     Put_Line ("-----");
16
17     Reset (Obj_1);
18     Append (Obj_2, " world");
19
20     Put_Line ("RESET / APPEND");
21     Put_Line ("Obj_1 : "
22             & To_String (Obj_1));
23     Put_Line ("Obj_2 : "
24             & To_String (Obj_2));
25     Put_Line ("-----");
26
27     Copy (From => Obj_2,
28          To   => Obj_1);
29
30     Put_Line ("COPY");
31     Put_Line ("Obj_1 : "
32             & To_String (Obj_1));
33     Put_Line ("Obj_2 : "
34             & To_String (Obj_2));
35     Put_Line ("-----");
36
37     Destroy (Obj_1);
38     Destroy (Obj_2);
39
40     Put_Line ("DESTROY");
41     Put_Line ("Obj_1 : "
42             & To_String (Obj_1));
43     Put_Line ("Obj_2 : "
44             & To_String (Obj_2));
45     Put_Line ("-----");
46
47     Append (Obj_1, "hey");
48
49     Put_Line ("APPEND");
50     Put_Line ("Obj_1 : "
51             & To_String (Obj_1));
52     Put_Line ("-----");
53

```

(continues on next page)

(continued from previous page)

```

54   Put_Line ("APPEND");
55   Append (Obj_1, " there");
56   Put_Line ("Obj_1 : "
57             & To_String (Obj_1));
58
59   Destroy (Obj_1);
60   Destroy (Obj_2);
61 end Main;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Design_Strategies.
↳ Access_Type_Abstraction
MD5: d652d26314b616d3e1b955c0ce5bbbd7
```

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
-----
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.

14.14.2 Controlled type for access types

In the previous code example, the Destroy procedure had to be called to deallocate the hidden access object. We could make sure that this deallocation happens automatically by using a controlled (or limited controlled) type. (We discuss controlled types in another chapter.)

Let's adapt the previous example and declare Info as a limited controlled type:

Listing 123: access_type_abstraction.ads

```

1 with Ada.Finalization;
2
3 package Access_Type_Abstraction is
```

(continues on next page)

(continued from previous page)

```

4
5  type Info is limited private;
6
7  function To_Info (S : String) return Info;
8
9  function To_String (Obj : Info)
10     return String;
11
12  function Copy (Obj : Info) return Info;
13
14  procedure Copy (To   : in out Info;
15                From :      Info);
16
17  procedure Append (Obj : in out Info;
18                  S   :      String);
19
20  procedure Reset (Obj : in out Info);
21
22  private
23
24  type String_Access is access String;
25
26  type Info is new
27     Ada.Finalization.Limited_Controlled with
28     record
29         Str_A : String_Access;
30     end record;
31
32  procedure Initialize (Obj : in out Info);
33  procedure Finalize (Obj : in out Info);
34
35  end Access_Type_Abstraction;

```

Listing 124: access_type_abstraction.adb

```

1  with Ada.Unchecked_Deallocation;
2
3  package body Access_Type_Abstraction is
4
5     --
6     --  STRING_ACCESS SUBPROGRAMS
7     --
8
9     function To_String_Access (S : String)
10        return String_Access
11  is
12     (new String'(S));
13
14     function To_String (S : String_Access)
15        return String is
16     (if S /= null then S.all else "");
17
18     procedure Free is
19     new Ada.Unchecked_Deallocation
20     (Object => String,
21      Name   => String_Access);
22
23     --
24     --  PRIVATE SUBPROGRAMS
25     --
26

```

(continues on next page)

(continued from previous page)

```

27  procedure Initialize (Obj : in out Info) is
28  begin
29      -- Put_Line ("Initializing Info");
30      Obj.Str_A := null;
31      -- ^^^^^^^^^^^^^^^^^
32      -- NOTE: This line has just been added to
33      --       illustrate the "automatic" call to
34      --       Initialize. Actually, this
35      --       assignment isn't needed, as
36      --       the Str_A component is
37      --       automatically initialized to null
38      --       upon object construction.
39  end Initialize;
40
41  procedure Finalize (Obj : in out Info) is
42  begin
43      -- Put_Line ("Finalizing Info");
44      Free (Obj.Str_A);
45  end Finalize;
46
47      --
48      -- PUBLIC SUBPROGRAMS
49      --
50
51  function To_Info (S : String) return Info is
52      (Ada.Finalization.Limited_Controlled
53       with Str_A => To_String_Access (S));
54
55  function To_String (Obj : Info)
56      return String is
57      (To_String (Obj.Str_A));
58
59  function Copy (Obj : Info) return Info is
60      (To_Info (To_String (Obj.Str_A)));
61
62  procedure Copy (To   : in out Info;
63                From :      Info) is
64  begin
65      Free (To.Str_A);
66      To.Str_A := To_String_Access
67                  (To_String (From.Str_A));
68  end Copy;
69
70  procedure Append (Obj : in out Info;
71                  S   :      String) is
72      New_Str_A : constant String_Access :=
73                  To_String_Access
74                  (To_String (Obj.Str_A) & S);
75  begin
76      Free (Obj.Str_A);
77      Obj.Str_A := New_Str_A;
78  end Append;
79
80  procedure Reset (Obj : in out Info) is
81  begin
82      Free (Obj.Str_A);
83  end Reset;
84
85  end Access_Type_Abstraction;

```

Listing 125: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_Type_Abstraction;
4 use Access_Type_Abstraction;
5
6 procedure Main is
7   Obj_1 : Info := To_Info ("hello");
8   Obj_2 : Info := Copy (Obj_1);
9 begin
10  --
11  -- TO_INFO / COPY
12  --
13  Put_Line ("TO_INFO / COPY");
14
15  Put_Line ("Obj_1 : "
16           & To_String (Obj_1));
17  Put_Line ("Obj_2 : "
18           & To_String (Obj_2));
19  Put_Line ("-----");
20
21  --
22  -- RESET: Obj_1
23  -- APPEND: Obj_2
24  --
25  Put_Line ("RESET / APPEND");
26
27  Reset (Obj_1);
28  Append (Obj_2, " world");
29
30  Put_Line ("Obj_1 : "
31           & To_String (Obj_1));
32  Put_Line ("Obj_2 : "
33           & To_String (Obj_2));
34  Put_Line ("-----");
35
36  --
37  -- COPY: Obj_2 => Obj_1
38  --
39  Put_Line ("COPY");
40
41  Copy (From => Obj_2,
42       To   => Obj_1);
43
44  Put_Line ("Obj_1 : "
45           & To_String (Obj_1));
46  Put_Line ("Obj_2 : "
47           & To_String (Obj_2));
48  Put_Line ("-----");
49
50  --
51  -- RESET: Obj_1, Obj_2
52  --
53  Put_Line ("RESET");
54
55  Reset (Obj_1);
56  Reset (Obj_2);
57
58  Put_Line ("Obj_1 : "
59           & To_String (Obj_1));
```

(continues on next page)

(continued from previous page)

```

60   Put_Line ("Obj_2 : "
61           & To_String (Obj_2));
62   Put_Line ("-----");
63
64   --
65   --  COPY: Obj_2 => Obj_1
66   --
67   Put_Line ("COPY");
68
69   Copy (From => Obj_2,
70        To   => Obj_1);
71
72   Put_Line ("Obj_1 : "
73           & To_String (Obj_1));
74   Put_Line ("Obj_2 : "
75           & To_String (Obj_2));
76   Put_Line ("-----");
77
78   --
79   --  APPEND: Obj_1 with "hey"
80   --
81   Put_Line ("APPEND");
82
83   Append (Obj_1, "hey");
84
85   Put_Line ("Obj_1 : "
86           & To_String (Obj_1));
87   Put_Line ("-----");
88
89   --
90   --  APPEND: Obj_1 with "there"
91   --
92   Put_Line ("APPEND");
93
94   Append (Obj_1, " there");
95
96   Put_Line ("Obj_1 : "
97           & To_String (Obj_1));
98 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
```

(continues on next page)

(continued from previous page)

```
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.)

14.15 Access to subprograms

So far in this chapter, we focused mainly on access-to-objects. However, we can use access types to subprograms. This is the topic of this section.

14.15.1 Static vs. dynamic calls

In a typical subprogram call, we indicate the subprogram we want to call statically. For example, let's say we've implemented a procedure `Proc` that calls a procedure `P`:

Listing 126: p.ads

```
1 procedure P (I : in out Integer);
```

Listing 127: p.adb

```
1 procedure P (I : in out Integer) is  
2 begin  
3   null;  
4 end P;
```

Listing 128: proc.adb

```
1 with P;  
2  
3 procedure Proc is  
4   I : Integer := 0;  
5 begin
```

(continues on next page)

(continued from previous page)

```
6   P (I);
7  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.)

14.15.2 Access to subprogram declaration

We declare an access to a subprogram as a type by writing **access procedure** or **access function** and the corresponding prototype:

Listing 129: access_to_subprogram_types.ads

```
1  package Access_To_Subprogram_Types is
2
3     type Access_To_Procedure is
4       access procedure (I : in out Integer);
5
6     type Access_To_Function is
7       access function (I : Integer) return Integer;
8
9  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

```
1  with Access_To_Subprogram_Types;
2  use Access_To_Subprogram_Types;
3
4  package Access_To_Subprogram_Params is
5
6     procedure Proc (P : Access_To_Procedure);
```

(continues on next page)

(continued from previous page)

```
7
8 end Access_To_Subprogram_Params;
```

Listing 131: access_to_subprogram_params.adb

```
1 package body Access_To_Subprogram_Params is
2
3   procedure Proc (P : Access_To_Procedure) is
4     I : Integer := 0;
5   begin
6     P (I);
7     -- P.all (I);
8   end Proc;
9
10 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

```
1 procedure Add_Ten (I : in out Integer);
```

Listing 133: add_ten.adb

```
1 procedure Add_Ten (I : in out Integer) is
2 begin
3   I := I + 10;
4 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

```
1 with Access_To_Subprogram_Params;
2 use Access_To_Subprogram_Params;
3
4 with Add_Ten;
5
6 procedure Show_Access_To_Subprograms is
7 begin
8   Proc (Add_Ten'Access);
9   --      ^ Getting access to Add_Ten
10  --      procedure and passing it
```

(continues on next page)

(continued from previous page)

```
11 -- to Proc
12 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](#)²¹⁷

14.15.3 Objects of access-to-subprogram type

In the previous example, the Proc procedure had a parameter of access-to-subprogram type. In addition to parameters, we can of course declare *objects* of access-to-subprogram types as well. For example, we can extend our previous test application and declare an object P of access-to-subprogram type. Before we do so, however, let's implement another small procedure that we'll use later on:

Listing 135: add_twenty.ads

```
1 procedure Add_Twenty (I : in out Integer);
```

Listing 136: add_twenty.adb

```
1 procedure Add_Twenty (I : in out Integer) is
2 begin
3   I := I + 20;
4 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

```
1 with Access_To_Subprogram_Types;
2 use Access_To_Subprogram_Types;
3
4 with Access_To_Subprogram_Params;
5 use Access_To_Subprogram_Params;
6
7 with Add_Ten;
8 with Add_Twenty;
9
10 procedure Show_Access_To_Subprograms is
```

(continues on next page)

²¹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

11     P      : Access_To_Procedure;
12     Some_Int : Integer := 0;
13 begin
14     P := Add_Ten'Access;
15     --      ^ Getting access to Add_Ten
16     --      procedure and assigning it
17     --      to P
18
19     Proc (P);
20     --      ^ Passing access-to-subprogram as an
21     --      actual parameter
22
23     P (Some_Int);
24     --      ^ Using access-to-subprogram object in a
25     --      subprogram call
26
27     P := Add_Twenty'Access;
28     --      ^ Getting access to Add_Twenty
29     --      procedure and assigning it
30     --      to P
31
32     Proc (P);
33     P (Some_Int);
34 end Show_Access_To_Subprograms;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
 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.

14.15.4 Components of access-to-subprogram type

In addition to declaring subprogram parameters and objects of access-to-subprogram types, we can declare components of these types. For example:

Listing 138: `access_to_subprogram_types.ads`

```

1 package Access_To_Subprogram_Types is
2
3     type Access_To_Procedure is
4         access procedure (I : in out Integer);
5
6     type Access_To_Function is
7         access function (I : Integer) return Integer;
8
9     type Access_To_Procedure_Array is
10        array (Positive range <>) of
11            Access_To_Procedure;
12

```

(continues on next page)

(continued from previous page)

```

13  type Access_To_Function_Array is
14      array (Positive range <>) of
15          Access_To_Function;
16
17  type Rec_Access_To_Procedure is record
18      AP : Access_To_Procedure;
19  end record;
20
21  type Rec_Access_To_Function is record
22      AF : Access_To_Function;
23  end record;
24
25  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

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Access_To_Subprogram_Types;
4  use Access_To_Subprogram_Types;
5
6  with Add_Ten;
7  with Add_Twenty;
8
9  procedure Show_Access_To_Subprograms is
10     PA : constant
11         Access_To_Procedure_Array (1 .. 2) :=
12         (Add_Ten'Access,
13          Add_Twenty'Access);
14
15     Some_Int : Integer := 0;
16  begin
17     Put_Line ("Some_Int: " & Some_Int'Image);
18
19     for I in PA'Range loop
20         PA (I) (Some_Int);
21         Put_Line ("Some_Int: " & Some_Int'Image);
22     end loop;
23  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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 with Add_Ten;
7 with Add_Twenty;
8
9 procedure Show_Access_To_Subprograms is
10     RA      : Rec_Access_To_Procedure;
11     Some_Int : Integer := 0;
12 begin
13     Put_Line ("Some_Int: " & Some_Int'Image);
14
15     RA := (AP => Add_Ten'Access);
16     RA.AP (Some_Int);
17     Put_Line ("Some_Int: " & Some_Int'Image);
18
19     RA := (AP => Add_Twenty'Access);
20     RA.AP (Some_Int);
21     Put_Line ("Some_Int: " & Some_Int'Image);
22 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.

14.15.5 Access-to-subprogram as discriminant types

As you might expect, we can use access-to-subprogram types when declaring discriminants. In fact, when we were talking about *discriminants as access values* (page 492) earlier on, we used access-to-object types in our code examples, but we could have used access-to-subprogram types as well. For example:

Listing 141: custom_processing.ads

```

1 package Custom_Processing is
2
3   -- Declaring an access type:
4   type Integer_Processing is
5     access procedure (I : in out Integer);
6
7   -- Declaring a discriminant with this
8   -- access type:
9   type Rec (IP : Integer_Processing) is
10    private;
11
12   procedure Init (R      : in out Rec;
13                 Value :      Integer);
14
15   procedure Process (R : in out Rec);
16
17   procedure Show (R : Rec);
18
19 private
20
21   type Rec (IP : Integer_Processing) is
22     record
23       I : Integer := 0;
24     end record;
25
26 end Custom_Processing;
```

Listing 142: custom_processing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Processing is
4
5   procedure Init (R      : in out Rec;
6                 Value :      Integer) is
7   begin
8     R.I := Value;
9   end Init;
10
11  procedure Process (R : in out Rec) is
12  begin
13    R.IP (R.I);
14    -- ^^^^^^
15    -- Calling procedure that we specified as
16    -- the record's discriminant
17  end Process;
18
19  procedure Show (R : Rec) is
20  begin
21    Put_Line ("R.I = "
22            & Integer'Image (R.I));
23  end Show;
```

(continues on next page)

(continued from previous page)

```
24
25 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`

```
1 procedure Mult_Two (I : in out Integer);
```

Listing 144: `mult_two.adb`

```
1 procedure Mult_Two (I : in out Integer) is
2 begin
3     I := I * 2;
4 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`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Custom_Processing; use Custom_Processing;
4
5 with Add_Ten;
6 with Mult_Two;
7
8 procedure Show_Access_To_Subprogram_Discriminants
9 is
10
11     R_Add_Ten : Rec (IP => Add_Ten'Access);
12     --           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
13     --           Using access-to-subprogram as a
14     --           discriminant
15
16     R_Mult_Two : Rec (IP => Mult_Two'Access);
17     --           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
18     --           Using access-to-subprogram as a
19     --           discriminant
20
21 begin
22     Init (R_Add_Ten, 1);
23     Init (R_Mult_Two, 2);
24
```

(continues on next page)

(continued from previous page)

```

25   Put_Line ("---- R_Add_Ten ----");
26   Show (R_Add_Ten);
27
28   Put_Line ("Calling Process procedure...");
29   Process (R_Add_Ten);
30   Show (R_Add_Ten);
31
32   Put_Line ("---- R_Mult_Two ----");
33   Show (R_Mult_Two);
34
35   Put_Line ("Calling Process procedure...");
36   Process (R_Mult_Two);
37   Show (R_Mult_Two);
38 end Show_Access_To_Subprogram_Discriminants;

```

Code block metadata

```

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.

14.15.6 Access-to-subprograms as formal parameters

We can use access-to-subprograms types when declaring formal parameters. For example, let's revisit the `Custom_Processing` package from the previous section and convert it into a generic package.

Listing 146: `gen_custom_processing.ads`

```

1  generic
2    type T is private;
3
4    --
5    -- Declaring formal access-to-subprogram
6    -- type:
7    --
8    type T_Processing is
9      access procedure (Element : in out T);
10
11   --
12   -- Declaring formal access-to-subprogram
13   -- parameter:
14   --
15   Proc : T_Processing;

```

(continues on next page)

(continued from previous page)

```
16
17     with function Image_T (Element : T)
18         return String;
19 package Gen_Custom_Processing is
20
21     type Rec is private;
22
23     procedure Init (R      : in out Rec;
24                   Value :      T);
25
26     procedure Process (R : in out Rec);
27
28     procedure Show (R : Rec);
29
30 private
31
32     type Rec is record
33         Comp : T;
34     end record;
35
36 end Gen_Custom_Processing;
```

Listing 147: gen_custom_processing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Gen_Custom_Processing is
4
5     procedure Init (R      : in out Rec;
6                   Value :      T) is
7     begin
8         R.Comp := Value;
9     end Init;
10
11     procedure Process (R : in out Rec) is
12     begin
13         Proc (R.Comp);
14     end Process;
15
16     procedure Show (R : Rec) is
17     begin
18         Put_Line ("R.Comp = "
19                 & Image_T (R.Comp));
20     end Show;
21
22 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

```

1  with Gen_Custom_Processing;
2
3  with Add_Ten;
4
5  with Ada.Text_IO; use Ada.Text_IO;
6
7  procedure
8  Show_Access_To_Subprogram_As_Formal_Parameter
9  is
10     type Integer_Processing is
11         access procedure (I : in out Integer);
12
13     package Custom_Processing is new
14         Gen_Custom_Processing
15         (T           => Integer,
16          T_Processing => Integer_Processing,
17           --
18           --         access-to-subprogram type
19          Proc        => Add_Ten'Access,
20           --
21           --         access-to-subprogram
22          Image_T     => Integer'Image);
23     use Custom_Processing;
24
25     R_Add_Ten  : Rec;
26
27 begin
28     Init (R_Add_Ten, 1);
29
30     Put_Line ("---- R_Add_Ten ----");
31     Show (R_Add_Ten);
32
33     Put_Line ("Calling Process procedure...");
34     Process (R_Add_Ten);
35     Show (R_Add_Ten);
36 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.

14.15.7 Selecting subprograms

A practical application of access to subprograms is that it enables us to dynamically select a subprogram and pass it to another subprogram, where it can then be called.

