LEARN.
ADACORE.COM

Introduction to Ada

Release 2020-05

Raphaël Amiard and Gustavo A. Hoffmann

Oct 05, 2020

CONTENTS:

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

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

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

CHAPTER

ONE

INTRODUCTION

1.1 History

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

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

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

1.2 Ada today

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

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

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

- Video game programming[2]
- Real-time audio[3]
- Kernel modules[4]

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

[2] https://github.com/AdaDoom3/AdaDoom3
[3] http://www.electronicdesign.com/embedded-revolution/assessing-ada-language-audio-applications
[4] http://www.nihamkin.com/tag/kernel.html

In terms of modern languages, the closest in terms of targets and level of abstraction are probably C++[5] and Rust[6].

1.3 Philosophy

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

- Readability is more important than conciseness. Syntactically this shows through the fact that keywords are preferred to symbols, that no keyword is an abbreviation, etc.
- Very strong typing. It is very easy to introduce new types in Ada, with the benefit of preventing data usage errors.
  - It is similar to many functional languages in that regard, except that the programmer has to be much more explicit about typing in Ada, because there is almost no type inference.
- Explicit is better than implicit. Although this is a Python[7] commandment, Ada takes it way further than any language we know of:
  - There is mostly no structural typing, and most types need to be explicitly named by the programmer.
  - As previously said, there is mostly no type inference.
  - Semantics are very well defined, and undefined behavior is limited to an absolute minimum.
  - The programmer can generally give a *lot* of information about what their program means to the compiler (and other programmers). This allows the compiler to be extremely helpful (read: strict) with the programmer.

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

1.4 SPARK

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

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

There is a dedicated course for the SPARK language[8] but keep in mind that every time we speak about the specification power of Ada during this course, it is power that you can leverage in SPARK to help proving the correctness of program properties ranging from absence of run-time errors to compliance with formally specified functional requirements.

[5] https://en.wikipedia.org/wiki/C%2B%2B
[6] https://www.rust-lang.org/en-US/
[7] https://www.python.org
[8] https://learn.adacore.com/courses/intro-to-spark/index.html

CHAPTER

TWO

IMPERATIVE LANGUAGE

Ada is a multi-paradigm language with support for object orientation and some elements of functional programming, but its core is a simple, coherent procedural/imperative language akin to C or Pascal.

In other languages

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

2.1 Hello world

Here's a very simple imperative Ada program:

```
with Ada.Text_IO;

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

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

If you compile that source with the GNAT compiler and run the executable, you will get an unsurprising result.

```
$ gprbuild greet.adb
using project file [...]_default.gpr
Compile
   [Ada]          greet.adb
Bind
   [gprbind]      greet.bexch
   [Ada]          greet.ali
Link
   [link]         greet.adb

 $ ./greet
Hello, World!
 $
```

There are several noteworthy things in the above program:

- A subprogram in Ada can be either a procedure or a function. A procedure, as illustrated above, does not return a value when called.
- `with` is used to reference external modules that are needed in the procedure. This is similar to `import` in various languages or roughly similar to `#include` in C and C++. We'll see later how they work in detail. Here, we are requesting a standard library module, the `Ada.Text_IO` package, which contains a procedure to print text on the screen: `Put_Line`.
- `Greet` is a procedure, and the main entry point for our first program. Unlike in C or C++, it can be named anything you prefer. The builder will determine the entry point. In our simple example, **`gprbuild`**, GNAT's builder, will use the file you passed as parameter.
- `Put_Line` is a procedure, just like `Greet`, except it is declared in the `Ada.Text_IO` module. It is the Ada equivalent of C's `printf`.
- Comments start with `--` and go to the end of the line. There is no multi-line comment syntax, that is, it is not possible to start a comment in one line and continue it in the next line. The only way to create multiple lines of comments in Ada is by using `--` on each line. For example:

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

In other languages

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

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

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

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

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

2.2 Imperative language - If/Then/Else

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

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

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

procedure Check_Positive is
   N : Integer;
begin
   Put ("Enter an integer value: ");  -- Put a String
   Get (N);  --  Read in an integer value
   if N > 0 then
      Put (N);  --  Put an Integer
      Put_Line (" is a positive number");
```

(continues on next page)

(continued from previous page)

```
   end if;
end Check_Positive;
```

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

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

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

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

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

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

procedure Check_Positive is
   N : Integer;
begin
   Put ("Enter an integer value: ");  -- Put a String
   Get (N);  --  Reads in an integer value
   Put (N);  --  Put an Integer
   if N > 0 then
      Put_Line (" is a positive number");
   else
      Put_Line (" is not a positive number");
   end if;
end Check_Positive;
```

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

Our final variation illustrates an `if` statement with `elsif` sections:

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

procedure Check_Direction is
   N : Integer;
begin
   Put ("Enter an integer value: ");  -- Puts a String
   Get (N);  --  Reads an Integer
   Put (N);  --  Puts an Integer
   if N = 0 or N = 360 then
      Put_Line (" is due east");
   elsif N in 1 .. 89 then
      Put_Line (" is in the northeast quadrant");
   elsif N = 90 then
      Put_Line (" is due north");
   elsif N in 91 .. 179 then
      Put_Line (" is in the northwest quadrant");
   elsif N = 180 then
      Put_Line (" is due west");
   elsif N in 181 .. 269 then
```

(continues on next page)

(continued from previous page)

```
      Put_Line (" is in the southwest quadrant");
   elsif N = 270 then
      Put_Line (" is due south");
   elsif N in 271 .. 359 then
      Put_Line (" is in the southeast quadrant");
   else
      Put_Line (" is not in the range 0..360");
   end if;
end Check_Direction;
```

This example expects the user to input an integer between 0 and 360 inclusive, and displays which quadrant or axis the value corresponds to. The `in` operator in Ada tests whether a scalar value is within a specified range and returns a Boolean result. The effect of the program should be self-explanatory; later we'll see an alternative and more efficient style to accomplish the same effect, through a `case` statement.