For example, we may have a Process procedure that receives a logging procedure as a parameter (Log_Proc). Also, this parameter may be **null** by default — so that no procedure is called if the parameter isn't specified:

Listing 149: data_processing.ads

```
1 package Data_Processing is
2
3   type Data_Container is
4     array (Positive range <>) of Float;
5
6   type Log_Procedure is
7     access procedure (D : Data_Container);
8
9   procedure Process
10    (D      : in out Data_Container;
11     Log_Proc : Log_Procedure := null);
12
13 end Data_Processing;
```

Listing 150: data_processing.adb

```
1 package body Data_Processing is
2
3   procedure Process
4     (D      : in out Data_Container;
5      Log_Proc : Log_Procedure := null) is
6   begin
7     -- missing processing part...
8
9     if Log_Proc /= null then
10      Log_Proc (D);
11    end if;
12  end Process;
13
14 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

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Data_Processing; use Data_Processing;
3
4 procedure Log_Element_Per_Line
5   (D : Data_Container) is
6 begin
```

(continues on next page)

(continued from previous page)

```

7   Put_Line ("Elements: ");
8   for V of D loop
9     Put_Line (V'Image);
10  end loop;
11  Put_Line ("-----");
12 end Log_Element_Per_Line;

```

Listing 152: log_csv.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Data_Processing; use Data_Processing;
3
4 procedure Log_Csv (D : Data_Container) is
5 begin
6   for I in D'First .. D'Last - 1 loop
7     Put (D (I)'Image & ", ");
8   end loop;
9   Put (D (D'Last)'Image);
10  New_Line;
11 end Log_Csv;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↳Subprograms.Log_Procedure
MD5: 468789f7331ffc16f754f7116b076d7

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

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Data_Processing; use Data_Processing;
3
4 with Log_Element_Per_Line;
5 with Log_Csv;
6
7 procedure Show_Access_To_Subprograms is
8   D : Data_Container (1 .. 5) := (others => 1.0);
9 begin
10  Put_Line ("==== Log_Element_Per_Line ====");
11  Process (D, Log_Element_Per_Line'Access);
12
13  Put_Line ("==== Log_Csv ====");
14  Process (D, Log_Csv'Access);
15
16  Put_Line ("==== None ====");
17  Process (D);
18 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:

```

(continues on next page)

(continued from previous page)

```

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.

14.15.8 Null exclusion

We can use null exclusion when declaring an access to subprograms. By doing so, we ensure that a subprogram must be specified — either as a parameter or when initializing an access object. Otherwise, an exception is raised. Let's adapt the previous example and introduce the `Init_Function` type:

Listing 154: `data_processing.ads`

```

1 package Data_Processing is
2
3   type Data_Container is
4     array (Positive range <>) of Float;
5
6   type Init_Function is
7     not null access function return Float;
8
9   procedure Process
10    (D      : in out Data_Container;
11     Init_Func : Init_Function);
12
13 end Data_Processing;

```

Listing 155: `data_processing.adb`

```

1 package body Data_Processing is
2
3   procedure Process
4     (D      : in out Data_Container;
5     Init_Func : Init_Function) is
6   begin
7     for I in D'Range loop
8       D (I) := Init_Func.all;
9     end loop;
10  end Process;
11
12 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`

```
1 function Init_Zero return Float;
```

Listing 157: `init_one.ads`

```
1 function Init_One return Float;
```

Listing 158: `init_zero.adb`

```
1 function Init_Zero return Float is
2 begin
3   return 0.0;
4 end Init_Zero;
```

Listing 159: `init_one.adb`

```
1 function Init_One return Float is
2 begin
3   return 1.0;
4 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`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Data_Processing; use Data_Processing;
3
4 procedure Log_Element_Per_Line
5   (D : Data_Container) is
6 begin
7   Put_Line ("Elements: ");
8   for V of D loop
9     Put_Line (V'Image);
10  end loop;
11  Put_Line ("-----");
12 end Log_Element_Per_Line;
```

Listing 161: `show_access_to_subprograms.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Data_Processing; use Data_Processing;
3
4 with Init_Zero;
5 with Init_One;
6
7 with Log_Element_Per_Line;
8
9 procedure Show_Access_To_Subprograms is
10  D : Data_Container (1 .. 5) := (others => 1.0);
```

(continues on next page)

(continued from previous page)

```
11 begin
12   Put_Line ("==== Init_Zero ====");
13   Process (D, Init_Zero'Access);
14   Log_Element_Per_Line (D);
15
16   Put_Line ("==== Init_One ====");
17   Process (D, Init_One'Access);
18   Log_Element_Per_Line (D);
19
20   -- Put_Line ("==== None ====");
21   -- Process (D, null);
22   -- Log_Element_Per_Line (D);
23 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](#)²¹⁸.

Here's another example, first with **null**:

Listing 162: show_null_procedure.ads

```
1 package Show_Null_Procedure is
2   type Element is limited null record;
3   -- Not implemented yet
4
```

(continues on next page)

²¹⁸ <https://www.adacore.com/gems/ada-gem-24>

(continued from previous page)

```

5  type Ref_Element is access all Element;
6
7  type Table is limited null record;
8  -- Not implemented yet
9
10 type Iterate_Action is
11   access procedure
12     (X : not null Ref_Element);
13
14 procedure Iterate
15   (T      : Table;
16    Action : Iterate_Action := null);
17 -- If Action is null, do nothing.
18
19 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

```

1  package Show_Null_Procedure is
2    type Element is limited null record;
3    -- Not implemented yet
4
5    type Ref_Element is access all Element;
6
7    type Table is limited null record;
8    -- Not implemented yet
9
10   procedure Do_Nothing
11     (X : not null Ref_Element) is null;
12
13   type Iterate_Action is
14     access procedure
15       (X : not null Ref_Element);
16
17   procedure Iterate
18     (T      : Table;
19      Action : not null Iterate_Action
20              := Do_Nothing'Access);
21
22 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

```
1 package Example is
2
3     type Element is limited private;
4     type Ref_Element is access all Element;
5
6     type Table is limited private;
7
8     type Iterate_Action is
9         access procedure
10            (X : not null Ref_Element);
11
12     procedure Iterate
13         (T : Table;
14          Action : Iterate_Action := null);
15     -- If Action is null, do nothing.
16
17     procedure Do_Nothing
18         (X : not null Ref_Element) is null;
19     procedure Iterate_2
20         (T : Table;
21          Action : not null Iterate_Action
22             := Do_Nothing'Access);
23
24 private
25     type Element is limited
26         record
27             Component : Integer;
28         end record;
29     type Table is limited null record;
30 end Example;
```

Listing 165: example.adb

```
1 package body Example is
2
3     An_Element : aliased Element;
4
5     procedure Iterate
6         (T : Table;
7          Action : Iterate_Action := null)
8     is
9     begin
10         if Action /= null then
11             Action (An_Element'Access);
12             -- In a real program, this would do
13             -- something more sensible.
14         end if;
15     end Iterate;
16
17     procedure Iterate_2
18         (T : Table;
19          Action : not null Iterate_Action
20             := Do_Nothing'Access)
21     is
22     begin
23         Action (An_Element'Access);
24         -- In a real program, this would do
25         -- something more sensible.
26     end Iterate_2;
```

(continues on next page)

(continued from previous page)

```
27
28 end Example;
```

Listing 166: show_example.adb

```
1 with Example; use Example;
2
3 procedure Show_Example is
4     T : Table;
5 begin
6     Iterate_2 (T);
7 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.

14.15.9 Access to protected subprograms

Up to this point, we've discussed access to *normal* Ada subprograms. In some situations, however, we might want to have access to protected subprograms. To do this, we can simply declare a type using **access protected**:

Listing 167: simple_protected_access.ads

```
1 package Simple_Protected_Access is
2
3     type Access_Proc is
4         access protected procedure;
5
6     protected Obj is
7
8         procedure Do_Something;
9
10    end Obj;
11
12    Acc : Access_Proc := Obj.Do_Something'Access;
13
14 end Simple_Protected_Access;
```

Listing 168: simple_protected_access.adb

```
1 package body Simple_Protected_Access is
2
3     protected body Obj is
4
5         procedure Do_Something is
6             begin
7                 -- Not doing anything
8                 -- for the moment...
9                 null;
```

(continues on next page)

(continued from previous page)

```
10     end Do_Something;
11
12     end Obj;
13
14 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`

```
1 package Work_Registry is
2
3     type Work_Id is tagged limited private;
4
5     type Work_Handler is
6         access protected procedure (T : Work_Id);
7
8     subtype Valid_Work_Handler is
9         not null Work_Handler;
10
11    type Work_Handlers is
12        array (Positive range <>) of Work_Handler;
13
14    protected type Work_Handler_Registry
15        (Last : Positive)
16    is
17
18        procedure Register (T : Valid_Work_Handler);
19
20        procedure Reset;
21
22        procedure Process_All;
23
24    private
25
26        D      : Work_Handlers (1 .. Last);
27        Curr  : Natural := 0;
28
29    end Work_Handler_Registry;
30
31 private
32
33     type Work_Id is tagged limited null record;
34
35 end Work_Registry;
```

Listing 170: `work_registry.adb`

```
1 package body Work_Registry is
2
```

(continues on next page)

(continued from previous page)

```

3  protected body Work_Handler_Registry is
4
5      procedure Register (T : Valid_Work_Handler)
6      is
7      begin
8          if Curr < Last then
9              Curr := Curr + 1;
10             D (Curr) := T;
11         end if;
12     end Register;
13
14     procedure Reset is
15     begin
16         Curr := 0;
17     end Reset;
18
19     procedure Process_All is
20     Dummy_ID : Work_Id;
21     begin
22         for I in D'First .. Curr loop
23             D (I).all (Dummy_ID);
24         end loop;
25     end Process_All;
26
27 end Work_Handler_Registry;
28
29 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`

```
1 with Work_Registry;
2
3 package Integer_Storage_System is
4
5     protected type Integer_Storage is
6
7         procedure Set (V : Integer);
8
9         procedure Show (T : Work_Registry.Work_Id);
10
11     private
12
13         I : Integer := 0;
14
15     end Integer_Storage;
16
17     type Integer_Storage_Access is
18         access Integer_Storage;
19
20     type Integer_Storage_Array is
21         array (Positive range <>) of
22             Integer_Storage_Access;
23
24 end Integer_Storage_System;
```

Listing 172: `integer_storage_system.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Integer_Storage_System is
4
5     protected body Integer_Storage is
6
7         procedure Set (V : Integer) is
8             begin
9                 I := V;
10            end Set;
11
12        procedure Show (T : Work_Registry.Work_Id)
13            is
14                pragma Unreferenced (T);
15            begin
16                Put_Line ("Value: " & Integer'Image (I));
17            end Show;
18
19        end Integer_Storage;
20
21 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

```

1  with Work_Registry;
2  use  Work_Registry;
3
4  with Integer_Storage_System;
5  use  Integer_Storage_System;
6
7  procedure Show_Access_To_Protected_Subprograms is
8
9      WHR      : Work_Handler_Registry (5);
10     Int_Stor : Integer_Storage_Array (1 .. 3);
11
12  begin
13     -- Allocate and initialize integer storage
14     --
15     -- (For the initialization, we're just
16     -- assigning the index here, but we could
17     -- really have used any integer value.)
18
19     for I in Int_Stor'Range loop
20         Int_Stor (I) := new Integer_Storage;
21         Int_Stor (I).Set (I);
22     end loop;
23
24     -- Register handlers
25
26     for I in Int_Stor'Range loop
27         WHR.Register (Int_Stor (I).all.Show'Access);
28     end loop;
29
30     -- Now, use Process_All to call the handlers
31     -- (in this case, the Show procedure for
32     -- each protected object from Int_Stor).
33
34     WHR.Process_All;
35
36 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

```
1 with Work_Registry;
2 use Work_Registry;
3
4 with Integer_Storage_System;
5 use Integer_Storage_System;
6
7 procedure Show_Access_To_Protected_Subprograms
8 is
9     WHR      : Work_Handler_Registry (5);
10
11     Int_Stor : array (1 .. 3) of Integer_Storage;
12
13 begin
14     -- Allocate and initialize integer storage
15     --
16     -- (For the initialization, we're just
17     -- assigning the index here, but we could
18     -- really have used any integer value.)
19
20     for I in Int_Stor'Range loop
21         -- Int_Stor (I) := new Integer_Storage;
22         Int_Stor (I).Set (I);
23     end loop;
24
25     -- Register handlers
26
27     for I in Int_Stor'Range loop
28         WHR.Register (Int_Stor (I).Show'Access);
29         --           ^ ERROR!
30     end loop;
31
32     -- Now, call the handlers
33     -- (i.e. the Show procedure of each
34     -- protected object).
35
36     WHR.Process_All;
37
38 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.

14.16 Accessibility Rules and Access-To-Subprograms

In general, the accessibility rules that we discussed *previously for access-to-objects* (page 534) also apply to access-to-subprograms. In this section, we discuss minor differences when applying those rules to access-to-subprograms.

In our discussion about accessibility rules, we've looked into *accessibility levels* (page 535) and the *accessibility rules* (page 536) that are based on those levels. The same accessibility rules apply to access-to-subprograms. *As we said previously* (page 539), operations targeting objects at a *less-deep* level are illegal, as it's the case for subprograms as well:

Listing 175: `access_to_subprogram_types.ads`

```
1 package Access_To_Subprogram_Types is
2
3   type Access_To_Procedure is
4     access procedure (I : in out Integer);
5
6   type Access_To_Function is
7     access function (I : Integer) return Integer;
8
9 end Access_To_Subprogram_Types;
```

Listing 176: show_access_to_subprogram_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7   Func : Access_To_Function;
8
9   Value : Integer := 0;
10 begin
11   declare
12     function Add_One (I : Integer)
13                       return Integer is
14       (I + 1);
15   begin
16     Func := Add_One'Access;
17     -- This assignment is illegal because the
18     -- Access_To_Function type is less deep
19     -- than Add_One.
20   end;
21
22   Put_Line ("Value: " & Value'Image);
23   Value := Func (Value);
24   Put_Line ("Value: " & Value'Image);
25 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
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

```
1 package Access_To_Subprogram_Types is
2
3   type Access_To_Procedure is
4     access procedure (I : in out Integer);
5
6   type Access_To_Function is
7     access function (I : Integer) return Integer;
8
9 end Access_To_Subprogram_Types;
```

Listing 178: add_one.ads

```
1 function Add_One (I : Integer) return Integer;
```

Listing 179: add_one.adb

```

1 function Add_One (I : Integer) return Integer is
2 begin
3     return I + 1;
4 end Add_One;
```

Listing 180: show_access_to_subprogram_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 with Add_One;
7
8 procedure Show_Access_To_Subprogram_Error is
9     Func : Access_To_Function;
10
11     Value : Integer := 0;
12 begin
13     Func := Add_One'Access;
14
15     Put_Line ("Value: " & Value'Image);
16     Value := Func (Value);
17     Put_Line ("Value: " & Value'Image);
18 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.

14.16.1 Unchecked Access

Previously, we discussed about the *Unchecked_Access attribute* (page 544), which we can use to circumvent accessibility issues in specific cases for access-to-objects. We also said in that section that this attribute only exists for objects, not for subprograms. We can use the previous example to illustrate this limitation:

Listing 181: access_to_subprogram_types.ads

```

1 package Access_To_Subprogram_Types is
2
3     type Access_To_Procedure is
4         access procedure (I : in out Integer);
5
6     type Access_To_Function is
7         access function (I : Integer) return Integer;
```

(continues on next page)

(continued from previous page)

```
8
9 end Access_To_Subprogram_Types;
```

Listing 182: show_access_to_subprogram_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7     Func : Access_To_Function;
8
9     function Add_One (I : Integer)
10         return Integer is
11         (I + 1);
12
13     Value : Integer := 0;
14 begin
15     Func := Add_One'Access;
16
17     Put_Line ("Value: " & Value'Image);
18     Value := Func (Value);
19     Put_Line ("Value: " & Value'Image);
20 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
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

```
1 package Access_To_Subprogram_Types is
2
3     type Access_To_Procedure is
4         access procedure (I : in out Integer);
```

(continues on next page)

(continued from previous page)

```

5
6  type Access_To_Function is
7     access function (I : Integer) return Integer;
8
9  end Access_To_Subprogram_Types;

```

Listing 184: show_access_to_subprogram_error.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Access_To_Subprogram_Types;
4  use Access_To_Subprogram_Types;
5
6  procedure Show_Access_To_Subprogram_Error is
7     Func : Access_To_Function;
8
9     function Add_One (I : Integer)
10        return Integer is
11        (I + 1);
12
13    Value : Integer := 0;
14  begin
15    Func := Add_One'Unrestricted_Access;
16    --
17    --     Allowing access to local function
18
19    Put_Line ("Value: " & Value'Image);
20    Value := Func (Value);
21    Put_Line ("Value: " & Value'Image);
22  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.

14.17 Access and Address

As we know, an access type is not a pointer, and it doesn't just indicate an address in memory. In fact, to represent an address in Ada, we use *the Address type* (page 131). Also, as we discussed earlier, we can use operators such as `<`, `>`, `+` and `-` for addresses. In contrast to that, those operators aren't available for access types — except, of course, for `=` and `/=`.

In certain situations, however, we might need to convert between access types and addresses. In this section, we discuss how to do so.

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes²¹⁹
 - 13.7 The Package System²²⁰
-

14.17.1 Address and access conversion

The generic `System.Address_To_Access_Conversions` package allows us to convert between access types and addresses. This might be useful for specific low-level operations. Let's see an example:

Listing 185: `show_address_conversion.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with System.Address_To_Access_Conversions;
4 with System.Address_Image;
5
6 procedure Show_Address_Conversion is
7
8     package Integer_AAC is
9         new System.Address_To_Access_Conversions
10            (Object => Integer);
11     use Integer_AAC;
12
13     subtype Integer_Access is
14         Integer_AAC.Object_Pointer;
15     -- This is similar to:
16     --
17     -- type Integer_Access is access all Integer;
18
19     I : aliased Integer := 5;
20     AI : Integer_Access := I'Access;
21 begin
22     Put_Line ("I'Address : "
23             & System.Address_Image (I'Address));
24
25     Put_Line ("AI.all'Address : "
26             & System.Address_Image
27             (AI.all'Address));
28
29     Put_Line ("To_Address (AI) : "
30             & System.Address_Image
31             (To_Address (AI)));
32 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 : 00007FFC6DB64054
AI.all'Address : 00007FFC6DB64054
To_Address (AI) : 00007FFC6DB64054
```

²¹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

²²⁰ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

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 402) fail because we have to dereference the access object (via `.all`) before retrieving its address (via the `Address` attribute).

In addition to the `To_Address` function, the `To_Pointer` function is available to convert from an address to an object of access type. For example:

Listing 186: `show_address_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with System;      use System;
3
4  with System.Address_To_Access_Conversions;
5  with System.Address_Image;
6
7  procedure Show_Address_Conversion is
8
9      package Integer_AAC is
10         new System.Address_To_Access_Conversions
11             (Object => Integer);
12         use Integer_AAC;
13
14         subtype Integer_Access is
15             Integer_AAC.Object_Pointer;
16
17         I      : aliased Integer := 5;
18         AI_1, AI_2 : Integer_Access;
19         A      : Address;
20     begin
21         AI_1 := I'Access;
22         A    := To_Address (AI_1);
23         AI_2 := To_Pointer (A);
24
25         Put_Line ("AI_1.all'Address : "
26                 & System.Address_Image
27                 (AI_1.all'Address));
28         Put_Line ("AI_2.all'Address : "
29                 & System.Address_Image
30                 (AI_2.all'Address));
31
32         if AI_1 = AI_2 then
33             Put_Line ("AI_1 = AI_2");
34         else
35             Put_Line ("AI_1 /= AI_2");
36         end if;
37     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 : 00007FFF064581CC
AI_2.all'Address : 00007FFF064581CC
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`

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with System;      use System;
3
4  with System.Address_To_Access_Conversions;
5  with System.Address_Image;
6
7  procedure Show_Address_Conversion is
8
9      package String_AAC is
10         new System.Address_To_Access_Conversions
11             (Object => String);
12         use String_AAC;
13
14         subtype Integer_Access is
15             String_AAC.Object_Pointer;
16
17         S      : aliased String := "Hello";
18         AI_1, AI_2 : Integer_Access;
19         A      : Address;
20     begin
21         AI_1 := S'Access;
22         A   := To_Address (AI_1);
23
24         AI_2 := To_Pointer (A);
25         --      ^^^^^^^^^^^^^^^^^
26         --      WARNING: Result might not have bounds
27
28         Put_Line ("AI_1.all'Address : "
29                 & System.Address_Image
30                 (AI_1.all'Address));
31         Put_Line ("AI_2.all'Address : "
32                 & System.Address_Image
33                 (AI_2.all'Address));
34
35         if AI_1 = AI_2 then
36             Put_Line ("AI_1 = AI_2");
37         else
38             Put_Line ("AI_1 /= AI_2");
39         end if;
40
```

(continues on next page)

(continued from previous page)

```
41 Put_Line ("AI_1: " & AI_1.all);
42 Put_Line ("AI_2: " & AI_2.all);
43 --
44 -- WARNING: As AI_2 might not have bounds
45 --           due to the call to To_Pointer
46 --           the behavior of this call to
47 --           the "&" operator is
48 --           unpredictable.
49 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 : 00007FFE7BEC388
AI_2.all'Address : 00007FFE7BEC388
AI_1 = AI_2
AI_1: Hello
AI_2: Hello
```

In this case, the call to `To_Pointer` (A) might not have bounds, so any operation on `AI_2` might lead to unpredictable results.

In the Ada Reference Manual

- [13.7.2 The Package System.Address_To_Access_Conversions](#)²²¹

²²¹ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-2.html>

ANONYMOUS ACCESS TYPES

15.1 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 625) and it's specifying an *anonymous access-to-object type* (page 605). Another flavor of anonymous access types are *anonymous access-to-subprograms* (page 648). We discuss all these topics in more details later.

Let's see a complete example:

Listing 1: show_anonymous_access_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Show_Anonymous_Access_Types is  
4   I_Var : aliased Integer;  
5  
6   A      : access Integer;  
7   --      ^ Anonymous access type  
8 begin  
9   A := I_Var'Access;  
10  --      ^ Assignment to object of  
11  --      anonymous access type.  
12  
13  A.all := 22;  
14  
15  Put_Line ("A.all: " & Integer'Image (A.all));  
16 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](#)²²²

15.1.1 Relation to named types

Anonymous access types were not part of the first version of the Ada standard, which only had support for named access types. They were introduced later to cover some use-cases that were difficult — or even impossible — with access types.

In this sense, anonymous access types aren't just access types without names. Certain accessibility rules for anonymous access types are a bit less strict. In those cases, it might be interesting to consider using them instead of named access types.

In general, however, we should only use anonymous access types in those specific cases where using named access types becomes too cumbersome. As a general recommendation, we should give preference to named access types whenever possible. (Anonymous access-to-object types have *drawbacks that we discuss later* (page 608).)

15.1.2 Benefits of anonymous access types

One of the main benefits of anonymous access types is their flexibility: since there isn't an explicit access type declaration associated with them, we only have to worry about the subtype S we intend to access.

Also, as long as the subtype S in a declaration `access S` is always the same, no conversion is needed between two access objects of that anonymous type, and the S'Access attribute always works.

Let's see an example:

Listing 2: show.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show (Name : String;
4               V    : access Integer) is
5 begin
6   Put_Line (Name & ".all: "
7             & Integer'Image (V.all));
8 end Show;
```

²²² <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Listing 3: show_anonymous_access_types.adb

```

1 with Show;
2
3 procedure Show_Anonymous_Access_Types is
4   I_Var : aliased Integer;
5   A     : access Integer;
6   B     : access Integer;
7 begin
8   A := I_Var'Access;
9   B := A;
10
11   A.all := 22;
12
13   Show ("A", A);
14   Show ("B", B);
15 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

```

1 package Aux is
2
3   type Integer_Access is access all Integer;
4
5   procedure Show (Name : String;
6                 V     : Integer_Access);
7
8 end Aux;
```

Listing 5: aux.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Aux is
4
5   procedure Show (Name : String;
6                 V     : Integer_Access) is
7
8   begin
```

(continues on next page)

(continued from previous page)

```
8     Put_Line (Name & ".all: "  
9               & Integer'Image (V.all));  
10    end Show;  
11  
12 end Aux;
```

Listing 6: show_anonymous_access_types.adb

```
1 with Aux; use Aux;  
2  
3 procedure Show_Anonymous_Access_Types is  
4     -- I_Var : aliased Integer;  
5  
6     A : Integer_Access;  
7     B : Integer_Access;  
8 begin  
9     -- A := I_Var'Access;  
10    --     ^ ERROR: non-local pointer cannot  
11    --     point to local object.  
12  
13    A := new Integer;  
14    B := A;  
15  
16    A.all := 22;  
17  
18    Show ("A", A);  
19    Show ("B", B);  
20 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

```

1 procedure Show_Anonymous_Access_Subtype_Error is
2   subtype Integer_1_10 is Integer range 1 .. 10;
3
4   I_Var : aliased Integer;
5   A     : access Integer := I_Var'Access;
6   B     : access Integer_1_10;
7 begin
8   A := I_Var'Access;
9
10  B := A;
11  -- ^ ERROR: subtype doesn't match!
12
13  B := I_Var'Access;
14  -- ^ ERROR: subtype doesn't match!
15 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 605) and *anonymous access-to-subprograms* (page 648) cover more specific details on anonymous access types.

15.2 Anonymous Access-To-Object Types

In the *previous chapter* (page 481), we introduced named access-to-object types and used those types throughout the chapter. Also, in the *previous section* (page 601), we've seen some simple examples of anonymous access-to-object types:

```

procedure Add_One (A : access Integer);
--           ^ Anonymous access type

A : access Integer;
-- ^ Anonymous access type

```

In addition to parameters and objects, we can use anonymous access types in discriminants, components of array and record types, renamings and function return types. (We discuss *anonymous access discriminants* (page 615) and *anonymous access parameters* (page 625) later on.) Let's see a code example that includes all these cases:

Listing 8: all_anonymous_access_to_object_types.ads

```
1 package All_Anonymous_Access_To_Object_Types is
2
3   procedure Add_One (A : access Integer) is null;
4     -- ^ Anonymous access type
5
6   AI : access Integer;
7     -- ^ Anonymous access type
8
9   type Rec (AI : access Integer) is private;
10    -- ^ Anonymous access type
11
12   type Access_Array is
13     array (Positive range <>) of
14       access Integer;
15     -- ^ Anonymous access type
16
17   Arr : array (1 .. 5) of access Integer;
18     -- ^ Anonymous access type
19
20   AI_Renaming : access Integer renames AI;
21     -- ^ Anonymous access type
22
23   function Init_Access_Integer
24     return access Integer is (null);
25     -- ^ Anonymous access type
26
27 private
28
29   type Rec (AI : access Integer) is record
30     -- ^ Anonymous access type
31     Internal_AI : access Integer;
32     -- ^ Anonymous access type
33
34   end record;
35
36 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 Types²²³

15.2.1 Not Null Anonymous Access-To-Object Types

As expected, `null` is a valid value for an anonymous access type. However, we can forbid `null` as a valid value by using `not null` in the anonymous access type declaration. For example:

Listing 9: all_anonymous_access_to_object_types.ads

```

1 package All_Anonymous_Access_To_Object_Types is
2
3   procedure Add_One (A : not null access Integer)
4     is null;
5     --           ^ Anonymous access type
6
7   I : aliased Integer;
8
9   AI : not null access Integer := I'Access;
10  -- ^ Anonymous access type
11  --           ~~~~~
12  --           Initialization required!
13
14  type Rec (AI : not null access Integer) is
15    private;
16    --           ^ Anonymous access type
17
18  type Access_Array is
19    array (Positive range <>) of
20      not null access Integer;
21    -- ^ Anonymous access type
22
23  Arr : array (1 .. 5) of
24    not null access Integer :=
25    -- ^ Anonymous access type
26    (others => I'Access);
27    -- ~~~~~
28    --           Initialization required!
29
30  AI_Renaming : not null access Integer
31    renames AI;
32    --           ^ Anonymous access type
33
34  function Init_Access_Integer
35    return not null access Integer is (I'Access);
36    -- ^ Anonymous access type
37    --           ~~~~~
38    --           Initialization required!
39
40 private
41
42  type Rec (AI : not null access Integer) is
43    record
44    --           ^ Anonymous access type
45      Internal_AI : not null access Integer;
46    --           ^ Anonymous access type
47
48  end record;

```

(continues on next page)

²²³ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```
49  
50 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.

15.2.2 Drawbacks of Anonymous Access-To-Object Types

Anonymous access-to-object types have important drawbacks. For example, some features that are available for named access types aren't available for the anonymous access types. Also, most of the drawbacks are related to how anonymous access-to-object types can potentially make the allocation and deallocation quite complicated or even error-prone.

For starters, some pool-related features aren't available for anonymous access-to-object types. For example, we cannot specify which pool is going to be used in the allocation of an anonymous access-to-object. In fact, the memory pool selection is compiler-dependent, so we cannot rely on an object being allocated from a specific pool when using **new** with an anonymous access-to-object type. (In contrast, as we know, each named access type has an associated pool, so objects allocated via **new** will be allocated from that pool.) Also, we cannot identify which pool was selected for the allocation of a specific object, so we don't have any information to use for the deallocation of that object.

Because the pool selection is hidden from us, this makes the memory deallocation more complicated. For example, we cannot instantiate the `Ada.Unchecked_Deallocation` procedure for anonymous access types. Also, some of the methods we could use to circumvent this limitation are error-prone, as we discuss in this section.

Also, storage-related features aren't available: specifying the storage size — especially, specifying that the access type has a storage size of zero — isn't possible.

Missing features

Let's see a code example that shows some of the features that aren't available for anonymous access-to-object types:

Listing 10: missing_features.ads

```
1 with Ada.Unchecked_Deallocation;  
2  
3 package Missing_Features is  
4
```

(continues on next page)

(continued from previous page)

```

5  -- We cannot specify which pool will be used
6  -- in the anonymous access-to-object
7  -- allocation; the pool is selected by the
8  -- compiler:
9  IA : access Integer := new Integer;
10
11  --
12  -- All the features below aren't available
13  -- for an anonymous access-to-object:
14  --
15
16  -- Having a specific storage pool associated
17  -- with the access type:
18  type String_Access is
19      access String;
20  -- Automatically creates
21  -- String_Access'Storage_Pool
22
23  type Integer_Access is
24      access Integer
25      with Storage_Pool =>
26          String_Access'Storage_Pool;
27  -- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
28  -- Using the pool from another
29  -- access type.
30
31  -- Specifying a deallocation function for the
32  -- access type:
33  procedure Free is
34      new Ada.Unchecked_Deallocation
35      (Object => Integer,
36       Name  => Integer_Access);
37
38  -- Specifying a limited storage size for
39  -- the access type:
40  type Integer_Access_Store_128 is
41      access Integer
42      with Storage_Size => 128;
43
44  -- Limiting the storage size for the
45  -- access type to zero:
46  type Integer_Access_Store_0 is
47      access Integer
48      with Storage_Size => 0;
49
50 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`

```
1 with Ada.Unchecked_Deallocation;
2
3 procedure Show_Dangerous_Deallocation is
4   type Integer_Access is
5     access all Integer;
6
7   procedure Free is
8     new Ada.Unchecked_Deallocation
9       (Object => Integer,
10        Name  => Integer_Access);
11
12   IA : access Integer;
13 begin
14   IA := new Integer;
15   IA.all := 30;
16
17   -- Potentially erroneous deallocation via type
18   -- conversion:
19   Free (Integer_Access (IA));
20
21 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`

```
1 pragma Restrictions (No_Anonymous_Allocators);
2
3 procedure Show_Dangerous_Allocation is
4   IA : access Integer;
5 begin
6   IA := new Integer;
7   IA.all := 30;
8 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`

```
1 with Ada.Unchecked_Deallocation;
2
3 procedure Show_Successful_Deallocation is
4
5   type Integer_Access is
6     access Integer;
7
8   procedure Free is
9     new Ada.Unchecked_Deallocation
10      (Object => Integer,
11       Name   => Integer_Access);
12
13   IA      : access Integer;
14   Typed_IA : Integer_Access;
15
16 begin
17   Typed_IA := new Integer;
18   IA := Typed_IA;
19   IA.all := 30;
```

(continues on next page)

(continued from previous page)

```
20
21  -- Deallocation of the access object that has
22  -- an associated type:
23  Free (Typed_IA);
24
25 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 create problems afterwards in the object's deallocation. (When calling `Free`, we only refer to the object of named access type, so the object is deallocated from a known pool.)

Of course, a potential issue here is that `IA` becomes a *dangling reference* (page 541) after the call to `Free`. Therefore, we can improve this solution by completely hiding the memory allocation and deallocation for the anonymous access types in subprograms — e.g. as part of a package. By doing so, we don't expose the named access type, thereby reducing the possibility of dangling references.

In fact, we can generalize this approach with the following (generic) package:

Listing 14: `hidden_anonymous_allocation.ads`

```
1 generic
2   type T is private;
3 package Hidden_Anonymous_Allocation is
4
5   function New_T
6     return not null access T;
7
8   procedure Free (Obj : access T);
9
10 end Hidden_Anonymous_Allocation;
```

Listing 15: `hidden_anonymous_allocation.adb`

```
1 with Ada.Unchecked_Deallocation;
2
3 package body Hidden_Anonymous_Allocation is
4
5   type T_Access is access all T;
6
7   procedure T_Access_Free is
8     new Ada.Unchecked_Deallocation
9     (Object => T,
10      Name  => T_Access);
11
12   function New_T
13     return not null access T is
14   begin
15     return T_Access'(new T);
16   -- Using allocation of the T_Access type:
```

(continues on next page)

(continued from previous page)

```

17     -- object is allocated from T_Access's pool
18 end New_T;
19
20 procedure Free (Obj : access T) is
21     Tmp : T_Access := T_Access (Obj);
22 begin
23     T_Access_Free (Tmp);
24     -- Using deallocation procedure of the
25     -- T_Access type
26 end Free;
27
28 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

```

1 with Hidden_Anonymous_Allocation;
2
3 package Info is
4
5     type Rec is private;
6
7     function New_Rec return not null access Rec;
8
9     procedure Free (Obj : access Rec);
10
11 private
12
13     type Rec is record
14         I : Integer;
15     end record;
16
17     package Rec_Allocation is new
18         Hidden_Anonymous_Allocation (T => Rec);
19
20     function New_Rec return not null access Rec
21         renames Rec_Allocation.New_T;
22
23     procedure Free (Obj : access Rec)
24         renames Rec_Allocation.Free;
25
26 end Info;

```


Listing 17: show_info_allocation_deallocation.adb

```
1 with Info; use Info;
2
3 procedure Show_Info_Allocation_Deallocation is
4   RA : constant not null access Rec := New_Rec;
5 begin
6   Free (RA);
7 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

```
1 procedure Show_Automatic_Deallocation is
2   I : aliased Integer;
3   -- ^ Allocating object on the stack
4
5   IA : access Integer;
6 begin
7   IA := I'Access;
8   -- Indirect allocation:
9   -- object creation on the stack.
10
11  IA.all := 30;
12
13  -- Automatic deallocation at the end of the
14  -- procedure because the integer variable is
15  -- on the stack.
16 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 602). In fact, allocation for those types can quickly become very complicated. Therefore, in general, they're not a good alternative to named access types. Indeed, the difficulties that we've just seen might make them a much worse option than just using named access types instead.

We might consider using anonymous access-to-objects types only in cases when we reach a point in our implementation work where using named access types becomes impossible — or when using them becomes even more complicated than equivalent solutions using anonymous access types. This scenario, however, is usually the exception rather than the rule. Thus, as a general guideline, we should always aim to use named access types.

That being said, an important exception to this advice is when we're *interfacing to other languages* (page 628). In this case, as we'll discuss later, using anonymous access-to-objects types can be significantly simpler (compared to named access types) without the drawbacks that we've just discussed.

15.3 Access discriminants

Previously, we've discussed *discriminants as access values* (page 492). In that section, we only used named access types. Now, in this section, we see how to use anonymous access types as discriminants. This feature is also known as *access discriminants* and it provides some flexibility that can be interesting in terms of software design, as we'll discuss later.

Let's start with an example:

Listing 19: custom_rec.ads

```

1 package Custom_Recs is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type:
5   type Rec (IA : access Integer) is record
6     I : Integer := IA.all;
7   end record;
8
9   procedure Show (R : Rec);
10
11 end Custom_Recs;
```

Listing 20: custom_rec.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Recs is
4
5   procedure Show (R : Rec) is
6   begin
7     Put_Line ("R.IA = "
8               & Integer'Image (R.IA.all));
9     Put_Line ("R.I = "
10              & Integer'Image (R.I));
11   end Show;
12
13 end Custom_Recs;
```

Listing 21: show_access_discriminants.adb

```
1 with Custom_Recs; use Custom_Recs;
2
3 procedure Show_Access_Discriminants is
4   I : aliased Integer := 10;
5   R : Rec (I'Access);
6 begin
7   Show (R);
8
9   I := 20;
10  R.I := 30;
11  Show (R);
12 end Show_Access_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↳Discriminants.Custom_Recs
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

```
1 package Persons is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type whose designated subtype is
5   -- unconstrained:
6   type Person (Name : access String) is record
7     Age : Integer;
8   end record;
9
10  procedure Show (P : Person);
11
12 end Persons;
```

Listing 23: persons.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Persons is
4
5   procedure Show (P : Person) is
6   begin
7     Put_Line ("Name = "
8               & P.Name.all);
9     Put_Line ("Age = "
10              & Integer'Image (P.Age));
```

(continues on next page)

(continued from previous page)

```

11   end Show;
12
13 end Persons;

```

Listing 24: show_person.adb

```

1 with Persons; use Persons;
2
3 procedure Show_Person is
4   S : aliased String := "John";
5   P : Person (S'Access);
6 begin
7   P.Age := 30;
8   Show (P);
9 end Show_Person;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_↵Discriminants.Persons
 MD5: f0149d572e0ec192476836bfd00dd9e

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](#)²²⁴
 - [3.10.2 Operations of Access Types](#)²²⁵
-

15.3.1 Default Value of Access Discriminants

In contrast to named access types, we cannot use a default value for the access discriminant of a non-limited type:

Listing 25: custom_recs.ads

```

1 package Custom_Recs is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type and a default value:
5   type Rec (IA : access Integer :=
6             new Integer'(0)) is
7
8     record
9       I : Integer := IA.all;
10    end record;
11
12   procedure Show (R : Rec);

```

(continues on next page)

²²⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-7.html>

²²⁵ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html>

(continued from previous page)

```
12
13 end Custom_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↳Discriminants.Custom_Recs
MD5: 9269cea113f29443a6d7bb719d0616f1
```

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