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

2.3 Imperative language - Loops

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

2.3.1 For loops

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

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

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

Executing this procedure yields the following output:

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

A few things to note:

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

- Integer'Image is a function that takes an Integer and converts it to a String. It is an example of a language construct known as an *attribute*, indicated by the "'" syntax, which will be covered in more detail later.
- The & symbol is the concatenation operator for String values
- The `end loop` marks the end of the loop

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

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

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

Executing this procedure yields the following output:

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

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

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

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

The `for` loop is more general than what we illustrated here; more on that later.

2.3.2 Bare loops

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

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

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

(continues on next page)

(continued from previous page)

```
      --  Assignment
      I := I + 1; -- There is no I++ short form to increment a variable
   end loop;
end Greet_5b;
```

This example has the same effect as `Greet_5a` shown earlier.

It illustrates several concepts:

- We have declared a variable named `I` between the `is` and the `begin`. This constitutes a *declarative region*. Ada clearly separates the declarative region from the statement part of a subprogram. A declaration can appear in a declarative region but is not allowed as a statement.
- The bare loop statement is introduced by the keyword `loop` on its own and, like every kind of loop statement, is terminated by the combination of keywords `end loop`. On its own, it is an infinite loop. You can break out of it with an `exit` statement.
- The syntax for assignment is `:=`, and the one for equality is `=`. There is no way to confuse them, because as previously noted, in Ada, statements and expressions are distinct, and expressions are not valid statements.

2.3.3 While loops

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

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

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

      I := I + 1;
   end loop;
end Greet_5c;
```

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

This program has the same effect as the previous examples.

In other languages

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

2.4 Imperative language - Case statement

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

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

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

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

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

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

Notable points about Ada's case statement:

- The case expression (here the variable N) must be of a discrete type, i.e. either an integer type or an enumeration type. Discrete types will be covered in more detail later *discrete types* (page 33).
- Every possible value for the case expression needs to be covered by a unique branch of the case statement. This will be checked at compile time.
- A branch can specify a single value, such as `0`; a range of values, such as `1 .. 89`; or any combination of the two (separated by a `|`).
- As a special case, an optional final branch can specify `others`, which covers all values not included in the earlier branches.
- Execution consists of the evaluation of the case expression and then a transfer of control to the statement sequence in the unique branch that covers that value.
- When execution of the statements in the selected branch has completed, control resumes after the `end case`. Unlike C, execution does not fall through to the next branch. So Ada doesn't need (and doesn't have) a `break` statement.

2.5 Imperative language - Declarative regions

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

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

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

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

procedure Main is
   X : Integer;
begin
   X := 0;
   Put_Line ("The initial value of X is " & Integer'Image (X));

   Put_Line ("Performing operation on X...");
   X := X + 1;

   Put_Line ("The value of X now is " & Integer'Image (X));
end Main;
```

Let's look at an example of a nested procedure:

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

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

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

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

procedure Greet is
begin
   loop
      Put_Line ("Please enter your name: ");

      declare
         Name : String := Get_Line;
         --               ^ Call to the Get_Line function
      begin
         exit when Name = "";
         Put_Line ("Hi " & Name & "!");
      end;

  --  Name is undefined here
```

(continues on next page)

(continued from previous page)

```
   end loop;

  Put_Line ("Bye!");
end Greet;
```

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

It returns a String, which, as we will see later, is an *Unconstrained array type* (page 53). For now we simply note that, if you wish to declare a `String` variable and do not know its size in advance, then you need to initialize the variable during its declaration.

2.6 Imperative language - conditional expressions

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

2.6.1 If expressions

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

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

procedure Check_Positive is
   N : Integer;
begin
   Put ("Enter an integer value: ");  --  Put a String
   Get (N);  --  Reads in an integer value
   Put (N);  --  Put an Integer
   declare
      S : String :=
        (if N > 0 then " is a positive number"
         else " is not a positive number");
   begin
      Put_Line (S);
   end;
end Check_Positive;
```

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

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

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

Here's another example:

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

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

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

2.6.2 Case expressions

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

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

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

This program has the same effect as the preceding example.

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

CHAPTER

THREE

SUBPROGRAMS

3.1 Subprograms

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

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

This example shows the declaration and definition of a function:

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

```
--  We define the Increment function

function Increment (I : Integer) return Integer is
begin
    return I + 1;
end Increment;
```

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

```
procedure Proc;

function Func return Integer;
```

Here's another variation on the previous example:

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

In this example, we see that parameters can have default values. When calling the subprogram, you can then omit parameters if they have a default value. Unlike C/C++, a call to a subprogram without parameters does not include parentheses.

This is the implementation of the function above:

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

3.1.1 Subprogram calls

We can then call our subprogram this way:

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

procedure Show_Increment is
   A, B, C : Integer;
begin
   C := Increment_By;
   --              ^ Parameterless call, value of I is 0
   --                and Incr is 1

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

   A := 10;
   B := 3;
   C := Increment_By (A, B);
   --                 ^ Regular parameter passing

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

   A := 20;
   B := 5;
   C := Increment_By (I    => A,
                      Incr => B);
    --                ^ Named parameter passing

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

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

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

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

3.1.2 Nested subprograms

As briefly mentioned earlier, Ada allows you to declare one subprogram inside of another.

This is useful for two reasons:

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

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

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

procedure Show_Increment is
   A, B, C : Integer;

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

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

3.1.3 Function calls

An important feature of function calls in Ada is that the return value at a call cannot be ignored; that is, a function call cannot be used as a statement.

If you want to call a function and do not need its result, you will still need to explicitly store it in a local variable.