```
1 package Custom_Recs is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type and a default value:
5   type Rec (IA : access Integer :=
6             new Integer'(0)) is limited
7
8   record
9     I : Integer := IA.all;
10  end record;
11
12  procedure Show (R : Rec);
13 end Custom_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↳Discriminants.Custom_Recs
MD5: 9e8683c7a27e9097fd2003ad91bac269
```

Build output

```
custom_recs.ads:6:21: warning: coextension will not be deallocated when its
↳associated owner is deallocated [enabled by default]
```

Note that, if we don't provide a value for the access discriminant when declaring an object R, the default value is allocated (via **new**) during R's creation.

Listing 27: show_access_discriminants.adb

```
1 with Custom_Recs; use Custom_Recs;
2
3 procedure Show_Access_Discriminants is
4   R : Rec;
5   --   ^^
6   -- This triggers "new Integer'(0)", so an
7   -- integer object is allocated and stored in
8   -- the R.IA discriminant.
9 begin
```

(continues on next page)

(continued from previous page)

```

10 Show (R);
11
12 -- R gets out of scope here, and the object
13 -- allocated via new hasn't been deallocated.
14 end Show_Access_Discriminants;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↳Discriminants.Custom_Recs
MD5: f5d9dee26044ccab2193ab419638de79
```

Build output

```

show_access_discriminants.adb:4:04: warning: coextension will not be deallocated_
↳when its associated owner is deallocated [enabled by default]
custom_recs.ads:6:21: warning: coextension will not be deallocated when its_
↳associated owner is deallocated [enabled by default]
```

Runtime output

```

R.IA = 0
R.I = 0
```

In this case, the allocated object won't be deallocated when R gets out of scope!

15.3.2 Benefits of Access Discriminants

Access discriminants have the same benefits that we've already seen earlier while discussing *discriminants as access values* (page 492). An additional benefit is its extended flexibility: access discriminants are compatible with any access T'Access, as long as T is of the designated subtype.

Consider the following example using the named access type Access_String:

Listing 28: persons.ads

```

1 package Persons is
2
3     type Access_String is access all String;
4
5     -- Declaring a discriminant with a named
6     -- access type:
7     type Person (Name : Access_String) is record
8         Age : Integer;
9     end record;
10
11     procedure Show (P : Person);
12
13 end Persons;
```

Listing 29: persons.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Persons is
4
5     procedure Show (P : Person) is
6     begin
```

(continues on next page)

(continued from previous page)

```
7     Put_Line ("Name = "
8             & P.Name.all);
9     Put_Line ("Age = "
10            & Integer'Image (P.Age));
11 end Show;
12
13 end Persons;
```

Listing 30: show_person.adb

```
1 with Persons; use Persons;
2
3 procedure Show_Person is
4   S : aliased String := "John";
5   P : Person (S'Access);
6   --      ^^^^^^^ ERROR: cannot use local
7   --                      object
8   --
9   -- We can, however, allocate the string via
10  -- new:
11  --
12  -- S : Access_String := new String("John");
13  -- P : Person (S);
14 begin
15   P.Age := 30;
16   Show (P);
17 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 31: persons.ads

```
1 package Persons is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type:
5   type Person (Name : access String) is record
6     Age : Integer;
7   end record;
8
9   procedure Show (P : Person);
10
11 end Persons;
```

Listing 32: show_person.adb

```

1 with Persons; use Persons;
2
3 procedure Show_Person is
4     S : aliased String := "John";
5     P : Person (S'Access);
6     --      ^^^^^^^ OK
7
8     -- Allocating the string via new and using it
9     -- in P's declaration is OK as well, but we
10    -- should manually deallocate it before S
11    -- gets out of scope:
12    --
13    -- S : access String := new String("John");
14    -- P : Person (S);
15 begin
16     P.Age := 30;
17     Show (P);
18 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.

15.3.3 Preventing dangling pointers

Note that the usual rules that prevent dangling pointers still apply here. This ensures that we can safely use access discriminants. For example:

Listing 33: show_person.adb

```

1 with Persons; use Persons;
2
3 procedure Show_Person is
4
5     function Local_Init return Person is
6         S : aliased String := "John";
7         begin
8             return (Name => S'Access, Age => 30);
9             --      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
10            --      ERROR: dangling reference!
11         end Local_Init;
12
13     P : Person := Local_Init;
14 begin
15     Show (P);
16 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.

15.4 Self-reference

Previously, we've seen that we can declare *self-references* (page 509) using named access types. We can do the same with anonymous access types. Let's revisit the code example that implements linked lists:

Listing 34: `linked_lists.ads`

```
1 generic
2   type T is private;
3 package Linked_Lists is
4
5   type List is limited private;
6
7   procedure Append_Front
8     (L : in out List;
9      E : T);
10
11  procedure Append_Rear
12    (L : in out List;
13     E : T);
14
15  procedure Show (L : List);
16
17 private
18
19  type Component is record
20    Next : access Component;
21    -- ~~~~~
22    -- Self-reference
23    --
24    -- (Note that we haven't finished the
25    -- declaration of the "Component" type
26    -- yet, but we're already referring to
27    -- it.)
28
29    Value : T;
30  end record;
31
32  type List is access all Component;
33
34 end Linked_Lists;
```

Listing 35: linked_lists.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  package body Linked_Lists is
6
7      procedure Append_Front
8          (L : in out List;
9           E :          T)
10     is
11         New_First : constant List := new
12             Component'(Value => E,
13                        Next => L);
14     begin
15         L := New_First;
16     end Append_Front;
17
18     procedure Append_Rear
19         (L : in out List;
20          E :          T)
21     is
22         New_Last : constant List := new
23             Component'(Value => E,
24                      Next => null);
25     begin
26         if L = null then
27             L := New_Last;
28         else
29             declare
30                 Last : List := L;
31             begin
32                 while Last.Next /= null loop
33                     Last := List (Last.Next);
34                     --      ^^^^
35                     --      type conversion:
36                     --      "access Component" to
37                     --      "List"
38                 end loop;
39                 Last.Next := New_Last;
40             end;
41         end if;
42     end Append_Rear;
43
44     procedure Show (L : List) is
45         Curr : List := L;
46     begin
47         if L = null then
48             Put_Line ("[ ]");
49         else
50             Put ("[");
51             loop
52                 Put (Curr.Value'Image);
53                 Put (" ");
54                 exit when Curr.Next = null;
55                 Curr := Curr.Next;
56             end loop;
57             Put_Line ("]");
58         end if;
59     end Show;

```

(continues on next page)

(continued from previous page)

```
60
61 end Linked_Lists;
```

Listing 36: test_linked_list.adb

```
1 with Linked_Lists;
2
3 procedure Test_Linked_List is
4   package Integer_Lists is new
5     Linked_Lists (T => Integer);
6   use Integer_Lists;
7
8   L : List;
9 begin
10  Append_Front (L, 3);
11  Append_Rear (L, 4);
12  Append_Rear (L, 5);
13  Append_Front (L, 2);
14  Append_Front (L, 1);
15  Append_Rear (L, 6);
16  Append_Rear (L, 7);
17
18  Show (L);
19 end Test_Linked_List;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Self_Reference.Linked_List_Example
MD5: 9e42bf9fa630a0af8dcf7c85a1565edb

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.

15.5 Mutually dependent types using anonymous access types

In the section on *mutually dependent types using access types* (page 512), we've seen a code example that was using named access types. We could now rewrite it using anonymous access types:

Listing 37: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T2;
4
5     type T1 is record
6         B : access T2;
7     end record;
8
9     type T2 is record
10        A : access T1;
11    end record;
12
13 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.

15.6 Access parameters

In the previous chapter, we talked about *parameters as access values* (page 498). As you might have expected, we can also use anonymous access types as parameters of a subprogram. However, they're limited to be **in** parameters of a subprogram or return type of a function (also called the access result type):

Listing 38: names.ads

```

1 package Names is
2
3     function Init (S1, S2 : String)
4         return access String;
5         ~~~~~
6     -- Anonymous access type as the access
7     -- result type.
8
9     procedure Show (N : access constant String);
10        ~~~~~
11    -- Anonymous access type as a parameter type.
12
13 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 39: names.ads

```
1 package Names is
2
3     function Init (S1, S2 : String)
4         return access String;
5
6     procedure Show (N : access constant String);
7
8 private
9
10    function Init (S1, S2 : String)
11        return access String is
12        (new String'(S1 & "-" & S2));
13
14 end Names;
```

Listing 40: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Names is
4
5     procedure Show (N : access constant String) is
6     begin
7         Put_Line ("Name: " & N.all);
8     end Show;
9
10 end Names;
```

Listing 41: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4     N : access String := Init ("Lily", "Ann");
5 begin
6     Show (N);
7 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 498) in the previous chapter, we

saw how we can simply use different parameter modes to write a program instead of using access types. Basically, to implement the same functionality, we just replaced the access types by selecting the correct parameter modes instead and used *simpler* data types.

Let's do the same exercise again, this time by adapting the previous code example with anonymous access types:

Listing 42: names.ads

```
1 package Names is
2
3     function Init (S1, S2 : String)
4                   return String;
5
6     procedure Show (N : String);
7
8 private
9
10    function Init (S1, S2 : String)
11                  return String is
12      (S1 & "-" & S2);
13
14 end Names;
```

Listing 43: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Names is
4
5     procedure Show (N : String) is
6     begin
7         Put_Line ("Name: " & N);
8     end Show;
9
10 end Names;
```

Listing 44: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4     N : String := Init ("Lily", "Ann");
5 begin
6     Show (N);
7 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 647).

In general, we should avoid access parameters whenever possible and simply use objects and parameter modes directly, as it makes the design simpler and less error-prone. One exception is when we're interfacing to other languages, especially C: this is our *next topic* (page 628). Another time when access parameters are vital is for inherited primitive operations for tagged types. We discuss this *later on* (page 631).

In the Ada Reference Manual

- [3.10 Access Types](#)²²⁶
-

15.6.1 Interfacing To Other Languages

We can use access parameters to interface to other languages. This can be particularly useful when interfacing to C code that makes use of pointers. For example, let's assume we want to call the `add_one` function below in our Ada implementation:

Listing 45: operations_c.h

```
1 void add_one(int *p_i);
```

Listing 46: operations_c.c

```
1 void add_one(int *p_i)
2 {
3     *p_i = *p_i + 1;
4 }
```

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 47: operations.ads

```
1 package Operations is
2
3     procedure Add_One (IA : access Integer)
4         with Import, Convention => C;
5
6 end Operations;
```

²²⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Listing 48: show_operations.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Operations; use Operations;
4
5 procedure Show_Operations is
6   I : aliased Integer := 42;
7 begin
8   Put_Line (I'Image);
9   Add_One (I'Access);
10  Put_Line (I'Image);
11 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 49: operations.ads

```

1 package Operations is
2
3   procedure Add_One
4     (IA : aliased in out Integer)
5     with Import, Convention => C;
6
7 end Operations;
```

Listing 50: show_operations.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Operations; use Operations;
4
5 procedure Show_Operations is
6   I : aliased Integer := 42;
7 begin
8   Put_Line (I'Image);
9   Add_One (I);
10  Put_Line (I'Image);
11 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 51: operations_c.h

```
1 int * alloc_integer();
2
3 void dealloc_integer(int *p_i);
4
5 void add_one(int *p_i);
```

Listing 52: operations_c.c

```
1 #include <stdlib.h>
2
3 int * alloc_integer()
4 {
5     return malloc(sizeof(int));
6 }
7
8 void dealloc_integer(int *p_i)
9 {
10    free (p_i);
11 }
12
13 void add_one(int *p_i)
14 {
15     *p_i = *p_i + 1;
16 }
```

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 53: operations.ads

```
1 package Operations is
2
3     function Alloc_Integer return access Integer
4         with Import, Convention => C;
5
6     procedure Dealloc_Integer (IA : access Integer)
7         with Import, Convention => C;
8
9     procedure Add_One
10        (IA : aliased in out Integer)
11        with Import, Convention => C;
12
13 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 54: show_operations.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Operations; use Operations;
4
5  procedure Show_Operations is
6      I : access Integer := Alloc_Integer;
7  begin
8      I.all := 42;
9      Put_Line (I.all'Image);
10
11     Add_One (I.all);
12     Put_Line (I.all'Image);
13
14     Dealloc_Integer (I);
15 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](#)²²⁷

15.6.2 Inherited Primitive Operations For Tagged Types

In order to declare inherited primitive operations for tagged types that use access types, we need to use access parameters. The reason is that, to be a primitive operation for some tagged type — and hence inheritable — the subprogram must reference the tagged type name directly in the parameter profile. This means that a named access type won't suffice, because only the access type name would appear in the profile. For example:

Listing 55: inherited_primitives.ads

```

1  package Inherited_Primitives is
2
3      type T is tagged private;
4
5      type T_Access is access all T;
6
7      procedure Proc (N : T_Access);
8      -- Proc is not a primitive of type T.
9
10     type T_Child is new T with private;
11
12     type T_Child_Access is access all T_Child;
```

(continues on next page)

²²⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```
13
14 private
15
16     type T is tagged null record;
17
18     type T_Child is new T with null record;
19
20 end Inherited_Primitives;
```

Listing 56: inherited_primitives.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Inherited_Primitives is
4
5     procedure Proc (N : T_Access) is null;
6
7 end Inherited_Primitives;
```

Listing 57: show_inherited_primitives.adb

```
1 with Inherited_Primitives;
2 use Inherited_Primitives;
3
4 procedure Show_Inherited_Primitives is
5     Obj      : T_Access      := new T;
6     Obj_Child : T_Child_Access := new T_Child;
7 begin
8     Proc (Obj);
9     Proc (Obj_Child);
10    --     ^^^^^^^^^^
11    --     ERROR: Proc is not inherited!
12 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 access parameter (**access** T), the subprogram becomes a primitive of T:

Listing 58: inherited_primitives.ads

```
1 package Inherited_Primitives is
2
```

(continues on next page)

(continued from previous page)

```

3  type T is tagged private;
4
5  procedure Proc (N : access T);
6  -- Proc is a primitive of type T.
7
8  type T_Child is new T with private;
9
10 private
11
12  type T is tagged null record;
13
14  type T_Child is new T with null record;
15
16 end Inherited_Primitives;
```

Listing 59: inherited_primitives.adb

```

1  package body Inherited_Primitives is
2
3  procedure Proc (N : access T) is null;
4
5  end Inherited_Primitives;
```

Listing 60: show_inherited_primitives.adb

```

1  with Inherited_Primitives;
2  use Inherited_Primitives;
3
4  procedure Show_Inherited_Primitives is
5  Obj      : access T      := new T;
6  Obj_Child : access T_Child := new T_Child;
7  begin
8  Proc (Obj);
9  Proc (Obj_Child);
10 -- ^^^^^^^^^^
11 -- OK: Proc is inherited!
12 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:

```

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 Types²²⁸

²²⁸ <http://www.ada-auth.org/standards/22rm/html/RM-3-9-2.html>

15.7 User-Defined References

Implicit dereferencing (page 514) isn't limited to the contexts that Ada supports by default: we can also add implicit dereferencing to our own types by using the `Implicit_Dereference` aspect.

To do this, we have to declare:

- a reference type, where we use the `Implicit_Dereference` aspect to specify the reference discriminant, which is the record discriminant that will be dereferenced; and
- a reference object, which contains an access value that will be dereferenced.

Also, for the reference type, we have to:

- specify the reference discriminant as an *access discriminant* (page 615); and
- indicate the name of the reference discriminant when specifying the `Implicit_Dereference` aspect.

Let's see a simple example:

Listing 61: show_user_defined_reference.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_User_Defined_Reference is
4
5      type Id_Number is record
6          Id : Positive;
7      end record;
8
9      --
10     -- Reference type:
11     --
12     type Id_Ref (Ref : access Id_Number) is
13         --      ^ reference discriminant
14         null record
15             with Implicit_Dereference => Ref;
16         --      ^^^
17         --      name of the reference
18         --      discriminant
19
20     --
21     -- Access value:
22     --
23     I : constant access Id_Number :=
24         new Id_Number'(Id => 42);
25
26     --
27     -- Reference object:
28     --
29     R : Id_Ref (I);
30 begin
31     Put_Line ("ID: "
32             & Positive'Image (R.Id));
33     --      ^ Equivalent to:
34     --      R.Ref.Id
35     --      or:
36     --      R.Ref.all.Id
37 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 References²²⁹

²²⁹ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-5.html>

15.7.1 Dereferencing of tagged types

Naturally, implicit dereferencing is also possible when calling primitives of a tagged type. For example, let's change the declaration of the `Id_Number` type from the previous code example and add a `Show` primitive.

Listing 62: info.ads

```
1 package Info is
2   type Id_Number (Id : Positive) is
3     tagged private;
4
5   procedure Show (R : Id_Number);
6 private
7   type Id_Number (Id : Positive) is
8     tagged null record;
9 end Info;
```

Listing 63: info.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Info is
4
5   procedure Show (R : Id_Number) is
6   begin
7     Put_Line ("ID: " & Positive'Image (R.Id));
8   end Show;
9
10 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 64: show_user_defined_reference.adb

```
1 with Info; use Info;
2
3 procedure Show_User_Defined_Reference is
4
5   -- Reference type:
6   type Id_Ref (Ref : access Id_Number) is
7     null record
8     with Implicit_Dereference => Ref;
9
10  -- Access value:
11  I : constant access Id_Number :=
12    new Id_Number (42);
13
14  -- Reference object:
15  R : Id_Ref (I);
16 begin
17  R.Show;
18  -- Equivalent to:
19  -- R.Ref.all.Show;
20
21
```

(continues on next page)

(continued from previous page)

```
22 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.

15.7.2 Simple container

A typical application of user-defined references is to create cursors when iterating over a container. As an example, let's implement the National_Date_Info package to store the national day of a country:

Listing 65: national_date_info.ads

```
1 package National_Date_Info is
2
3     subtype Country_Code is String (1 .. 3);
4
5     type Time is record
6         Year  : Integer;
7         Month : Positive range 1 .. 12;
8         Day   : Positive range 1 .. 31;
9     end record;
10
11    type National_Date is tagged record
12        Country : Country_Code;
13        Date    : Time;
14    end record;
15
16    type National_Date_Access is
17        access National_Date;
18
19    procedure Show (Nat_Date : National_Date);
20
21 end National_Date_Info;
```

Listing 66: national_date_info.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body National_Date_Info is
4
5     procedure Show (Nat_Date : National_Date) is
6     begin
7         Put_Line ("Country: "
8                 & Nat_Date.Country);
9         Put_Line ("Year: "
10                & Integer'Image
11                (Nat_Date.Date.Year));
12     end Show;
```

(continues on next page)

(continued from previous page)

```
13
14 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 67: `national_date_containers.ads`

```
1 with National_Date_Info; use National_Date_Info;
2
3 package National_Date_Containers is
4
5     -- Reference type:
6     type National_Date_Reference
7         (Ref : access National_Date) is
8         tagged limited null record
9         with Implicit_Dereference => Ref;
10
11     -- Container (as an array):
12     type National_Dates is
13         array (Positive range <>) of
14             National_Date_Access;
15
16     -- The Find function scans the container to
17     -- find a specific country, which is returned
18     -- as a reference object.
19     function Find (Nat_Dates : National_Dates;
20                  Country   : Country_Code)
21         return National_Date_Reference;
22
23 end National_Date_Containers;
```

Listing 68: `national_date_containers.adb`

```
1 package body National_Date_Containers is
2
3     function Find (Nat_Dates : National_Dates;
4                  Country   : Country_Code)
5         return National_Date_Reference
6
7     is
8     begin
9         for I in Nat_Dates'Range loop
10            if Nat_Dates (I).Country = Country then
11                return National_Date_Reference'(
12                    Ref => Nat_Dates (I));
13                -- ~~~~~
14                -- Returning reference object with a
15                -- reference to the national day we
16                -- found.
17            end if;
18        end loop;
19
20    return
```

(continues on next page)

(continued from previous page)

```

20     National_Date_Reference'(Ref => null);
21     -- ~~~~~
22     -- Returning reference object with a null
23     -- reference in case the country wasn't
24     -- found. This will trigger an exception
25     -- if we try to dereference it.
26 end Find;
27
28 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 69: `show_national_dates.adb`

```

1  with National_Date_Info;
2  use  National_Date_Info;
3
4  with National_Date_Containers;
5  use  National_Date_Containers;
6
7  procedure Show_National_Dates is
8
9      Nat_Dates : constant National_Dates (1 .. 5) :=
10     (new National_Date'("USA",
11         Time'(1776, 7, 4)),
12     new National_Date'("FRA",
13         Time'(1789, 7, 14)),
14     new National_Date'("DEU",
15         Time'(1990, 10, 3)),
16     new National_Date'("SPA",
17         Time'(1492, 10, 12)),
18     new National_Date'("BRA",
19         Time'(1822, 9, 7)));
20
21 begin
22     Find (Nat_Dates, "FRA").Show;
23     -- ^ implicit dereference
24 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 70: generic_containers.ads

```
1 generic
2   type T is private;
3   type T_Access is access T;
4   type T_Cmp is private;
5   with function Matches (E    : T_Access;
6                         Elem : T_Cmp)
7                         return Boolean;
8 package Generic_Containers is
9
10  type Ref_Type (Ref : access T) is
11    tagged limited null record
12    with Implicit_Dereference => Ref;
13
14  type Container is
15    array (Positive range <>) of
16      T_Access;
17
18  function Find (Cont : Container;
19               Elem : T_Cmp)
20                return Ref_Type;
21
22 end Generic_Containers;
```

Listing 71: generic_containers.adb

```
1 package body Generic_Containers is
2
3   function Find (Cont : Container;
4                 Elem : T_Cmp)
5                 return Ref_Type is
6 begin
7   for I in Cont'Range loop
8     if Matches (Cont (I), Elem) then
9       return Ref_Type'(Ref => Cont (I));
10    end if;
11  end loop;
12
13  return Ref_Type'(Ref => null);
```

(continues on next page)

(continued from previous page)

```

14   end Find;
15
16 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 72: show_national_dates.adb

```

1  with Generic_Containers;
2  with National_Date_Info; use National_Date_Info;
3
4  procedure Show_National_Dates is
5
6      function Matches_Country
7          (E      : National_Date_Access;
8           Elem   : Country_Code)
9          return Boolean is
10             (E.Country = Elem);
11
12     package National_Date_Containers is new
13         Generic_Containers
14         (T           => National_Date,
15          T_Access   => National_Date_Access,
16          T_Cmp      => Country_Code,
17          Matches    => Matches_Country);
18
19     use National_Date_Containers;
20
21     subtype National_Dates is Container;
22
23     Nat_Dates : constant
24         National_Dates (1 .. 5) :=
25         (new National_Date'("USA",
26                             Time'(1776, 7, 4)),
27          new National_Date'("FRA",
28                             Time'(1789, 7, 14)),
29          new National_Date'("DEU",
30                             Time'(1990, 10, 3)),
31          new National_Date'("SPA",
32                             Time'(1492, 10, 12)),
33          new National_Date'("BRA",
34                             Time'(1822, 9, 7)));
35
36     begin
37         Find (Nat_Dates, "FRA").Show;
38     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)
```

(continues on next page)

(continued from previous page)

```

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](#)²³⁰
-

15.8 Anonymous Access Types and Accessibility Rules

In general, the *accessibility rules* (page 536) we've seen earlier also apply to anonymous access types. However, there are some subtle differences, which we discuss in this section.

Let's adapt the *code example from that section* (page 536) to make use of anonymous access types:

Listing 73: library_level.ads

```

1 package Library_Level is
2
3   L0_A0 : access Integer;
4
5   L0_Var : aliased Integer;
6
7 end Library_Level;

```

²³⁰ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html>

Listing 74: show_library_level.adb

```

1  with Library_Level; use Library_Level;
2
3  procedure Show_Library_Level is
4      L1_Var : aliased Integer;
5
6      L1_A0  : access Integer;
7
8      procedure Test is
9          L2_A0  : access Integer;
10
11         L2_Var : aliased Integer;
12         begin
13             L1_A0 := L2_Var'Access;
14             --      ^^^^^
15             --      ILLEGAL: L2 object to
16             --                L1 access object
17
18             L2_A0 := L2_Var'Access;
19             --      ^^^^^
20             --      LEGAL: L2 object to
21             --                L2 access object
22         end Test;
23
24     begin
25         L0_A0 := new Integer'(22);
26         --      ^^^^^^^^^
27         --      LEGAL: L0 object to
28         --                L0 access object
29
30         L0_A0 := L1_Var'Access;
31         --      ^^^^^
32         --      ILLEGAL: L1 object to
33         --                L0 access object
34
35         L1_A0 := L0_Var'Access;
36         --      ^^^^^
37         --      LEGAL: L0 object to
38         --                L1 access object
39
40         L1_A0 := L1_Var'Access;
41         --      ^^^^^
42         --      LEGAL: L1 object to
43         --                L1 access object
44
45         L0_A0 := L1_A0; -- legal!!
46         --      ^^^^^
47         --      LEGAL: L1 access object to
48         --                L0 access object
49
50         --      ILLEGAL: L1 object
51         --                (L1_A0 = L1_Var'Access)
52         --                to
53         --                L0 access object
54
55         --      This is actually OK at compile time,
56         --      but the accessibility check fails at
57         --      runtime.
58
59         Test;
60     end Show_Library_Level;

```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↳Accessibility_Levels_Rules_Introduction.Accessibility_Library_Level
MD5: 255bdecebd735408db082edd583a0c
```

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_A0 := L1_Var'Access` is illegal because we're trying to assign to an access object of less deep level.

However, assignment such as `L0_A0 := L1_A0` are possible now: we don't get a type mismatch — as we did with named access types — because both objects are of anonymous access types. Note that the accessibility level cannot be determined at compile time: `L1_A0` can hold an access value at library level (which would make the assignment legal) or at a deeper level. Therefore, the compiler introduces an accessibility check here.

However, the accessibility check used in `L0_A0 := L1_A0` fails at runtime because the corresponding access value (`L1_Var'Access`) is of a deeper level than `L0_A0`, which is illegal. (If you comment out the `L1_A0 := L1_Var'Access` assignment prior to the `L0_A0 := L1_A0` assignment, this accessibility check doesn't fail anymore.)

15.8.1 Conversions between Anonymous and Named Access Types

In the previous sections, we've discussed accessibility rules for named and anonymous access types separately. In this section, we see that the same accessibility rules apply when mixing both flavors together and converting objects of anonymous to named access types.

Let's adapt parts of the previous *code example* (page 536) and add anonymous access types to it:

Listing 75: library_level.ads

```
1 package Library_Level is
2
3     type L0_Integer_Access is
4         access all Integer;
5
6     L0_Var : aliased Integer;
7
8     L0_IA  : L0_Integer_Access;
9     L0_A0 : access Integer;
10
11 end Library_Level;
```

Listing 76: show_library_level.adb

```
1 with Library_Level; use Library_Level;
2
3 procedure Show_Library_Level is
4     type L1_Integer_Access is
5         access all Integer;
6
```

(continues on next page)

(continued from previous page)

```

7  L1_IA : L1_Integer_Access;
8  L1_AO : access Integer;
9
10 L1_Var : aliased Integer;
11
12 begin
13  -----
14  -- From named type to anonymous type
15  -----
16
17  L0_IA := new Integer'(22);
18  L1_IA := new Integer'(42);
19
20  L0_AO := L0_IA;
21  --      ^^^^^
22  --      LEGAL: assignment from
23  --              L0 access object (named type)
24  --              to
25  --              L0 access object
26  --              (anonymous type)
27
28  L0_AO := L1_IA;
29  --      ^^^^^
30  --      ILLEGAL: assignment from
31  --              L1 access object (named type)
32  --              to
33  --              L0 access object
34  --              (anonymous type)
35
36  L1_AO := L0_IA;
37  --      ^^^^^
38  --      LEGAL: assignment from
39  --              L0 access object (named type)
40  --              to
41  --              L1 access object
42  --              (anonymous type)
43
44  L1_AO := L1_IA;
45  --      ^^^^^
46  --      LEGAL: assignment from
47  --              L1 access object (named type)
48  --              to
49  --              L1 access object
50  --              (anonymous type)
51
52  -----
53  -- From anonymous type to named type
54  -----
55
56  L0_AO := L0_Var'Access;
57  L1_AO := L1_Var'Access;
58
59  L0_IA := L0_Integer_Access (L0_AO);
60  --      ^^^^^^^^^^^^^^^^^^^^^
61  --      LEGAL: conversion / assignment from
62  --              L0 access object
63  --              (anonymous type)
64  --              to
65  --              L0 access object (named type)
66
67  L0_IA := L0_Integer_Access (L1_AO);

```

(continues on next page)

(continued from previous page)

```

68  --      ~~~~~
69  --      ILLEGAL: conversion / assignment from
70  --              L1 access object
71  --              (anonymous type)
72  --              to
73  --              L0 access object (named type)
74  --              (accessibility check fails)
75
76  L1_IA := L1_Integer_Access (L0_A0);
77  --      ~~~~~
78  --      LEGAL: conversion / assignment from
79  --              L0 access object
80  --              (anonymous type)
81  --              to
82  --              L1 access object (named type)
83
84  L1_IA := L1_Integer_Access (L1_A0);
85  --      ~~~~~
86  --      LEGAL: conversion / assignment from
87  --              L1 access object
88  --              (anonymous type)
89  --              to
90  --              L1 access object (named type)
91  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_A0 := L1_IA` and the `L0_IA := L0_Integer_Access (L1_A0)` assignments. Otherwise, mixing those access objects doesn't impose additional hurdles.

15.8.2 Accessibility rules on access parameters

In the previous chapter, we saw that the accessibility rules also apply to *access values as subprogram parameters* (page 540). In the case of access parameters, the rules are a bit less strict (as you may generally expect for anonymous access types), and the accessibility rules are checked at runtime. This allows use to use access values that would be illegal in the case of named access types because of their accessibility levels.

Let's adapt a previous code example to make use of access parameters:

Listing 77: names.ads

```

1  package Names is
2
3     procedure Show (N : access constant String);
```

(continues on next page)

(continued from previous page)

```
4
5 end Names;
```

Listing 78: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Show (N : access constant String) is
9     begin
10         -- for I in N'Range loop
11         --     N (I) := To_Lower (N (I));
12         -- end loop;
13         Put_Line ("Name: " & N.all);
14     end Show;
15
16 end Names;
```

Listing 79: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4     S : aliased String := "John";
5 begin
6     Show (S'Access);
7 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.

15.9 Anonymous Access-To-Subprograms

In the previous chapter, we talked about *named access-to-subprogram types* (page 566). Now, we'll see that the anonymous version of those types isn't much different from the named version.

Let's start our discussion by declaring a subprogram parameter using an anonymous access-to-procedure type:

Listing 80: anonymous_access_to_subprogram.ads

```

1 package Anonymous_Access_To_Subprogram is
2
3   procedure Proc
4     (P : access procedure (I : in out Integer));
5
6 end Anonymous_Access_To_Subprogram;
```

Listing 81: anonymous_access_to_subprogram.adb

```

1 package body Anonymous_Access_To_Subprogram is
2
3   procedure Proc
4     (P : access procedure (I : in out Integer))
5   is
6     I : Integer := 0;
7   begin
8     P (I);
9   end Proc;
10
11 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 82: add_ten.ads

```

1 procedure Add_Ten (I : in out Integer);
```

Listing 83: add_ten.adb

```

1 procedure Add_Ten (I : in out Integer) is
2 begin
3   I := I + 10;
4 end Add_Ten;
```

Listing 84: add_twenty.ads

```

1 procedure Add_Twenty (I : in out Integer);
```

Listing 85: add_twenty.adb

```

1 procedure Add_Twenty (I : in out Integer) is
2 begin
3   I := I + 20;
4 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 86: show_anonymous_access_to_subprograms.adb

```
1 with Anonymous_Access_To_Subprogram;  
2 use Anonymous_Access_To_Subprogram;  
3  
4 with Add_Ten;  
5  
6 procedure Show_Anonymous_Access_To_Subprograms is  
7 begin  
8   Proc (Add_Ten'Access);  
9   --           ^ Getting access to Add_Ten  
10  --           procedure and passing it  
11  --           to Proc  
12 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 568). In fact, in terms of usage, anonymous access-to-subprogram types are very similar to named access-to-subprogram types. The major differences can be found in the corresponding *accessibility rules* (page 658).

In the Ada Reference Manual

- 3.10 Access Types²³¹
-

15.9.1 Examples of anonymous access-to-subprogram usage

In the section about *named access-to-subprogram types* (page 566), we've seen a couple of different usages for those types. In all those examples we discussed, we could instead have used anonymous access-to-subprogram types. Let's see a code example that illustrates that:

Listing 87: all_anonymous_access_to_subprogram.ads

```
1 package All_Anonymous_Access_To_Subprogram is  
2  
3   --  
4   -- Anonymous access-to-subprogram as  
5   -- subprogram parameter:  
6   --  
7   procedure Proc  
8     (P : access procedure (I : in out Integer));  
9  
10  --
```

(continues on next page)

²³¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

11  -- Anonymous access-to-subprogram in
12  -- array type declaration:
13  --
14  type Access_To_Procedure_Array is
15  array (Positive range <>) of
16  access procedure (I : in out Integer);
17
18  protected type Protected_Integer is
19
20  procedure Mult_Ten;
21
22  procedure Mult_Twenty;
23
24  private
25  I : Integer := 1;
26  end Protected_Integer;
27
28  --
29  -- Anonymous access-to-subprogram as
30  -- component of a record type.
31  --
32  type Rec_Access_To_Procedure is record
33  AP : access procedure (I : in out Integer);
34  end record;
35
36  --
37  -- Anonymous access-to-subprogram as
38  -- discriminant:
39  --
40  type Rec_Access_To_Procedure_Discriminant
41  (AP : access procedure
42  (I : in out Integer)) is
43  record
44  I : Integer := 0;
45  end record;
46
47  procedure Process
48  (R : in out
49  Rec_Access_To_Procedure_Discriminant);
50
51  generic
52  type T is private;
53
54  --
55  -- Anonymous access-to-subprogram as
56  -- formal parameter:
57  --
58  Proc_T : access procedure
59  (Element : in out T);
60  procedure Gen_Process (Element : in out T);
61
62  end All_Anonymous_Access_To_Subprogram;

```

Listing 88: all_anonymous_access_to_subprogram.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body All_Anonymous_Access_To_Subprogram is
4
5  procedure Proc
6  (P : access procedure (I : in out Integer))

```

(continues on next page)

(continued from previous page)

```

7   is
8     I : Integer := 0;
9   begin
10    Put_Line
11      ("Calling procedure for Proc...");
12    P (I);
13    Put_Line ("Finished.");
14  end Proc;
15
16  procedure Process
17    (R : in out
18      Rec_Access_To_Procedure_Discriminant)
19  is
20  begin
21    Put_Line
22      ("Calling procedure for"
23      & " Rec_Access_To_Procedure_Discriminant"
24      & " type...");
25    R.AP (R.I);
26    Put_Line ("Finished.");
27  end Process;
28
29  procedure Gen_Process (Element : in out T) is
30  begin
31    Put_Line
32      ("Calling procedure for Gen_Process...");
33    Proc_T (Element);
34    Put_Line ("Finished.");
35  end Gen_Process;
36
37  protected body Protected_Integer is
38
39    procedure Mult_Ten is
40    begin
41      I := I * 10;
42    end Mult_Ten;
43
44    procedure Mult_Twenty is
45    begin
46      I := I * 20;
47    end Mult_Twenty;
48
49  end Protected_Integer;
50
51 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 89: show_anonymous_access_to_subprograms.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Add_Ten;
4  with Add_Twenty;
5
6  with All_Anonymous_Access_To_Subprogram;
7  use All_Anonymous_Access_To_Subprogram;
8
9  procedure Show_Anonymous_Access_To_Subprograms is
10     --
11     -- Anonymous access-to-subprogram as
12     -- an object:
13     --
14     P : access procedure (I : in out Integer);
15
16     --
17     -- Array of anonymous access-to-subprogram
18     -- components
19     --
20     PA : constant
21         Access_To_Procedure_Array (1 .. 2) :=
22         (Add_Ten'Access,
23          Add_Twenty'Access);
24
25     --
26     -- Anonymous array of anonymous
27     -- access-to-subprogram components:
28     --
29     PAA : constant
30         array (1 .. 2) of access
31         procedure (I : in out Integer) :=
32         (Add_Ten'Access,
33          Add_Twenty'Access);
34
35     --
36     -- Record with anonymous
37     -- access-to-subprogram components:
38     --
39     RA : constant Rec_Access_To_Procedure :=
40         (AP => Add_Ten'Access);
41
42     --
43     -- Record with anonymous
44     -- access-to-subprogram discriminant:
45     --
46     RD : Rec_Access_To_Procedure_Discriminant
47         (AP => Add_Twenty'Access) :=
48         (AP => Add_Twenty'Access, I => 0);
49
50     --
51     -- Generic procedure with formal anonymous
52     -- access-to-subprogram:
53     --
54     procedure Process_Integer is new
55         Gen_Process (T => Integer,
56                    Proc_T => Add_Twenty'Access);
57
58     --
59     -- Object (APP) of anonymous

```

(continues on next page)

(continued from previous page)

```

60  -- access-to-protected-subprogram:
61  --
62  PI : Protected_Integer;
63  APP : constant access protected procedure :=
64        PI.Mult_Ten'Access;
65
66  Some_Int : Integer := 0;
67  begin
68    Put_Line ("Some_Int: " & Some_Int'Image);
69
70    --
71    -- Using object of
72    -- anonymous access-to-subprogram type:
73    --
74    P := Add_Ten'Access;
75    Proc (P);
76    P (Some_Int);
77
78    P := Add_Twenty'Access;
79    Proc (P);
80    P (Some_Int);
81
82    Put_Line ("Some_Int: " & Some_Int'Image);
83
84    --
85    -- Using array with component of
86    -- anonymous access-to-subprogram type:
87    --
88    Put_Line
89      ("Calling procedure from PA array...");
90
91    for I in PA'Range loop
92      PA (I) (Some_Int);
93      Put_Line ("Some_Int: " & Some_Int'Image);
94    end loop;
95
96    Put_Line ("Finished.");
97
98    Put_Line
99      ("Calling procedure from PAA array...");
100
101    for I in PAA'Range loop
102      PAA (I) (Some_Int);
103      Put_Line ("Some_Int: " & Some_Int'Image);
104    end loop;
105
106    Put_Line ("Finished.");
107
108    Put_Line ("Some_Int: " & Some_Int'Image);
109
110    --
111    -- Using record with component of
112    -- anonymous access-to-subprogram type:
113    --
114    RA.AP (Some_Int);
115    Put_Line ("Some_Int: " & Some_Int'Image);
116
117    --
118    -- Using record with discriminant of
119    -- anonymous access-to-subprogram type:
120    --

```

(continues on next page)

(continued from previous page)

```

121 Process (RD);
122 Put_Line ("RD.I: " & RD.I'Image);
123
124 --
125 -- Using procedure instantiated with
126 -- formal anonymous access-to-subprogram:
127 --
128 Process_Integer (Some_Int);
129 Put_Line ("Some_Int: " & Some_Int'Image);
130
131 --
132 -- Using object of anonymous
133 -- access-to-protected-subprogram type:
134 --
135 APP.all;
136 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 566), we won't repeat this discussion here. If anything in this

code example is still unclear to you, make sure to revisit that section from the previous chapter.