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

   Res : Integer := Double (Double (I));
   --               ^ Calling the double function
begin
   Double (I);
   --  ERROR: cannot use call to function "Double" as a statement

   return Res;
end Quadruple;
```

In the GNAT toolchain

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

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

procedure Main is
    Stream : Network_Stream := Get_Stream;
    My_Int : Integer;
    B : Boolean := Read_Int (Stream, My_Int);  -- Warning here, B is never read
begin
   null;
end Main;
```

You then have two solutions to silence this warning:

- Either annotate the variable with pragma Unreferenced, thus:

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

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

3.2 Parameter modes

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

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

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

`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

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

Historically

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

3.3 Subprogram calls

3.3.1 In parameters

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

```
procedure Swap (A, B : Integer) is
   Tmp : Integer;
begin
   Tmp := A;

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

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

3.3.2 In out parameters

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

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

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

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

An in out parameter will allow read and write access to the object passed as parameter, so in the example above, we can see that A is modified after the call to Swap.

Attention: While in out parameters look a bit like references in C++, or regular parameters in Java that are passed by-reference, the Ada language standard does not mandate "by reference" passing for in out parameters except for certain categories of types as will be explained later.

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

3.3.3 Out parameters

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

In other languages

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

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

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

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

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

In the GNAT toolchain

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

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

3.3.4 Forward declaration of subprograms

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

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

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

    procedure Compute_A (V : Natural) is
    begin
       if V > 2 then
          Compute_B (V - 1);
          --  Call to Compute_B
       end if;
    end Compute_A;
begin
   Compute_A (15);
end Mutually_Recursive_Subprograms;
```

3.4 Renaming

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

```
procedure New_Proc renames Original_Proc;
```

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

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

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

procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
  (A_Message : String) is
begin
   Put_Line (A_Message);
end A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
```

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

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming is

   procedure Show (S : String) renames
     A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

begin
   Show ("Hello World!");
end Show_Renaming;
```

Note that the original name (`A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed`) is still visible after the declaration of the Show procedure.

We may also rename subprograms from the standard library. For example, we may rename `Integer'Image` to `Img`:

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

procedure Show_Image_Renaming is

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

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

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

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
```

(continues on next page)

(continued from previous page)

```
procedure Show_Renaming_Defaults is

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

begin
   Show;
end Show_Renaming_Defaults;
```

CHAPTER

FOUR

MODULAR PROGRAMMING

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

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

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

4.1 Packages

Here is an example of a package declaration in Ada:

```
package Week is

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

end Week;
```

And here is how you use it:

```
with Ada.Text_IO; use Ada.Text_IO;
with Week;
--  References the Week package, and adds a dependency from Main
--  to Week

procedure Main is
begin
   Put_Line ("First day of the week is " & Week.Mon);
end Main;
```

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

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

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

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

In other languages

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

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

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

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

In the GNAT toolchain

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

4.2 Using a package

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

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

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

procedure Main is
   use Week;
   --  Make every entity of the Week package directly visible.
begin
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("First day of the week is " & Mon);
end Main;
```

As you can see in the example above:

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

4.3 Package body

In the simple example above, the `Week` package only has declarations and no body. That's not a mistake: in a package specification, which is what is illustrated above, you cannot declare bodies. Those have to be in the package body.

```
package Operations is

   --  Declaration
   function Increment_By
     (I    : Integer;
      Incr : Integer := 0) return Integer;

   function Get_Increment_Value return Integer;

end Operations;
```

```
package body Operations is

   Last_Increment : Integer := 1;

   function Increment_By
     (I    : Integer;
      Incr : Integer := 0) return Integer is
   begin
      if Incr /= 0 then
         Last_Increment := Incr;
      end if;

      return I + Last_Increment;
   end Increment_By;

   function Get_Increment_Value return Integer is
   begin
      return Last_Increment;
   end Get_Increment_Value;

end Operations;
```

Here we can see that the body of the `Increment_By` function has to be declared in the body. Coincidentally, introducing a body allows us to put the `Last_Increment` variable in the body, and make them inaccessible to the user of the `Operations` package, providing a first form of encapsulation.

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

This example shows how `Last_Increment` is used indirectly:

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

procedure Main is
   use Operations;

   I : Integer := 0;
   R : Integer;

   procedure Display_Update_Values is
      Incr : constant Integer := Get_Increment_Value;
   begin
      Put_Line (Integer'Image (I)
                & " incremented by " & Integer'Image (Incr)
                & " is "             & Integer'Image (R));
      I := R;
   end Display_Update_Values;
begin
   R := Increment_By (I);
   Display_Update_Values;
   R := Increment_By (I);
   Display_Update_Values;

   R := Increment_By (I, 5);
   Display_Update_Values;
   R := Increment_By (I);
   Display_Update_Values;

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

4.4 Child packages

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

Important

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

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

```
package Week is

   Mon : constant String := "Monday";
   Tue : constant String := "Tuesday";
   Wed : constant String := "Wednesday";
   Thu : constant String := "Thursday";
   Fri : constant String := "Friday";
```

(continues on next page)

(continued from previous page)

```
   Sat : constant String := "Saturday";
   Sun : constant String := "Sunday";

end Week;
```

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

```
package Week.Child is

   function Get_First_Of_Week return String;

end Week.Child;
```

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

```
package body Week.Child is

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

end Week.Child;
```

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

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

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

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

4.4.1 Child of a child package

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

```
package Week.Child.Grandchild is

   function Get_Second_Of_Week return String;

end Week.Child.Grandchild;
```

```
package body Week.Child.Grandchild is

   function Get_Second_Of_Week return String is
```

(continues on next page)

(continued from previous page)

```
   begin
      return Tue;
   end Get_Second_Of_Week;

end Week.Child.Grandchild;
```

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

```
with Ada.Text_IO;           use Ada.Text_IO;
with Week.Child.Grandchild; use Week.Child.Grandchild;

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

Again, this isn't the limit for the package hierarchy. We could continue to extend the hierarchy of the previous example by implementing a `Week.Child.Grandchild.Grand_grandchild` package.

4.4.2 Multiple children

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

```
package Week.Child_2 is

   function Get_Last_Of_Week return String;

end Week.Child_2;
```

Here, `Week` is still the parent package of the `Child` package, but it's also the parent of the `Child_2` package. In the same way, `Child_2` is obviously one of the child packages of `Week`.

This is the corresponding package body of `Week.Child_2`:

```
package body Week.Child_2 is

   function Get_Last_Of_Week return String is
   begin
      return Sun;
   end Get_Last_Of_Week;

end Week.Child_2;
```

We can now reference both children in our test application:

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

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

4.4.3 Visibility

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

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

```
package Book is

   Title : constant String := "Visible for my children";

   function Get_Title return String;

   function Get_Author return String;

end Book;
```

```
package Book.Additional_Operations is

   function Get_Extended_Title return String;

   function Get_Extended_Author return String;

end Book.Additional_Operations;
```

This is the body of both packages:

```
package body Book is

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

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

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

end Book;
```

```
package body Book.Additional_Operations is

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

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

      return "Book Author: Unknown";
   end Get_Extended_Author;

end Book.Additional_Operations;
```

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

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

```
package body Book.Additional_Operations is

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

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

end Book.Additional_Operations;
```

This is a simple test application for the packages above:

```
with Ada.Text_IO;                 use Ada.Text_IO;
with Book.Additional_Operations; use Book.Additional_Operations;

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

By declaring elements in the body of a package, we can implement encapsulation in Ada. Those elements will only be visible in the package body, but nowhere else. This isn't, however, the only way to achieve encapsulation in Ada: we'll discuss other approaches in the *Privacy* (page 75) chapter.

4.5 Renaming

Previously, we've mentioned that *subprograms can be renamed* (page 21). We can rename packages, too. Again, we use the `renames` keyword for that. The following example renames the `Ada.Text_IO` package as `T_IO`:

```
with Ada.Text_IO;

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

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

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

```
with Ada.Text_IO;

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

CHAPTER

FIVE

STRONGLY TYPED LANGUAGE

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

5.1 What is a type?

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

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

Todo: Expand / clarify (on section: "what is a type?")

5.2 Integers

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

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

procedure Integer_Type_Example is
   --  Declare a signed integer type, and give the bounds
   type My_Int is range -1 .. 20;
   --                        ^ High bound
   --                   ^ Low bound

   --  Like variables, type declarations can only appear in
   --  declarative regions
begin
   for I in My_Int loop
      Put_Line (My_Int'Image (I));
      --              ^ 'Image attribute, converts a value to a
      --                String
```

(continues on next page)

(continued from previous page)

```
   end loop;
end Integer_Type_Example;
```

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

Every type declaration in Ada starts with the `type` keyword (except for *task types* (page 108)). After the type, we can see a range that looks a lot like the ranges that we use in for loops, that defines the low and high bound of the type. Every integer in the inclusive range of the bounds is a valid value for the type.

Ada integer types

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

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

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

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

** is the exponent operator, which means that the first valid value for `Integer` is -2^{31}, and the last valid value is $2^{31} - 1$.

Ada does not mandate the range of the built-in type Integer. An implementation for a 16-bit target would likely choose the range -2^{15} through $2^{15} - 1$.

5.2.1 Operational semantics

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

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

There are two types of overflow checks:

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

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

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

procedure Main is
   type My_Int is range 1 .. 20;
   A : My_Int := 12;
   B : My_Int := 15;
   M : My_Int := (A + B) / 2;
   --  No overflow here, overflow checks are done at
   --  specific boundaries.
begin
   for I in 1 .. M loop
      Put_Line ("Hello, World!");
   end loop;
   --  Loop body executed 13 times
end Main;
```

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

5.3 Unsigned types

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

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

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

procedure Main is
   type Mod_Int is mod 2 ** 5;
   --              ^ Range is 0 .. 31

   A : Mod_Int := 20;
   B : Mod_Int := 15;
   M : Mod_Int := A + B;
   --  No overflow here, M = (20 + 15) mod 32 = 3
begin
   for I in 1 .. M loop
      Put_Line ("Hello, World!");
   end loop;
end Main;
```

Unlike in C/C++, since this behavior is guaranteed by the Ada specification, you can rely on it to implement portable code. Also, being able to leverage the wrapping on arbitrary bounds is very useful – the modulus does not need to be a power of 2 – to implement certain algorithms and data structures, such as ring buffers[9].

5.4 Enumerations

Enumeration types are another nicety of Ada's type system. Unlike C's enums, they are *not* integers, and each new enumeration type is incompatible with other enumeration types. Enumeration types

[9] https://en.m.wikipedia.org/wiki/Circular_buffer

are part of the bigger family of discrete types, which makes them usable in certain situations that we will describe later but one context that we have already seen is a case statement.

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

procedure Enumeration_Example is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
   --  An enumeration type
begin
   for I in Days loop
      case I is
         when Saturday .. Sunday =>
            Put_Line ("Week end!");

         when Monday .. Friday =>
            Put_Line ("Hello on " & Days'Image (I));
            --  'Image attribute, works on enums too
      end case;
   end loop;
end Enumeration_Example;
```

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

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

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

5.5 Floating-point types

5.5.1 Basic properties

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

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

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

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

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

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

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

procedure Floating_Point_Operations is
   A : Float := 2.5;
begin
   A := abs (A - 4.5);
```

(continues on next page)

(continued from previous page)

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

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

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

5.5.2 Precision of floating-point types

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

```
type T is digits <number_of_decimal_digits>;
```

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

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

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

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

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

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

procedure Display_Custom_Floating_Types is
   type T3  is digits 3;
   type T18 is digits 18;

   C1 : constant := 1.0e-4;

   A : T3  := 1.0 + C1;
   B : T18 := 1.0 + C1;
begin
   Put_Line ("The value of A is " & T3'Image (A));
   Put_Line ("The value of B is " & T18'Image (B));
end Display_Custom_Floating_Types;
```

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

5.5.3 Range of floating-point types

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

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

procedure Floating_Point_Range is
   type T_Norm  is new Float range -1.0 .. 1.0;
   A  : T_Norm;
begin
   A := 1.0;
   Put_Line ("The value of A is " & T_Norm'Image (A));
end Floating_Point_Range;
```

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

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

procedure Floating_Point_Range_Exception is
   type T_Norm  is new Float range -1.0 .. 1.0;
   A  : T_Norm;
begin
   A := 2.0;
   Put_Line ("The value of A is " & T_Norm'Image (A));
end Floating_Point_Range_Exception;
```