15.9.2 Application of anonymous access-to-subprogram types

In general, there isn't much that speaks against using anonymous access-to-subprogram types. We can say, for example, that they're much more useful than *anonymous access-to-objects types* (page 605), which have *many drawbacks* (page 608) — as we discussed earlier.

There isn't much to be concerned when using anonymous access-to-subprogram types. For example, we cannot allocate or deallocate a subprogram. As a consequence, we won't have storage management issues affecting these types because the access to those subprograms will always be available and no memory leak can occur.

Also, anonymous access-to-subprogram types can be easier to use than named access-to-subprogram types because of their less strict *accessibility rules* (page 658). Some of the accessibility issues we might encounter when using named access-to-subprogram types can be solved by declaring them as anonymous types. (We discuss the accessibility rules of anonymous access-to-subprogram types in the next section.)

15.9.3 Readability

Note that readability suffers if you use a *cascade* of anonymous access-to-subprograms. For example:

Listing 90: readability_issue.ads

```
1 package Readability_Issue is
2
3     function F
4         return access
5             function (A : Integer)
6                 return access
7                     function (B : Float)
8                         return Integer;
9
10 end Readability_Issue;
```

Listing 91: readability_issue-functions.ads

```
1 package Readability_Issue.Functions is
2
3     function To_Integer (V : Float)
4         return Integer is
5         (Integer (V));
6
7     function Select_Conversion
8         (A : Integer)
9         return access
10            function (B : Float)
11                return Integer is
12            (To_Integer'Access);
13
14 end Readability_Issue.Functions;
```

Listing 92: readability_issue.adb

```

1 with Readability_Issue.Functions;
2 use Readability_Issue.Functions;
3
4 package body Readability_Issue is
5
6     function F
7         return access
8         function (A : Integer)
9             return access
10            function (B : Float)
11                return Integer is
12            (Select_Conversion'Access);
13
14 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 93: show_readability_issue.adb

```

1 with Readability_Issue;
2 use Readability_Issue;
3
4 procedure Show_Readability_Issue is
5     F1 : access
6         function (A : Integer)
7             return access
8             function (B : Float)
9                 return Integer
10            := F;
11     F2 : access function (B : Float)
12         return Integer
13         := F1 (2);
14     I : Integer := F2 (0.1);
15 begin
16     I := F1 (2) (0.1);
17 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.

15.10 Accessibility Rules and Anonymous Access-To-Subprograms

In principle, the *accessibility rules for anonymous access types* (page 643) that we've seen before apply to anonymous access-to-subprograms as well. Also, we had a discussion about *accessibility rules and access-to-subprograms* (page 591) in the previous chapter. In this section, we review some of the rules that we already know and discuss how they relate to anonymous access-to-subprograms.

In the Ada Reference Manual

- 3.10 Access Types²³²
-

15.10.1 Named vs. anonymous access-to-subprograms

Let's see an example of a named access-to-subprogram type:

Listing 94: show_access_to_subprogram_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Access_To_Subprogram_Error is
4
5     type PI is access
6         procedure (I : in out Integer);
7
8     P : PI;
9
10    I : Integer := 0;
11 begin
12     declare
13         procedure Add_One (I : in out Integer) is
14             begin
15                 I := I + 1;
16             end Add_One;
17         begin
18             P := Add_One'Access;
19         end;
20 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`.

²³² <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

In contrast, using an anonymous access-to-subprogram type eliminates the compilation error, i.e. the assignment `P := Add_One'Access` becomes legal:

Listing 95: show_access_to_subprogram_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Access_To_Subprogram_Error is
4   P : access procedure (I : in out Integer);
5
6   I : Integer := 0;
7 begin
8   declare
9     procedure Add_One (I : in out Integer) is
10      begin
11        I := I + 1;
12      end Add_One;
13    begin
14      P := Add_One'Access;
15      -- RUNTIME ERROR: Add_One is out-of-scope
16      -- after this line.
17    end;
18 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`.

15.10.2 Named vs. anonymous access-to-subprograms as parameters

Using anonymous access-to-subprograms as parameters allows us to pass subprograms at any level. For certain applications, the restrictions that are applied to named access types might be too strict, so using anonymous access-to-subprograms might be a good way to circumvent those restrictions. They also allow the component developer to be independent of the clients' specific access types.

Note that the increased flexibility for anonymous access-to-subprograms means that some of the checks that are performed at compile time for named access-to-subprograms are done at runtime for anonymous access-to-subprograms.

Named access-to-subprograms as a parameter

Let's see an example using a named access-to-procedure type:

Listing 96: access_to_subprogram_types.ads

```
1 package Access_To_Subprogram_Types is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     type Process_Procedure is
7         access
8             procedure (Arr : in out Integer_Array);
9
10    procedure Process
11        (Arr : in out Integer_Array;
12         P   : Process_Procedure);
13
14 end Access_To_Subprogram_Types;
```

Listing 97: access_to_subprogram_types.adb

```
1 package body Access_To_Subprogram_Types is
2
3     procedure Process
4         (Arr : in out Integer_Array;
5          P   : Process_Procedure) is
6     begin
7         P (Arr);
8     end Process;
9
10 end Access_To_Subprogram_Types;
```

Listing 98: show_access_to_subprogram_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7
8     procedure Add_One
9         (Arr : in out Integer_Array) is
10    begin
11        for E of Arr loop
12            E := E + 1;
13        end loop;
14    end Add_One;
15
16    procedure Display
17        (Arr : in out Integer_Array) is
18    begin
19        for I in Arr'Range loop
20            Put_Line ("Arr (" &
21                    Integer'Image (I)
22                    & "): "
23                    & Integer'Image (Arr (I)));
24        end loop;
25    end Display;
```

(continues on next page)

(continued from previous page)

```

26
27   Arr : Integer_Array (1 .. 3) := (1, 2, 3);
28 begin
29   Process (Arr, Display'Access);
30
31   Put_Line ("Add_One...");
32   Process (Arr, Add_One'Access);
33
34   Process (Arr, Display'Access);
35 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 99: `access_to_subprogram_types.ads`

```

1 package Access_To_Subprogram_Types is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   procedure Process
7     (Arr : in out Integer_Array;
8      P   : access procedure
9         (Arr : in out Integer_Array));
10
11 end Access_To_Subprogram_Types;
```


Listing 100: access_to_subprogram_types.adb

```
1 package body Access_To_Subprogram_Types is
2
3   procedure Process
4     (Arr : in out Integer_Array;
5      P   : access procedure
6         (Arr : in out Integer_Array)) is
7   begin
8     P (Arr);
9   end Process;
10
11 end Access_To_Subprogram_Types;
```

Listing 101: show_access_to_subprogram_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7
8   procedure Add_One
9     (Arr : in out Integer_Array) is
10  begin
11    for E of Arr loop
12      E := E + 1;
13    end loop;
14  end Add_One;
15
16  procedure Display
17    (Arr : in out Integer_Array) is
18  begin
19    for I in Arr'Range loop
20      Put_Line ("Arr (" &
21               Integer'Image (I)
22               & "): "
23               & Integer'Image (Arr (I)));
24    end loop;
25  end Display;
26
27  Arr : Integer_Array (1 .. 3) := (1, 2, 3);
28  begin
29    Process (Arr, Display'Access);
30
31    Put_Line ("Add_One...");
32    Process (Arr, Add_One'Access);
33
34    Process (Arr, Display'Access);
35  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 run-time exception because the subprograms are still *accessible* when we call `Process`.

15.10.3 Iterator

A typical example that illustrates well the necessity of using anonymous access-to-subprograms is that of a container iterator. In fact, many of the standard Ada containers — the child packages of `Ada.Containers` — make use of anonymous access-to-subprograms for their `Iterate` subprograms.

In the Ada Reference Manual

- [A.18.2 The Package Containers.Vectors](#)²³³
 - [A.18.4 Maps](#)²³⁴
 - [A.18.7 Sets](#)²³⁵
-

Using named access-to-subprograms

Let's start with a simplified container type (`Data_Container`) using a named access-to-subprogram type (`Process_Element`) for iteration:

Listing 102: `data_processing.ads`

```

1  generic
2    type Element is private;
3  package Data_Processing is
4
5    type Data_Container (Last : Positive) is
6      private;
7
8    Data_Container_Full : exception;
9
10   procedure Append (D : in out Data_Container;
11                   E :      Element);
12
13   type Process_Element is
14     not null access procedure (E : Element);
15
16   procedure Iterate
17     (D      : Data_Container;
18      Proc  : Process_Element);
19
20 private

```

(continues on next page)

²³³ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html>

²³⁴ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-4.html>

²³⁵ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-7.html>

(continued from previous page)

```

21
22 type Data_Container_Storage is
23     array (Positive range <>) of Element;
24
25 type Data_Container (Last : Positive) is
26     record
27         S      : Data_Container_Storage (1 .. Last);
28         Curr : Natural := 0;
29     end record;
30
31 end Data_Processing;

```

Listing 103: data_processing.adb

```

1 package body Data_Processing is
2
3     procedure Append (D : in out Data_Container;
4                     E :      Element) is
5     begin
6         if D.Curr < D.S'Last then
7             D.Curr := D.Curr + 1;
8             D.S (D.Curr) := E;
9         else
10            raise Data_Container_Full;
11            -- NOTE: This is just a dummy
12            --       implementation. A better
13            --       strategy is to add actual error
14            --       handling when the container is
15            --       full.
16        end if;
17    end Append;
18
19    procedure Iterate
20        (D      : Data_Container;
21         Proc : Process_Element) is
22    begin
23        for I in D.S'First .. D.Curr loop
24            Proc (D.S (I));
25        end loop;
26    end Iterate;
27
28 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 104: float_data_processing.ads

```

1 with Data_Processing;
2
3 package Float_Data_Processing is
4     new Data_Processing (Element => Float);

```

Listing 105: show_access_to_subprograms.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Float_Data_Processing;
4 use Float_Data_Processing;
5
6 procedure Show_Access_To_Subprograms is
7
8     procedure Display (F : Float) is
9     begin
10         Put_Line ("F :" & Float'Image (F));
11     end Display;
12
13     D : Data_Container (5);
14 begin
15     Append (D, 1.0);
16     Append (D, 2.0);
17     Append (D, 3.0);
18
19     Iterate (D, Display'Access);
20 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 106: data_processing.ads

```

1 generic
2     type Element is private;
3 package Data_Processing is
4
5     type Data_Container (Last : Positive) is
6     private;
7
8     Data_Container_Full : exception;
9
10    procedure Append (D : in out Data_Container;
11                    E : Element);
12
13    procedure Iterate
14    (D : Data_Container;
15     Proc : not null access
16     procedure (E : Element));
```

(continues on next page)

(continued from previous page)

```

17
18 private
19
20     type Data_Container_Storage is
21         array (Positive range <>) of Element;
22
23     type Data_Container (Last : Positive) is
24         record
25             S      : Data_Container_Storage (1 .. Last);
26             Curr : Natural := 0;
27         end record;
28
29 end Data_Processing;
```

Listing 107: data_processing.adb

```

1 package body Data_Processing is
2
3     procedure Append (D : in out Data_Container;
4                     E :      Element) is
5     begin
6         if D.Curr < D.S'Last then
7             D.Curr := D.Curr + 1;
8             D.S (D.Curr) := E;
9         else
10            raise Data_Container_Full;
11            -- NOTE: This is just a dummy
12            --       implementation. A better
13            --       strategy is to add actual error
14            --       handling when the container is
15            --       full.
16        end if;
17    end Append;
18
19    procedure Iterate
20        (D      : Data_Container;
21         Proc : not null access
22             procedure (E : Element)) is
23    begin
24        for I in D.S'First .. D.Curr loop
25            Proc (D.S (I));
26        end loop;
27    end Iterate;
28
29 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 108: float_data_processing.ads

```
1 with Data_Processing;  
2  
3 package Float_Data_Processing is  
4   new Data_Processing (Element => Float);
```

Listing 109: show_access_to_subprograms.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 with Float_Data_Processing;  
4 use Float_Data_Processing;  
5  
6 procedure Show_Access_To_Subprograms is  
7  
8   procedure Display (F : Float) is  
9     begin  
10      Put_Line ("F :" & Float'Image (F));  
11     end Display;  
12  
13   D : Data_Container (5);  
14 begin  
15   Append (D, 1.0);  
16   Append (D, 2.0);  
17   Append (D, 3.0);  
18  
19   Iterate (D, Display'Access);  
20 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`.

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.

16.1 Assignment and equality

Limited types have the following restrictions, which we discussed in the [Introduction to Ada²³⁶](#) 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

```
1 package Nonlimited_Types is
2
3   type Simple_Rec is private;
4
5   type Integer_Access is access Integer;
6
7   function Init (I : Integer) return Simple_Rec;
8
9   procedure Set (E : Simple_Rec;
10                I : Integer);
11
12  procedure Show (E       : Simple_Rec;
13                 E_Name : String);
14
15 private
16
17  type Simple_Rec is record
```

(continues on next page)

²³⁶ <https://learn.adacore.com/courses/intro-to-ada/chapters/privacy.html#intro-ada-limited-types>

(continued from previous page)

```
18     V : Integer_Access;
19     end record;
20
21 end Nonlimited_Types;
```

Listing 2: nonlimited_types.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Nonlimited_Types is
4
5     function Init (I : Integer) return Simple_Rec
6     is
7     begin
8         return E : Simple_Rec do
9             E.V := new Integer'(I);
10        end return;
11    end Init;
12
13    procedure Set (E : Simple_Rec;
14                 I : Integer) is
15    begin
16        E.V.all := I;
17    end Set;
18
19    procedure Show (E      : Simple_Rec;
20                  E_Name : String) is
21    begin
22        Put_Line (E_Name
23                & ".V.all = "
24                & Integer'Image (E.V.all));
25    end Show;
26
27 end Nonlimited_Types;
```

Listing 3: show_wrong_assignment_equality.adb

```
1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Nonlimited_Types; use Nonlimited_Types;
3
4  procedure Show_Wrong_Assignment_Equality is
5      A, B : Simple_Rec := Init (0);
6
7      procedure Show_Compare is
8      begin
9          if A = B then
10             Put_Line ("A = B");
11          else
12             Put_Line ("A /= B");
13          end if;
14      end Show_Compare;
15  begin
16
17      Put_Line ("A := Init (0); A := Init (0);");
18      Show (A, "A");
19      Show (B, "B");
20      Show_Compare;
21      Put_Line ("-----");
22
23      Put_Line ("Set (A, 2); Set (B, 3);");
```

(continues on next page)

(continued from previous page)

```

24   Set (A, 2);
25   Set (B, 3);
26
27   Show (A, "A");
28   Show (B, "B");
29   Put_Line ("-----");
30
31   Put_Line ("B := A");
32   B := A;
33
34   Show (A, "A");
35   Show (B, "B");
36   Show_Compare;
37   Put_Line ("-----");
38
39   Put_Line ("Set (B, 7);");
40   Set (B, 7);
41
42   Show (A, "A");
43   Show (B, "B");
44   Show_Compare;
45   Put_Line ("-----");
46
47 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](#)²³⁷
-

16.1.1 Assignments

Assignments are forbidden when using objects of limited types. For example:

Listing 4: `limited_types.ads`

```
1 package Limited_Types is
2
3     type Simple_Rec is limited private;
4
5     type Integer_Access is access Integer;
6
7     function Init (I : Integer) return Simple_Rec;
8
9 private
10
11     type Simple_Rec is limited record
12         V : Integer_Access;
13     end record;
14
15 end Limited_Types;
```

Listing 5: `limited_types.adb`

```
1 package body Limited_Types is
2
3     function Init (I : Integer) return Simple_Rec
```

(continues on next page)

²³⁷ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

(continued from previous page)

```

4   is
5   begin
6       return E : Simple_Rec do
7           E.V := new Integer'(I);
8       end return;
9   end Init;
10
11 end Limited_Types;

```

Listing 6: show_limited_assignment.adb

```

1 with Limited_Types; use Limited_Types;
2
3 procedure Show_Limited_Assignment is
4     A, B : Simple_Rec := Init (0);
5 begin
6     B := A;
7 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
↳ limited type
gprbuild: *** compilation phase failed

```

In this example, we declare the limited private type `Simple_Rec` and two objects of this type (A and B) in the `Show_Limited_Assignment` procedure. (We discuss more about limited private types *later* (page 677)).