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

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

procedure Custom_Range_Types is
   type T6_Inv_Trig  is digits 6 range -Pi / 2.0 .. Pi / 2.0;
begin
   null;
end Custom_Range_Types;
```

In this example, we are defining a type called `T6_Inv_Trig`, which has a range from $-\pi/2$ to $\pi/2$ with a minimum precision of 6 digits. (`Pi` is defined in the predefined package `Ada.Numerics`.)

5.6 Strong typing

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

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

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

(continues on next page)

(continued from previous page)

```
   Dist_Imperial : Miles;

   --  Declare a constant
   Dist_Metric : constant Meters := 1000.0;
begin
   --  Not correct: types mismatch
   Dist_Imperial := Dist_Metric * 621.371e-6;
   Put_Line (Miles'Image (Dist_Imperial));
end Illegal_Example;
```

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

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Conv is
   type Meters is new Float;
   type Miles is new Float;
   Dist_Imperial : Miles;
   Dist_Metric : constant Meters := 1000.0;
begin
   Dist_Imperial := Miles (Dist_Metric) * 621.371e-6;
   --               ^ Type conversion, from Meters to Miles
   --  Now the code is correct

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

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

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

procedure Conv is
   type Meters is new Float;
   type Miles is new Float;

   --  Function declaration, like procedure but returns a value.
   function To_Miles (M : Meters) return Miles is
   --                            ^ Return type
   begin
      return Miles (M) * 621.371e-6;
   end To_Miles;

   Dist_Imperial : Miles;
   Dist_Metric   : constant Meters := 1000.0;
begin
   Dist_Imperial := To_Miles (Dist_Metric);
   Put_Line (Miles'Image (Dist_Imperial));
end Conv;
```

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

In other languages

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

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

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

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

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

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

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

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

The offending line must be changed to `F := Float (A) / Float (B);` in order to be accepted by the compiler.

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

5.7 Derived types

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

```
procedure Main is
   --  ID card number type, incompatible with Integer.
   type Social_Security_Number
   is new Integer range 0 .. 999_99_9999;
   --                   ^ Since a SSN has 9 digits max, and cannot be
   --                     negative, we enforce a validity constraint.

   SSN : Social_Security_Number := 555_55_5555;
   --                             ^ You can put underscores as formatting in
   --                               any number.

   I   : Integer;

   Invalid : Social_Security_Number := -1;
   --                                  ^ This will cause a runtime error
```

(continues on next page)

(continued from previous page)

```
   --                                    (and a compile time warning with
   --                                     GNAT)
begin
   I := SSN;                            -- Illegal, they have different types
   SSN := I;                            -- Likewise illegal
   I := Integer (SSN);                  -- OK with explicit conversion
   SSN := Social_Security_Number (I);   -- Likewise OK
end Main;
```

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

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

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

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

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

   type Weekend_Days is new Days range Saturday .. Sunday;
   --  New type, where only Saturday and Sunday are valid literals.
begin
   null;
end Greet;
```

5.8 Subtypes

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

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

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

   --  Declaration of a subtype
   subtype Weekend_Days is Days range Saturday .. Sunday;
   --                            ^ Constraint of the subtype

   M : Days := Sunday;

   S : Weekend_Days := M;
   --  No error here, Days and Weekend_Days are of the same type.
begin
   for I in Days loop
      case I is
         --  Just like a type, a subtype can be used as a
         --  range
         when Weekend_Days =>
            Put_Line ("Week end!");
         when others =>
            Put_Line ("Hello on " & Days'Image (I));
      end case;
```

(continues on next page)

(continued from previous page)

```
   end loop;
end Greet;
```

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

```
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

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

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

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

   subtype Weekend_Days is Days range Saturday .. Sunday;
   Day : Days := Saturday;
   Weekend : Weekend_Days;
begin
   Weekend := Day;
   --         ^ Correct: Same type, subtype constraints are respected
   Weekend := Monday;
   --         ^ Wrong value for the subtype
   --           Compiles, but exception at runtime
end Greet;
```

5.8.1 Subtypes as type aliases

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

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

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

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

   Dist_Imperial : Miles;

   --  Declare a constant
   Dist_Metric : constant Meters := 100.0;
begin
   --  No conversion to Miles type required:
   Dist_Imperial := (Dist_Metric * 1609.0) / 1000.0;

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

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

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

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

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

We could write:

```
subtype Amount is Float;

Paid, Due : Amount;
```

In other languages

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

```
typedef float meters;
```

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

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

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

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

Let's look at another example:

```
subtype Degree_Celsius is Float;
subtype Liquid_Water_Temperature is Degree_Celsius range 0.0 .. 100.0;
subtype Running_Water_Temperature is Liquid_Water_Temperature;
```

In this example, `Liquid_Water_Temperature` isn't an alias of `Degree_Celsius`, since it adds a new constraint that wasn't part of the declaration of the `Degree_Celsius`. However, we do have two type aliases here:

- `Degree_Celsius` is an alias of `Float`;
- `Running_Water_Temperature` is an alias of `Liquid_Water_Temperature`, even if `Liquid_Water_Temperature` itself has a constrained range.

CHAPTER

SIX

RECORDS

So far, all the types we have encountered have values that are not decomposable: each instance represents a single piece of data. Now we are going to see our first class of composite types: records.