As expected, we get a compilation error for the `B := A` statement (in the `Show_Limited_Assignment` procedure). If we need to copy two objects of limited type, we have to provide a custom procedure to do that. For example, we can implement a `Copy` procedure for the `Simple_Rec` type:

Listing 7: limited_types.ads

```

1 package Limited_Types is
2
3     type Integer_Access is access Integer;
4
5     type Simple_Rec is limited private;
6
7     function Init (I : Integer) return Simple_Rec;
8
9     procedure Copy (From : Simple_Rec;
10                  To : in out Simple_Rec);
11
12 private
13
14     type Simple_Rec is limited record
15         V : Integer_Access;
16     end record;
17
18 end Limited_Types;

```

Listing 8: limited_types.adb

```
1 package body Limited_Types is
2
3     function Init (I : Integer) return Simple_Rec
4     is
5     begin
6         return E : Simple_Rec do
7             E.V := new Integer'(I);
8         end return;
9     end Init;
10
11    procedure Copy (From : Simple_Rec;
12                  To   : in out Simple_Rec)
13    is
14    begin
15        -- Copying record components
16        To.V.all := From.V.all;
17    end Copy;
18
19 end Limited_Types;
```

Listing 9: show_limited_assignment.adb

```
1 with Limited_Types; use Limited_Types;
2
3 procedure Show_Limited_Assignment is
4     A, B : Simple_Rec := Init (0);
5 begin
6     Copy (From => A, To => B);
7 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 cases, so this design decision must be made carefully. In any case, using limited types ensures that only the assignment subprograms that are explicitly declared in the package specification are available.

16.1.2 Equality

Limited types don't have a predefined equality operator. For example:

Listing 10: limited_types.ads

```
1 package Limited_Types is
2
3     type Integer_Access is access Integer;
4
5     type Simple_Rec is limited private;
6
7     function Init (I : Integer) return Simple_Rec;
8
```

(continues on next page)

(continued from previous page)

```

9 private
10
11     type Simple_Rec is limited record
12         V : Integer_Access;
13     end record;
14
15 end Limited_Types;
```

Listing 11: limited_types.adb

```

1 package body Limited_Types is
2
3     function Init (I : Integer) return Simple_Rec
4     is
5     begin
6         return E : Simple_Rec do
7             E.V := new Integer'(I);
8         end return;
9     end Init;
10
11 end Limited_Types;
```

Listing 12: show_limited_equality.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Limited_Types; use Limited_Types;
3
4 procedure Show_Limited_Equality is
5     A : Simple_Rec := Init (5);
6     B : Simple_Rec := Init (6);
7 begin
8     if A = B then
9         Put_Line ("A = B");
10    else
11        Put_Line ("A /= B");
12    end if;
13 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

```

1 package Limited_Types is
2
3     type Integer_Access is access Integer;
```

(continues on next page)

(continued from previous page)

```
4
5  type Simple_Rec is limited private;
6
7  function Init (I : Integer) return Simple_Rec;
8
9  function "=" (Left, Right : Simple_Rec)
10             return Boolean;
11
12 private
13
14  type Simple_Rec is limited record
15     V : Integer_Access;
16  end record;
17
18 end Limited_Types;
```

Listing 14: limited_types.adb

```
1  package body Limited_Types is
2
3     function Init (I : Integer) return Simple_Rec
4     is
5     begin
6         return E : Simple_Rec do
7             E.V := new Integer'(I);
8         end return;
9     end Init;
10
11    function "=" (Left, Right : Simple_Rec)
12                return Boolean is
13    begin
14        -- Comparing record components
15        return Left.V.all = Right.V.all;
16    end "=";
17
18 end Limited_Types;
```

Listing 15: show_limited_equality.adb

```
1  with Ada.Text_IO;   use Ada.Text_IO;
2  with Limited_Types; use Limited_Types;
3
4  procedure Show_Limited_Equality is
5     A : Simple_Rec := Init (5);
6     B : Simple_Rec := Init (6);
7  begin
8     if A = B then
9         Put_Line ("A = B");
10    else
11        Put_Line ("A /= B");
12    end if;
13 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;
```

16.2 Limited private types

As we've seen in code examples from the previous section, we can apply *information hiding* (page 35) to limited types. In other words, we can declare a type as **limited private** instead of just **limited**. For example:

Listing 16: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is limited record
8         I : Integer;
9     end record;
10
11 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](#)²³⁸
-

16.2.1 Non-Record Limited Types

In principle, only record types can be declared limited, so we cannot use scalar or array types. For example, the following declarations won't compile:

Listing 17: `non_record_limited_error.ads`

```
1 package Non_Record_Limited_Error is
2
3     type Limited_Enumeration is
4         limited (Off, On);
5
6     type Limited_Integer is new
7         limited Integer;
8
9     type Integer_Array is
10        array (Positive range <>) of Integer;
11
12    type Rec is new
13        limited Integer_Array (1 .. 2);
14
15 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 38) that private types don't have to be record types necessarily. In this sense, limited private types makes it possible for us to use types other than record types in the full view and still benefit from the restrictions of limited types. For example:

Listing 18: `simple_recs.ads`

```
1 package Simple_Recs is
2
3     type Limited_Enumeration is
4         limited private;
5
6     type Limited_Integer is
```

(continues on next page)

²³⁸ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

(continued from previous page)

```

7     limited private;
8
9     type Limited_Integer_Array_2 is
10        limited private;
11
12 private
13
14     type Limited_Enumeration is (Off, On);
15
16     type Limited_Integer is new Integer;
17
18     type Integer_Array is
19        array (Positive range <>) of Integer;
20
21     type Limited_Integer_Array_2 is
22        new Integer_Array (1 .. 2);
23
24 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.

16.2.2 Partial and full view of limited types

In the previous example, both partial and full views of the `Rec` type were limited. We may actually declare a type as `limited private` (in the public part of a package), while its full view is nonlimited. For example:

Listing 19: `simple_recs.ads`

```

1 package Simple_Recs is
2
3     type Rec is limited private;
4     -- Partial view of Rec is limited
5
6 private
7
8     type Rec is record
9     -- Full view of Rec is nonlimited
10        I : Integer;
11    end record;
12
13 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

```
1 package Simple_Recs.Child
2 is
3   type Rec_Derived is new Rec;
4     -- As for its parent, the
5     -- partial view of Rec_Derived
6     -- is limited, but the full view
7     -- is nonlimited.
8
9 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

```
1 with Simple_Recs; use Simple_Recs;
2
3 procedure Use_Rec_In_Subprogram is
4   R1, R2 : Rec;
5 begin
6   R1.I := 1;
7   R2 := R1;
8 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

```
1 package Simple_Recs is
2
3   type Rec is private;
4
5 private
```

(continues on next page)

(continued from previous page)

```

6
7   type Rec is limited record
8     I : Integer;
9   end record;
10
11 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

```

1 package Simple_Recs is
2
3   type Rec is tagged limited private;
4
5 private
6
7   type Rec is tagged record
8     I : Integer;
9   end record;
10
11 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.ads:7:09: error: completion of limited tagged type must be limited
gprbuild: *** compilation phase failed

```

Here, compilation fails because the type Rec is nonlimited in its full view.

16.2.3 Limited and nonlimited in full view

Declaring the full view of a type as limited or nonlimited has implications in the way we can use objects of this type in the package body. For example:

Listing 24: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Rec_Limited_Full is limited private;
4     type Rec_Nonlimited_Full is limited private;
5
6     procedure Copy
7         (From :      Rec_Limited_Full;
8          To   : in out Rec_Limited_Full);
9     procedure Copy
10        (From :      Rec_Nonlimited_Full;
11         To   : in out Rec_Nonlimited_Full);
12
13 private
14
15     type Rec_Limited_Full is limited record
16         I : Integer;
17     end record;
18
19     type Rec_Nonlimited_Full is record
20         I : Integer;
21     end record;
22
23 end Simple_Recs;
```

Listing 25: simple_recs.adb

```
1 package body Simple_Recs is
2
3     procedure Copy
4         (From :      Rec_Limited_Full;
5          To   : in out Rec_Limited_Full)
6     is
7     begin
8         To := From;
9         -- ERROR: assignment is forbidden because
10        --      Rec_Limited_Full is limited in
11        --      its full view.
12     end Copy;
13
14     procedure Copy
15         (From :      Rec_Nonlimited_Full;
16          To   : in out Rec_Nonlimited_Full)
17     is
18     begin
19         To := From;
20         -- OK: assignment is allowed because
21         --      Rec_Nonlimited_Full is
22         --      nonlimited in its full view.
23     end Copy;
24
25 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.

16.2.4 Limited private component

Another example mentioned by the Ada Reference Manual (7.3.1²³⁹, 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`

```
1 package Limited_Nonlimited_Arrays is
2
3   type Limited_Private is
4     limited private;
5
6   function Init return Limited_Private;
7
8   -- The array type Limited_Private_Array
9   -- is limited because the type of its
10  -- component is limited.
11  type Limited_Private_Array is
12    array (Positive range <>) of
13      Limited_Private;
14
15 private
16
17   type Limited_Private is
18     record
19     A : Integer;
20   end record;
21
22   -- Limited_Private_Array type is
23   -- nonlimited at this point because
24   -- its component is nonlimited.
25   --
26   -- The assignments below are OK:
27   A1 : Limited_Private_Array (1 .. 5);
28
29   A2 : Limited_Private_Array := A1;
30
31 end Limited_Nonlimited_Arrays;
```

²³⁹ <http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html>

Listing 27: limited_nonlimited_arrays.adb

```
1 package body Limited_Nonlimited_Arrays is
2
3     function Init return Limited_Private is
4         ((A => 1));
5
6 end Limited_Nonlimited_Arrays;
```

Listing 28: show_limited_nonlimited_array.adb

```
1 with Limited_Nonlimited_Arrays;
2 use Limited_Nonlimited_Arrays;
3
4 procedure Show_Limited_Nonlimited_Array is
5     A3 : Limited_Private_Array (1 .. 2) :=
6         (others => Init);
7     A4 : Limited_Private_Array (1 .. 2);
8 begin
9     -- ERROR: this assignment is illegal because
10    -- Limited_Private_Array is limited, as
11    -- its component is limited at this point.
12    A4 := A3;
13 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 Operations²⁴⁰

²⁴⁰ <http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html>

16.2.5 Tagged limited private types

For tagged private types, the partial and full views must match: if a tagged type is limited in the partial view, it must be limited in the full view. For example:

Listing 29: simple_recs.ads

```

1 package Simple_Recs is
2
3     type Rec is tagged limited private;
4
5 private
6
7     type Rec is tagged limited record
8         I : Integer;
9     end record;
10
11 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.

16.3 Explicitly limited types

Under certain conditions, limited types can be called explicitly limited — note that using the **limited** keyword in a part of the declaration doesn't necessary ensure this, as we'll see later.

Let's start with an example of an explicitly limited type:

Listing 30: simple_recs.ads

```

1 package Simple_Recs is
2
3     type Rec is limited record
4         I : Integer;
5     end record;
6
7 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`

```
1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is limited record
8         I : Integer;
9     end record;
10
11 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 679), we may actually declare a type as **limited private** in the public part of a package, while its full view is nonlimited. In this case, the limited type is not considered explicitly limited anymore.

For example, if we make the full view of the `Rec` nonlimited (by removing the **limited** keyword in the private part), then the `Rec` type isn't explicitly limited anymore:

Listing 32: `simple_recs.ads`

```
1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is record
8         I : Integer;
9     end record;
10
11 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 685) 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 Modes²⁴¹
- 6.4.1 Parameter Associations²⁴²
- 7.5 Limited Types²⁴³

16.4 Subtypes of Limited Types

We can declare subtypes of limited types. For example:

Listing 33: simple_rec.s.ads

```

1 package Simple_Recs is
2
3   type Limited_Integer_Array (L : Positive) is
4     limited private;
5
6   subtype Limited_Integer_Array_2 is
7     Limited_Integer_Array (2);
8
9 private
10
11  type Integer_Array is
12    array (Positive range <>) of Integer;
13
14  type Limited_Integer_Array (L : Positive) is
15    limited record
16      Arr : Integer_Array (1 .. L);
17    end record;
18
19 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

```

1 with Simple_Recs; use Simple_Recs;
2
3 procedure Test_Limitedness is
4   Dummy_1, Dummy_2 : Limited_Integer_Array_2;
5 begin
6   Dummy_2 := Dummy_1;
7 end Test_Limitedness;
```

²⁴¹ <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

²⁴² <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

²⁴³ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

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.

16.5 Deriving from limited types

In this section, we discuss the implications of deriving from limited types. As usual, let's start with a simple example:

Listing 35: `simple_recs.ads`

```
1 package Simple_Recs is
2
3     type Rec is limited null record;
4
5     type Rec_Derived is new Rec;
6
7 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`

```
1 with Simple_Recs; use Simple_Recs;
2
3 procedure Test_Limitedness is
4     Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6     Dummy_2 := Dummy_1;
7 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

```
1 package Simple_Recs is
2
3     type Rec is null record;
4
5     type Rec_Derived is limited new Rec;
6
7 end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳Limited_Types.Derived_Limited_Type_Nonlimited_Ancessor
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](#)²⁴⁴
- [7.5 Limited Types](#)²⁴⁵

16.5.1 Deriving from limited private types

Of course, we can also derive from limited private types. However, there are more rules in this case than the ones we've seen so far. Let's start with an example:

Listing 38: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is limited null record;
8
9 end Simple_Recs;
```

²⁴⁴ <http://www.ada-auth.org/standards/22rm/html/RM-7-3.html>

²⁴⁵ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

Listing 39: simple_recs-ext.ads

```
1 package Simple_Recs.Ext is
2
3     type Rec_Derived is new Rec;
4
5     -- OR:
6     --
7     -- type Rec_Derived is
8     --     limited new Rec;
9
10 end Simple_Recs.Ext;
```

Listing 40: test_limitedness.adb

```
1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4     Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6     Dummy_2 := Dummy_1;
7 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.

16.5.2 Deriving from non-explicitly limited private types

Up to this point, we have discussed *explicitly limited types* (page 685). Now, let's see how derivation works with *non-explicitly* limited types.

Any type derived from a limited type is always limited, even if the full view of its ancestor is nonlimited. For example, let's modify the full view of `Rec` and make it nonlimited (i.e. make it *not explicitly* limited):

Listing 41: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is null record;
8
9 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

```
1 package Simple_Recs.Ext
2   with Elaborate_Body is
3
4   -- Rec_Derived is derived from Rec, which is a
5   -- limited private type that is nonlimited in
6   -- its full view.
7   --
8   -- Rec_Derived isn't explicitly limited.
9   -- Therefore, it's nonlimited in the private
10  -- part of Simple_Recs.Ext and its package
11  -- body.
12  --
13  type Rec_Derived is new Rec;
14
15 end Simple_Recs.Ext;
```

Listing 43: simple_recs-ext.adb

```
1 package body Simple_Recs.Ext is
2
3   procedure Test_Child_Limitedness is
4     Dummy_1, Dummy_2 : Rec_Derived;
5   begin
6     -- Here, Rec_Derived is a nonlimited
7     -- type because Rec is nonlimited in
8     -- its full view.
9
10    Dummy_2 := Dummy_1;
11  end Test_Child_Limitedness;
12
13 end Simple_Recs.Ext;
```

Listing 44: test_limitedness.adb

```
1 -- We copied the code to the
2 -- Test_Child_Limitedness procedure (in the
3 -- body of the Simple_Recs.Ext package) and
```

(continues on next page)

(continued from previous page)

```
4  --  commented it out here.
5  --
6  --  You may uncomment the code to verify
7  --  that Rec_Derived is limited in this
8  --  procedure.
9  --
10
11 --  with Simple_Recs.Ext; use Simple_Recs.Ext;
12
13 procedure Test_Limitedness is
14   --  Dummy_1, Dummy_2 : Rec_Derived;
15 begin
16   --  Dummy_2 := Dummy_1;
17   null;
18 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 682), if a limited type is nonlimited in its full view, we can copy objects of this type in the private part of the package specification or in the package body.)

Outside of the package, both `Rec` and `Rec_Derived` types are limited types. Therefore, if we uncomment the code in the `Test_Limitedness` procedure, compilation fails there (because `Rec_Derived` is viewed as descending from a limited type).

Deriving from tagged limited private types

The rules for deriving from tagged limited private types are slightly different than the rules we've seen so far. This is because tagged limited types are always *explicitly limited types* (page 685).

Let's look at an example:

Listing 45: `simple_recs.ads`

```
1 package Simple_Recs is
2
3   type Tagged_Rec is tagged limited private;
4
5 private
6
7   type Tagged_Rec is tagged limited null record;
8
9 end Simple_Recs;
```

Listing 46: `simple_recs-ext.ads`

```
1 package Simple_Recs.Ext is
2
3   type Rec_Derived is new
4     Tagged_Rec with private;
5
```

(continues on next page)

(continued from previous page)

```

6 private
7
8   type Rec_Derived is new
9     Tagged_Rec with null record;
10
11 end Simple_Recs.Ext;
```

Listing 47: test_limitedness.adb

```

1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4   Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6   Dummy_2 := Dummy_1;
7 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

```

1 package Simple_Recs.Ext is
2
3   type Rec_Derived is limited new
4     Tagged_Rec with private;
5
6 private
7
8   type Rec_Derived is limited new
9     Tagged_Rec with null record;
10
11 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

```
1 package Simple_Recs is
2
3     type Limited_IF is limited interface;
4
5 end Simple_Recs;
```

Listing 50: simple_recs-ext.ads

```
1 package Simple_Recs.Ext is
2
3     type Rec_Derived is new
4         Limited_IF with private;
5
6 private
7
8     type Rec_Derived is new
9         Limited_IF with null record;
10
11 end Simple_Recs.Ext;
```

Listing 51: test_limitedness.adb

```
1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4     Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6     Dummy_2 := Dummy_1;
7 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

```
1 package Simple_Recs.Ext is
2
3     type Rec_Derived is limited new
4         Limited_IF with private;
```

(continues on next page)

(continued from previous page)

```

5
6 private
7
8     type Rec_Derived is limited new
9         Limited_IF with null record;
10
11 end Simple_Recs.Ext;
```

Listing 53: test_limitedness.adb

```

1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4     Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6     Dummy_2 := Dummy_1;
7 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.

16.6 Immutably Limited Types

According to the Annotated Ada Reference Manual, "an immutably limited type is a type that cannot become nonlimited subsequently in a private part or in a child unit." In fact, while we were talking about *partial and full view of limited types* (page 679), we've seen that limited private types can become nonlimited in their full view. Such limited types are *not* immutably limited.

The Annotated Ada Reference Manual also says that "if a view of the type makes it immutably limited, then no copying (assignment) operations are ever available for objects of the type. This allows other properties; for instance, it is safe for such objects to have access discriminants that have defaults or designate other limited objects." We'll see examples of this later on.

Immutably limited types include:

- explicitly limited types
- tagged limited types (i.e. with the keyword **limited**);
- tagged limited private type;
- limited private type that have at least one *access discriminant* (page 615) with a default expression;
- task types, protected types, and synchronized interfaces;
- any types derived from immutably limited types.

Let's look at a code example that shows instances of immutably limited types:

Listing 54: show_immutably_limited_types.ads

```
1 package Show_Immutably_Limited_Types is
2
3   --
4   -- Explicitly limited type
5   --
6   type Explicitly_Limited_Rec is limited
7   record
8     A : Integer;
9   end record;
10
11  --
12  -- Tagged limited type
13  --
14  type Limited_Tagged_Rec is tagged limited
15  record
16    A : Integer;
17  end record;
18
19  --
20  -- Tagged limited private type
21  --
22  type Limited_Tagged_Private is
23    tagged limited private;
24
25  --
26  -- Limited private type with an access
27  -- discriminant that has a default
28  -- expression
29  --
30  type Limited_Rec_Access_D
31    (AI : access Integer := new Integer) is
32    limited private;
33
34  --
35  -- Task type
36  --
37  task type TT is
38    entry Start;
39    entry Stop;
40  end TT;
41
42  --
43  -- Protected type
44  --
45  protected type PT is
46    function Value return Integer;
47  private
48    A : Integer;
49  end PT;
50
51  --
52  -- Synchronized interface
53  --
54  type SI is synchronized interface;
55
56  --
57  -- A type derived from an immutably
58  -- limited type
59  --
60  type Derived_Immutable is new
```

(continues on next page)

(continued from previous page)

```

61     Explicitly_Limited_Rec;
62
63 private
64
65     type Limited_Tagged_Private is tagged limited
66     record
67         A : Integer;
68     end record;
69
70     type Limited_Rec_Access_D
71     (AI : access Integer := new Integer)
72     is limited
73     record
74         A : Integer;
75     end record;
76
77 end Show_Immutablely_Limited_Types;

```

Listing 55: show_immutablely_limited_types.adb

```

1 package body Show_Immutablely_Limited_Types is
2
3     task body TT is
4     begin
5         accept Start;
6         accept Stop;
7     end TT;
8
9     protected body PT is
10    function Value return Integer is
11        (PT.A);
12    end PT;
13
14 end Show_Immutablely_Limited_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Immutablely_Limited_Types.Example
 MD5: 6bcb9582a10eedc96040ab11cd320153

Build output

```

show_immutablely_limited_types.ads:31:30: warning: coextension will not be
↳ deallocated when its associated owner is deallocated [enabled by default]

```

In the Show_Immutablely_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](#)²⁴⁶

²⁴⁶ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

16.6.1 Non immutably limited types

Not every limited type is immutably limited. We already mentioned untagged private limited types, which can become nonlimited in their full view. In addition, we have nonsynchronized limited interface types. As mentioned earlier in this chapter, a *type derived from a nonsynchronized limited interface* (page 694), can be nonlimited, so it's not immutably limited.

In the Ada Reference Manual

- 7.3.1 Private Operations²⁴⁷
 - 7.5 Limited Types²⁴⁸
-

16.7 Record components of limited type

In this section, we discuss the implications of using components of limited type. Let's start by declaring a record component of limited type:

Listing 56: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Int_Rec is limited record
4         V : Integer;
5     end record;
6
7     type Rec is limited record
8         IR : Int_Rec;
9     end record;
10
11 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 57: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Int_Rec is limited record
4         V : Integer;
5     end record;
6
```

(continues on next page)

²⁴⁷ <http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html>

²⁴⁸ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

(continued from previous page)

```
7  type Rec is record
8      IR : Int_Rec;
9  end record;
10
11 end Simple_Recs;
```

Listing 58: show_implicitly_limited.adb

```
1  with Simple_Recs; use Simple_Recs;
2
3  procedure Show_Implicitly_Limited is
4      A, B : Rec;
5  begin
6      B := A;
7  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](#)²⁴⁹

16.8 Limited types and aggregates

Note: This section was originally written by Robert A. Duff and published as [Gem #1: Limited Types in Ada 2005](#)²⁵⁰ and [Gem #2](#)²⁵¹.