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

6.1 Record type declaration

Here is an example of a simple record declaration:

```
type Date is record
   --  The following declarations are components of the record
   Day   : Integer range 1 .. 31;
   Month : Months;
   Year  : Integer range 1 .. 3000; --  You can add custom constraints on fields
end record;
```

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

```
type Date is record
   Day   : Integer range 1 .. 31;
   Month : Months := January;
   --  This component has a default value
   Year  : Integer range 1 .. 3000 := 2012;
   --                                 ^ Default value
end record;
```

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

6.2 Aggregates

```
Ada_Birthday    : Date := (10, December, 1815);
Leap_Day_2020   : Date := (Day => 29, Month => February, Year => 2020);
--                 ^ By name
```

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

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

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

6.3 Component selection

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

Let's look at an example:

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

procedure Record_Selection is

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

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

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

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

begin
   Display_Date (Some_Day);

   Put_Line ("Changing year...");
   Some_Day.Year := 2001;

   Display_Date (Some_Day);
end Record_Selection;
```

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

6.4 Renaming

In previous chapters, we've discussed *subprogram* (page 21) and *package* (page 30) renaming. We can rename record components as well. Instead of writing the full component selection using the dot notation, we can declare an alias that allows us to access the same component. This is useful to simplify the implementation of a subprogram, for example.

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

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

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

Let's look at a complete example:

```
package Dates is

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

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

   procedure Increase_Month (Some_Day : in out Date);

   procedure Display_Month (Some_Day : Date);

end Dates;
```

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

package body Dates is

   procedure Increase_Month (Some_Day : in out Date) is
      --  Renaming components from the Date record
      M : Months  renames Some_Day.Month;
      Y : Integer renames Some_Day.Year;

      --  Renaming function (for Months enumeration)
      function Next (M : Months) return Months
        renames Months'Succ;
   begin
      if M = December then
         M := January;
         Y := Y + 1;
      else
         M := Next (M);
      end if;
   end Increase_Month;

   procedure Display_Month (Some_Day : Date) is
      --  Renaming components from the Date record
      M : Months  renames Some_Day.Month;
      Y : Integer renames Some_Day.Year;
```

(continues on next page)

(continued from previous page)

```
   begin
      Put_Line ("Month: " & Months'Image (M)
                & ", Year:" & Integer'Image (Y));
   end Display_Month;

end Dates;
```

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

procedure Main is
   D : Date := (1, January, 2000);
begin
   Display_Month (D);

   Put_Line ("Increasing month...");
   Increase_Month (D);

   Display_Month (D);
end Main;
```

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

Note that, in the example above, we also rename `Months'Succ` — which is the function that gives us the next month — to `Next`.

CHAPTER

SEVEN

ARRAYS

Arrays provide another fundamental family of composite types in Ada.

7.1 Array type declaration

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

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

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

   type My_Int_Array is array (Index) of My_Int;
   --                                    ^ Type of elements
   --                            ^ Bounds of the array
   Arr : My_Int_Array := (2, 3, 5, 7, 11);
   --                    ^ Array literal, called aggregate in Ada
begin
   for I in Index loop
      Put (My_Int'Image (Arr (I)));
      --                      ^ Take the Ith element
   end loop;
   New_Line;
end Greet;
```

The first point to note is that we specify the index type for the array, rather than its size. Here we declared an integer type named `Index` ranging from 1 to 5, so each array instance will have 5 elements, with the initial element at index 1 and the last element at index 5.

Although this example used an integer type for the index, Ada is more general: any discrete type is permitted to index an array, including *Enum types* (page 35). We will soon see what that means.

Another point to note is that querying an element of the array at a given index uses the same syntax as for function calls: that is, the array object followed by the index in parentheses.

Thus when you see an expression such as `A (B)`, whether it is a function call or an array subscript depends on what `A` refers to.

Finally, notice how we initialize the array with the `(2, 3, 5, 7, 11)` expression. This is another kind of aggregate in Ada, and is in a sense a literal expression for an array, in the same way that 3 is a literal expression for an integer. The notation is very powerful, with a number of properties that we will introduce later. A detailed overview appears in the notation of *aggregate types* (page 61).

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

- `Put`, which displays a string without a terminating end of line

- `New_Line`, which outputs an end of line

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

In other languages

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

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

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

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

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

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

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

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

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

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

   type My_Int_Array is array (Month) of Month_Duration;
   --                          ^ Can use an enumeration type as the
   --                            index

   Tab : constant My_Int_Array :=
   --    ^ constant is like a variable but cannot be
   --      modified
     (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
   --  Maps months to number of days (ignoring leap years)

   Feb_Days : Month_Duration := Tab (Feb);
```

(continues on next page)

(continued from previous page)

```
   --  Number of days in February
begin
   for M in Month loop
      Put_Line
        (Month'Image (M) & " has "
         & Month_Duration'Image (Tab (M))  & " days.");
         --                                ^ Concatenation operator
   end loop;
end Month_Example;
```

In the example above, we are:

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

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

7.2 Indexing

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

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

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

procedure Greet is
   type My_Int is range 0 .. 1000;

   type My_Index   is range 1 .. 5;
   type Your_Index is range 1 .. 5;

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

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

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

procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;
   type My_Int_Array is array (Index) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
```

(continues on next page)

(continued from previous page)

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

7.3 Simpler array declarations

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

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

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

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

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

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

7.4 Range attribute

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

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

procedure Range_Example is
   type My_Int is range 0 .. 1000;
   type My_Int_Array is array (1 .. 5) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Tab'Range loop
   --          ^ Gets the range of Tab
```

(continues on next page)

(continued from previous page)

```
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Range_Example;
```

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

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

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

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

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

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

7.5 Unconstrained arrays

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

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

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

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

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

   type Workload_Type is array (Days range <>) of Natural;
   --  Indefinite array type
   --                           ^ Bounds are of type Days,
   --                             but not known

   Workload : constant Workload_Type (Monday .. Friday) :=
   --                                  ^ Specify the bounds
   --                                    when declaring
      (Friday => 7, others => 8);
   --               ^ Default value
   --  ^ Specify element by name of index
begin
   for I in Workload'Range loop
```

(continues on next page)

(continued from previous page)

```
      Put_Line (Integer'Image (Workload (I)));
   end loop;
end Unconstrained_Array_Example;
```

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

If we define the index as `Discrete_Type range <>` then `Discrete_Type` serves as the type of the index, but different array instances may have different bounds from this type

An array type that is defined with the `Discrete_Type range <>` syntax for its index is referred to as an unconstrained array type, and, as illustrated above, the bounds need to be provided when an instance is created.

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

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

In other languages

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

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

7.6 Predefined array type: String

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

Here is how the string type is defined in Ada:

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

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

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

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

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

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

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

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

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

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

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

procedure Main is
   type Integer_Array is array (Natural range <>) of Integer;

   My_Array : constant Integer_Array := (1, 2, 3, 4);
   --                 ^ Bounds are automatically computed
   --                   from initialization value
begin
    null;
end Main;
```

Attention: As you can see above, the standard String type in Ada is an array. As such, it shares the advantages and drawbacks of arrays: a String value is stack allocated, it is accessed efficiently, and its bounds are immutable.

If you want something akin to C++'s `std::string`, you can use *Unbounded Strings* (page 170) from Ada's standard library. This type is more like a mutable, automatically managed string buffer to which you can add content.

7.7 Restrictions

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

```
declare
   A : String;
begin
   A := "World";
end;
```

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

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

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

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

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

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

procedure Indefinite_Subtypes is
    function Get_Number return Integer is
    begin
        return Integer'Value (Get_Line);
    end Get_Number;

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

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

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

7.8 Returning unconstrained arrays

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

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

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

procedure Main is

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

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

begin
   Put_Line ("First day is " & Get_Day_Name (Days'First));
end Main;
```

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

In other languages

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

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

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

7.9 Declaring arrays (2)

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

   Names : constant Days_Name_Type :=
     ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");
   --  Initial value given by aggregate
begin
   for I in Names'Range loop
      Put_Line (Names (I));
   end loop;
end Show_Days;
```

7.10 Array slices

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

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

procedure Main is
    Buf : String := "Hello ...";

    Full_Name : String := "John Smith";
begin
    Buf (7 .. 9) := "Bob";
    --  Careful! This works because the string on the right side is the
    --  same length as the replaced slice!

    Put_Line (Buf);  --  Prints "Hello Bob"

    Put_Line ("Hi " & Full_Name (1 .. 4)); --  Prints "Hi John"
end Main;
```

As we can see above, you can use a slice on the left side of an assignment, to replace only part of an array.

A slice of an array is of the same type as the array, but has a different subtype, constrained by the bounds of the slice.

Attention: Ada has multidimensional arrays[10], which are not covered in this course. Slices will only work on one dimensional arrays.

7.11 Renaming

So far, we've seen that the following elements can be renamed: *subprograms* (page 21), *packages* (page 30), and *record components* (page 47). We can also rename objects by using the `renames` keyword. This allows for creating alternative names for these objects. Let's look at an example:

[10] http://www.adaic.org/resources/add_content/standards/12rm/html/RM-3-6.html

```
package Measurements is

   subtype Degree_Celsius is Float;

   Current_Temperature : Degree_Celsius;

end Measurements;
```

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

procedure Main is
   subtype Degrees is Measurements.Degree_Celsius;

   T : Degrees renames Measurements.Current_Temperature;
begin
    T := 5.0;

    Put_Line (Degrees'Image (T));
    Put_Line (Degrees'Image (Measurements.Current_Temperature));

    T := T + 2.5;

    Put_Line (Degrees'Image (T));
    Put_Line (Degrees'Image (Measurements.Current_Temperature));
end Main;
```

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

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

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

Note that, in the example above, we're using `Degrees` as an alias of `Degree_Celsius`. We discussed this method *earlier in the course* (page 42).

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

```
package Colors is

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

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

   procedure Reverse_It (X : in out Color_Array);

end Colors;
```

```
package body Colors is

   procedure Reverse_It (X : in out Color_Array) is
   begin
      for I in X'First .. (X'Last + X'First) / 2 loop
         declare
            Tmp     : Color;
            X_Left  : Color renames X (I);
```

(continues on next page)

(continued from previous page)

```
            X_Right : Color renames X (X'Last + X'First - I);
         begin
            Tmp     := X_Left;
            X_Left  := X_Right;
            X_Right := Tmp;
         end;
      end loop;
   end Reverse_It;

end Colors;
```

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

with Colors; use Colors;

procedure Test_Reverse_Colors is

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

begin
   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

   New_Line;
   Put_Line ("Reversing My_Color...");
   New_Line;
   Reverse_It (My_Colors);

   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

end Test_Reverse_Colors;
```

In the example above, package `Colors` implements the procedure `Reverse_It` by declaring new names for two positions of the array. The actual implementation becomes easy to read:

```
begin
   Tmp     := X_Left;
   X_Left  := X_Right;
   X_Right := Tmp;
end;
```

Compare this to the alternative version without renaming:

```
begin
   Tmp                     := X (I);
   X (I)                   := X (X'Last + X'First - I);
   X (X'Last + X'First - I) := Tmp;
end;
```

CHAPTER

EIGHT

MORE ABOUT TYPES

8.1 Aggregates: A primer

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

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

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

This means that the following code is incorrect:

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

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

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

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

However, note that as soon as you used a named association, all subsequent components likewise need to be specified with names associations.

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

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

   Origin   : Point := (X | Y => <>);   -- use the default values
   Origin_2 : Point := (others => <>);  -- likewise use the defaults

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

8.2 Overloading and qualified expressions

Ada has a general concept of name overloading, which we saw earlier in the section on *enumeration types* (page 35).

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

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

This is a common concept in programming languages, called overloading[11], or name overloading.