In this section, we focus on using aggregates to initialize limited types.

Historically

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:

²⁴⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-8.html>

²⁵⁰ <https://www.adacore.com/gems/gem-1>

²⁵¹ <https://www.adacore.com/gems/gem-2>

Listing 59: persons.ads

```
1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 package Persons is
5
6     type Limited_Person;
7     type Limited_Person_Access is
8         access all Limited_Person;
9
10    type Limited_Person is limited record
11        Name      : Unbounded_String;
12        Age       : Natural;
13    end record;
14
15 end Persons;
```

Listing 60: show_non_aggregate_init.adb

```
1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 with Persons; use Persons;
5
6 procedure Show_Non_Aggregate_Init is
7     X : Limited_Person;
8 begin
9     X.Name := To_Unbounded_String ("John Doe");
10    X.Age := 25;
11 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.

16.8.1 Full coverage rules for limited types

Previously, we discussed *full coverage rules for aggregates* (page 180). They also apply to limited types.

Historically

The full coverage rules have been aiding maintenance since Ada 83. However, prior to Ada 2005, we couldn't use them for limited types.

Suppose we have the following limited type:

Listing 61: persons.ads

```

1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 package Persons is
5
6     type Limited_Person;
7     type Limited_Person_Access is
8         access all Limited_Person;
9
10    type Limited_Person is limited record
11        Self : Limited_Person_Access :=
12            Limited_Person'Unchecked_Access;
13        Name : Unbounded_String;
14        Age  : Natural;
15        Shoe_Size : Positive;
16    end record;
17
18 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 62: show_aggregate_box_init.adb

```

1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 with Persons; use Persons;
5
6 procedure Show_Aggregate_Box_Init is
7     X : aliased Limited_Person :=
8         (Self      => <>,
9          Name      =>
10             To_Unbounded_String ("John Doe"),
11          Age       => 25,
12          Shoe_Size => 10);
13 begin
14     null;
15 end Show_Aggregate_Box_Init;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↳Aggregates.Full_Coverage_Rules_Limited
 MD5: ded40ff29b53ea5528efba94efaaadbec

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 63: show_aggregate_init.adb

```
1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3 with Persons; use Persons;
4
5 procedure Show_Aggregate_Init is
6
7     X : aliased Limited_Person :=
8         (Self      => null, -- Wrong!
9           Name      =>
10              To_Unbounded_String ("John Doe"),
11           Age       => 25,
12           Shoe_Size => 10);
13 begin
14     X.Self := X'Unchecked\_Access;
15 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 64: show_string_box_init.adb

```
1 procedure Show_String_Box_Init is
2     Uninitialized_Const_Str : constant String :=
3                               (1 .. 10 => <>);
4 begin
5     null;
6 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 65: show_dangerous_string.adb

```
1 procedure Show_Dangerous_String is
2   Uninitialized_String_Var : String (1 .. 10);
3   -- ~~~~~
4   -- no initialization
5
6   Uninitialized_Const_Str : constant String :=
7     Uninitialized_String_Var;
8 begin
9   null;
10 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.

16.9 Constructor functions for limited types

Note: This section was originally written by Robert A. Duff and published as [Gem #3²⁵²](#).

Given that we can use build-in-place aggregates for limited types, the obvious next step is to allow such aggregates to be wrapped in an abstraction — namely, to return them from functions. After all, interesting types are usually private, and we need some way for clients to create and initialize objects.

Historically

Prior to Ada 2005, constructor functions (that is, functions that create new objects and return them) were not allowed for limited types. Since Ada 2005, fully-general constructor functions are allowed.

Let's see an example:

²⁵² <https://www.adacore.com/gems/gem-3>

Listing 66: p.ads

```
1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 package P is
5     task type Some_Task_Type;
6
7     protected type Some_Protected_Type is
8         -- dummy type
9     end Some_Protected_Type;
10
11     type T (<>) is limited private;
12     function Make_T (Name : String) return T;
13         --
14         -- constructor function
15 private
16     type T is limited
17         record
18             Name      : Unbounded_String;
19             My_Task   : Some_Task_Type;
20             My_Prot   : Some_Protected_Type;
21         end record;
22 end P;
```

Listing 67: p.adb

```
1 package body P is
2
3     task body Some_Task_Type is
4     begin
5         null;
6     end Some_Task_Type;
7
8     protected body Some_Protected_Type is
9     end Some_Protected_Type;
10
11     function Make_T (Name : String) return T is
12     begin
13         return (Name =>
14                 To_Unbounded_String (Name),
15                 others => <>);
16     end Make_T;
17
18 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 68: show_constructor_function.adb

```
1 with P; use P;
2
3 procedure Show_Constructor_Function is
4     My_T : T := Make_T
5         (Name => "Bartholomew Cubbins");
```

(continues on next page)

(continued from previous page)

```

6 begin
7   null;
8 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 69: `show_rumplestiltskin_constructor.adb`

```

1 with P; use P;
2
3 procedure Show_Rumplestiltskin_Constructor is
4
5   function Make_Rumplestiltskin return T is
6   begin
7     return Make_T (Name => "Rumplestiltskin");
8   end Make_Rumplestiltskin;
9
10  Rumplestiltskin_Is_My_Name : constant T :=
11    Make_Rumplestiltskin;
12 begin
13   null;
14 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.

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 70: aux.ads

```
1 with Ada.Containers.Ordered_Sets;
2
3 package Aux is
4   generic
5     with package OS is new
6       Ada.Containers.Ordered_Sets (<>);
7   function Gen_Singleton_Set
8     (Element : OS.Element_Type)
9     return OS.Set;
10 end Aux;
```

Listing 71: aux.adb

```
1 package body Aux is
2   function Gen_Singleton_Set
3     (Element : OS.Element_Type)
4     return OS.Set
5   is
6   begin
7     return S : OS.Set := OS.Empty_Set do
8       S.Insert (Element);
9     end return;
10  end Gen_Singleton_Set;
11 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 72: show_set_decl.adb

```
1 with Ada.Containers.Ordered_Sets;
2 with Aux;
3
4 procedure Show_Set_Decl is
5
6   package Integer_Sets is new
7     Ada.Containers.Ordered_Sets
8     (Element_Type => Integer);
9   use Integer_Sets;
10
11  function Singleton_Set is new
12    Aux.Gen_Singleton_Set
13    (OS => Integer_Sets);
14
15  This_Set : Set := Empty_Set;
16  That_Set : Set := Singleton_Set
17    (Element => 42);
```

(continues on next page)

(continued from previous page)

```
18 begin
19     null;
20 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 73: show_set_decl.adb

```
1 with Ada.Containers.Ordered_Sets;
2
3 procedure Show_Set_Decl is
4
5     package Integer_Sets is new
6         Ada.Containers.Ordered_Sets
7         (Element_Type => Integer);
8     use Integer_Sets;
9
10    This_Set : Set;
11 begin
12     null;
13 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".

16.10 Return objects

16.10.1 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*²⁵³.

Previously, we discussed *extended return statements* (page 354). For most types, extended return statements are no big deal — it's just syntactic sugar. But for limited types, this syntax is almost essential:

Listing 74: task_construct_error.ads

```
1 package Task_Construct_Error is
2
3     task type Task_Type (Discriminant : Integer);
```

(continues on next page)

²⁵³ <https://www.adacore.com/gems/ada-gem-10>

(continued from previous page)

```
4
5     function Make_Task (Val : Integer)
6         return Task_Type;
7
8 end Task_Construct_Error;
```

Listing 75: task_construct_error.adb

```
1 package body Task_Construct_Error is
2
3     task body Task_Type is
4     begin
5         null;
6     end Task_Type;
7
8     function Make_Task (Val : Integer)
9         return Task_Type
10    is
11        Result : Task_Type
12            (Discriminant => Val * 3);
13    begin
14        -- some statements...
15        return Result; -- Illegal!
16    end Make_Task;
17
18 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 76: task_construct.ads

```
1 package Task_Construct is
2
3     task type Task_Type (Discriminant : Integer);
4
5     function Make_Task (Val : Integer)
6         return Task_Type;
7
8 end Task_Construct;
```

Listing 77: task_construct.adb

```
1 package body Task_Construct is
2
3     task body Task_Type is
4     begin
5         null;
6     end Task_Type;
7
8     function Make_Task (Val : Integer)
9         return Task_Type is
10    begin
```

(continues on next page)

(continued from previous page)

```

11     return Result : Task_Type
12         (Discriminant => Val * 3)
13     do
14         -- some statements...
15         null;
16     end return;
17 end Make_Task;
18
19 end Task_Construct;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_Statements_Limited_Types.Extended_Return_Limited
 MD5: c91a24f09a76aef1c25d1a55bcbee910

If we call it like this:

Listing 78: show_task_construct.adb

```

1 with Task_Construct; use Task_Construct;
2
3 procedure Show_Task_Construct is
4     My_Task : Task_Type := Make_Task (Val => 42);
5 begin
6     null;
7 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.

16.10.2 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 79: limited_types.ads

```

1 package Limited_Types is
2
3     type Integer_Access is access Integer;
4
5     type Simple_Rec is limited private;

```

(continues on next page)

(continued from previous page)

```
6
7   function Init (I : Integer) return Simple_Rec;
8
9 private
10
11   type Simple_Rec is limited record
12     V : Integer_Access;
13   end record;
14
15 end Limited_Types;
```

Listing 80: limited_types.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with System;
3 with System.Address_Image;
4
5 package body Limited_Types is
6
7   function Init (I : Integer) return Simple_Rec
8   is
9   begin
10    return E : Simple_Rec do
11      E.V := new Integer'(I);
12
13      Put_Line ("E'Address (Init): "
14              & System.Address_Image
15              (E'Address));
16    end return;
17  end Init;
18
19 end Limited_Types;
```

Listing 81: show_limited_init.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with System;
3 with System.Address_Image;
4
5 with Limited_Types;         use Limited_Types;
6
7 procedure Show_Limited_Init is
8 begin
9   declare
10    A : Simple_Rec := Init (0);
11  begin
12    Put_Line ("A'Address (local): "
13            & System.Address_Image
14            (A'Address));
15  end;
16  Put_Line ("----");
17
18  declare
19    B : Simple_Rec := Init (0);
20  begin
21    Put_Line ("B'Address (local): "
22            & System.Address_Image
23            (B'Address));
24  end;
25 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): 00007FFC52732DC8
A'Address (local): 00007FFC52732DC8
----
E'Address (Init): 00007FFC52732DC0
B'Address (local): 00007FFC52732DC0
```

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 82: non_limited_types.ads

```
1 package Non_Limited_Types is
2
3     type Integer_Access is access Integer;
4
5     type Simple_Rec is private;
6
7     function Init (I : Integer)
8         return Simple_Rec;
9
10 private
11
12     type Simple_Rec is record
13         V : Integer_Access;
14     end record;
15
16 end Non_Limited_Types;
```

Listing 83: non_limited_types.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with System;
3 with System.Address_Image;
4
5 package body Non_Limited_Types is
6
7     function Init (I : Integer)
8         return Simple_Rec is
9     begin
10         return E : Simple_Rec do
11             E.V := new Integer'(I);
12
13             Put_Line ("E'Address (Init): "
14                 & System.Address_Image
15                 (E'Address));
```

(continues on next page)

(continued from previous page)

```
16     end return;
17   end Init;
18
19 end Non_Limited_Types;
```

Listing 84: show_non_limited_init_by_copy.adb

```
1  with Ada.Text_IO;           use Ada.Text_IO;
2  with System;
3  with System.Address_Image;
4
5  with Non_Limited_Types;
6  use Non_Limited_Types;
7
8  procedure Show_Non_Limited_Init_By_Copy is
9    A, B : Simple_Rec;
10 begin
11   declare
12     A : Simple_Rec := Init (0);
13   begin
14     Put_Line ("A'Address (local): "
15              & System.Address_Image
16              (A'Address));
17   end;
18   Put_Line ("----");
19
20   declare
21     B : Simple_Rec := Init (0);
22   begin
23     Put_Line ("B'Address (local): "
24              & System.Address_Image
25              (B'Address));
26   end;
27 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): 00007FFFF0D96010
A'Address (local): 00007FFFF0D96138
----
E'Address (Init): 00007FFFF0D96010
B'Address (local): 00007FFFF0D96130
```

In this case, we see that the local object E in the Init function is not the same as the object it's being assigned to in the Show_Non_Limited_Init_By_Copy procedure. In fact, E is being copied to A and B.

16.11 Building objects from constructors

Note: This section was originally written by Robert A. Duff and published as Gem #11: Limited Types in Ada 2005²⁵⁴.

We've earlier seen examples of constructor functions for limited types similar to this:

Listing 85: p.ads

```

1 with Ada.Strings.Unbounded;
2 use  Ada.Strings.Unbounded;
3
4 package P is
5     task type Some_Task_Type;
6
7     protected type Some_Protected_Type is
8         -- dummy type
9     end Some_Protected_Type;
10
11    type T is limited private;
12    function Make_T (Name : String) return T;
13    -- ^^^^^^
14    -- constructor function
15 private
16    type T is limited
17        record
18            Name      : Unbounded_String;
19            My_Task   : Some_Task_Type;
20            My_Prot   : Some_Protected_Type;
21        end record;
22 end P;
```

Listing 86: p.adb

```

1 package body P is
2
3     task body Some_Task_Type is
4     begin
5         null;
6     end Some_Task_Type;
7
8     protected body Some_Protected_Type is
9     end Some_Protected_Type;
10
11    function Make_T (Name : String) return T is
12    begin
13        return (Name =>
14                To_Unbounded_String (Name),
15                others => <>);
16    end Make_T;
17
18 end P;
```

Listing 87: p-aux.ads

```

1 package P.Aux is
2     function Make_Rumplestiltskin return T;
3 end P.Aux;
```

²⁵⁴ <https://www.adacore.com/gems/ada-gem-11>

Listing 88: p-aux.adb

```
1 package body P.Aux is
2
3     function Make_Rumplestiltskin return T is
4     begin
5         return Make_T (Name => "Rumplestiltskin");
6     end Make_Rumplestiltskin;
7
8 end P.Aux;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↳From_Constructors.Building_ObjFrom_Constructors
MD5: 1956721292a82899d244afcd10ff63ed
```

It is useful to consider the various contexts in which these functions may be called. We've already seen things like:

Listing 89: show_rumplestiltskin_constructor.adb

```
1 with P; use P;
2 with P.Aux; use P.Aux;
3
4 procedure Show_Rumplestiltskin_Constructor is
5     Rumplestiltskin_Is_My_Name : constant T :=
6         Make_Rumplestiltskin;
7 begin
8     null;
9 end Show_Rumplestiltskin_Constructor;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↳From_Constructors.Building_ObjFrom_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 90: show_parameter_constructor.adb

```
1 with P; use P;
2 with P.Aux; use P.Aux;
3
4 procedure Show_Parameter_Constructor is
5     procedure Do_Something (X : T) is null;
6 begin
7     Do_Something (X => Make_Rumplestiltskin);
8 end Show_Parameter_Constructor;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↳From_Constructors.Building_ObjFrom_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 91: show_heap_constructor.adb

```

1  with P;      use P;
2  with P.Aux; use P.Aux;
3
4  procedure Show_Heap_Constructor is
5
6      type T_Ref is access all T;
7
8      Global : T_Ref;
9
10     procedure Heap_Alloc is
11         Local : T_Ref;
12         To_Global : Boolean := True;
13     begin
14         Local := new T'(Make_Rumplestiltskin);
15         if To_Global then
16             Global := Local;
17         end if;
18     end Heap_Alloc;
19
20 begin
21     null;
22 end Show_Heap_Constructor;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_From_Constructors.Building_Objs_From_Constructors
 MD5: 8eb794884f1dfbdf1bc4369f45cf8

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 92: show_outer_type.adb

```

1  with P;      use P;
2  with P.Aux; use P.Aux;
3
4  procedure Show_Outer_Type is
5
6      type Outer_Type is limited record
7          This : T;
8          That : T;
9      end record;
10
11     Outer_Obj : Outer_Type :=
12         (This => Make_Rumplestiltskin,
13          That => Make_T (Name => ""));
14
15 begin
16     null;
17 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 93: `show_illegal_constructor.adb`

```
1 with P;      use P;
2 with P.Aux; use P.Aux;
3
4 procedure Show_Illegal_Constructor is
5     Rumplestiltskin_Is_My_Name : T;
6 begin
7     Rumplestiltskin_Is_My_Name :=
8         Make_T (Name => ""); -- Illegal!
9 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 94: `show_illegal_constructor.adb`

```
1 with P;      use P;
2 with P.Aux; use P.Aux;
3
4 procedure Show_Illegal_Constructor is
5     Rumplestiltskin_Is_My_Name : constant T :=
6         Make_T (Name => "");
7     Other : T :=
8         Rumplestiltskin_Is_My_Name; -- Illegal!
9 begin
10    null;
11 end Show_Illegal_Constructor;
```

16.12 Limited types as parameter

Previously, we saw that *parameters can be passed by copy or by reference* (page 357). Also, we discussed the concept of by-copy and by-reference types. *Explicitly limited types* (page 685) are by-reference types. Consequently, parameters of these types are always passed by reference.

For further reading...

As an example of the importance of this rule, consider the case of a lock (as an abstract data type). If such a lock object were passed by copy, the `Acquire` and `Release` operations

would be working on copies of this object, not on the original one. This would lead to timing-dependent bugs.

Let's reuse an example of an explicitly limited type:

Listing 95: simple_recs.ads

```
1 package Simple_Recs is
2
3   type Rec is limited record
4     I : Integer;
5   end record;
6
7 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 96: simple_recs.ads

```
1 with System;
2
3 package Simple_Recs is
4
5   type Rec is limited record
6     I : Integer;
7   end record;
8
9   procedure Proc (R : in out Rec;
10                  A : out System.Address);
11
12 end Simple_Recs;
```

Listing 97: simple_recs.adb

```
1 package body Simple_Recs is
2
3   procedure Proc (R : in out Rec;
4                  A : out System.Address) is
5   begin
6     R.I := 0;
7     A := R'Address;
8   end Proc;
9
10 end Simple_Recs;
```

Listing 98: test.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with System;                use System;
3 with System.Address_Image;
```

(continues on next page)

(continued from previous page)

```
4 with Simple_Recs;           use Simple_Recs;
5
6 procedure Test is
7   R : Rec;
8
9   AR_Proc, AR_Test : System.Address;
10 begin
11   AR_Proc := R'Address;
12
13   Proc (R, AR_Test);
14
15   Put_Line ("R'Address (Proc): "
16             & System.Address_Image (AR_Proc));
17   Put_Line ("R'Address (Test): "
18             & System.Address_Image (AR_Test));
19
20   if AR_Proc = AR_Test then
21     Put_Line ("R was passed by reference.");
22   else
23     Put_Line ("R was passed by copy.");
24   end if;
25
26 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): 00007FFFC414438C
R'Address (Test): 00007FFFC414438C
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](#)²⁵⁵

²⁵⁵ <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

- 6.4.1 Parameter Associations²⁵⁶
 - 7.5 Limited Types²⁵⁷
-

²⁵⁶ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

²⁵⁷ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>