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

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

   function Convert (Self : SSID) return Integer;
   function Convert (Self : SSID) return String;
end Pkg;
```

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

procedure Main is
   S : String := Convert (123_145_299);
   --            ^ Valid, will choose the proper Convert
begin
   Put_Line (S);
end Main;
```

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

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

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

   function Convert (Self : SSID) return Integer;
   function Convert (Self : SSID) return String;
   function Convert (Self : Integer) return String;
end Pkg;
```

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

procedure Main is
   S : String := Convert (123_145_299);
```

(continues on next page)

[11] https://en.m.wikipedia.org/wiki/Function_overloading

(continued from previous page)

```
   --            ^ Invalid, which convert should we call?

   S2 : String := Convert (SSID'(123_145_299));
   --                     ^ We specify that the type of the expression is
   --                       SSID.

   --  We could also have declared a temporary

   I : SSID := 123_145_299;

   S3 : String := Convert (I);
begin
   Put_Line (S);
end Main;
```

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

```
package Qual_Expr is
   type Point is record
      A, B : Integer;
   end record;

   P : Point := Point'(12, 15);

   A : Integer := Integer'(12);
end Qual_Expr;
```

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

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

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

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

```
X : Integer := Natural'(1);
```

8.3 Access types (pointers)

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

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

1. One approach, which we have already seen, is to provide alternative features so that the programmer does not need to use pointers. Parameter modes, arrays, and varying size types are all constructs that can replace typical pointer usages in C.
2. Second, Ada has made pointers as safe and restricted as possible, but allows "escape hatches" when the programmer explicitly requests them and presumably will be exercising such features with appropriate care.

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

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

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

```
with Dates; use Dates;

package Access_Types is
    --  Declare an access type
    type Date_Acc is access Date;
    --                      ^ "Designated type"
    --                      ^ Date_Acc values point to Date objects

    D : Date_Acc := null;
    --             ^ Literal for "access to nothing"
    --  ^ Access to date
end Access_Types;
```

This illustrates how to:

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

In line with Ada’s strong typing philosophy, if you declare a second access type whose designated type is Date, the two access types will be incompatible with each other, and you will need an explicit type conversion to convert from one to the other:

```
with Dates; use Dates;

package Access_Types is
    --  Declare an access type
    type Date_Acc   is access Date;
    type Date_Acc_2 is access Date;

    D  : Date_Acc   := null;
    D2 : Date_Acc_2 := D;
    --                 ^ Invalid! Different types

    D3 : Date_Acc_2 := Date_Acc_2 (D);
    --                 ^ Valid with type conversion
end Access_Types;
```

In other languages

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

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

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

   type Point_Access is access Point;
end Access_Types;
```

8.3.1 Allocation (by type)

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

```
with Dates; use Dates;

package Access_Types is
    type Date_Acc is access Date;

    D : Date_Acc := new Date;
    --             ^ Allocate a new Date record
end Access_Types;
```

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

```
with Dates; use Dates;

package Access_Types is
   type String_Acc is access String;
   --                      ^ Access to unconstrained array type
   Msg : String_Acc;
   --    ^ Default value is null

   Buffer : String_Acc := new String (1 .. 10);
   --                                ^ Constraint required
end Access_Types;
```

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

```
with Dates; use Dates;

package Access_Types is
   type Date_Acc is access Date;
   type String_Acc is access String;

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

8.3.2 Dereferencing

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

```
with Dates; use Dates;

package Access_Types is
   type Date_Acc is access Date;

   D     : Date_Acc := new Date'(30, November, 2011);

   Today : Date := D.all;
   --              ^ Access value dereference
   J     : Integer := D.Day;
   --                 ^ Implicit dereference for record and array components
   --                 Equivalent to D.all.day
end Access_Types;
```

8.3.3 Other features

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

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

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

Attention: The guideline in Ada is that most of the time you can avoid manual allocation, and you should.

There are many ways to avoid manual allocation, some of which have been covered (such as parameter modes). The language also provides library abstractions to avoid pointers:

1. One is the use of *containers* (page 139). Containers help users avoid pointers, because container memory is automatically managed.
2. A container to note in this context is the Indefinite holder[12]. This container allows you to store a value of an indefinite type such as String.
3. GNATCOLL has a library for smart pointers, called Refcount[13] Those pointers' memory is automatically managed, so that when an allocated object has no more references to it, the memory is automatically deallocated.

8.4 Mutually recursive types

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

```
package Simple_List is
   type Node;
   --  This is an incomplete type declaration, which is
   --  completed in the same declarative region.
```

(continues on next page)

[12] http://www.ada-auth.org/standards/12rat/html/Rat12-8-5.html
[13] https://github.com/AdaCore/gnatcoll-core/blob/master/src/gnatcoll-refcount.ads

(continued from previous page)

```
   type Node_Acc is access Node;

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

8.5 More about records

8.5.1 Dynamically sized record types

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

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

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

```
with Runtime_Length; use Runtime_Length;

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

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

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

    G : Growable_Stack;
end Var_Size_Record;
```

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

8.5.2 Records with discriminant

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

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

```
package Var_Size_Record_2 is
    type Items_Array is array (Positive range <>) of Integer;
```

(continues on next page)

(continued from previous page)

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

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

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

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

   P : Point;
   --  ERROR: Point is indefinite, so you need to specify the discriminants
   --  or give a default value

   P2 : Point (1, 2);
   P3 : Point := (1, 2);
   --  Those two declarations are equivalent.

end Test_Discriminants;
```

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

In most other respects discriminants behave like regular fields: You have to specify their values in aggregates, as seen above, and you can access their values via the dot notation.

```
with Var_Size_Record_2; use Var_Size_Record_2;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   procedure Print_Stack (G : Growable_Stack) is
   begin
      Put ("<Stack, items: [");
      for I in G.Items'Range loop
         exit when I > G.Len;
         Put (" " & Integer'Image (G.Items (I)));
      end loop;
      Put_Line ("]>");
   end Print_Stack;

   S : Growable_Stack :=
     (Max_Len => 128, Items => (1, 2, 3, 4, others => <>), Len => 4);
begin
   Print_Stack (S);
end Main;
```

8.5.3 Variant records

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

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

```
package Variant_Record is
   type Expr;                          --  Forward declaration of Expr
   type Expr_Access is access Expr; --  Access to a Expr

   type Expr_Kind_Type is (Bin_Op_Plus, Bin_Op_Minus, Num);
   --  A regular enumeration type

   type Expr (Kind : Expr_Kind_Type) is record
      --       ^ The discriminant is an enumeration value
      case Kind is
         when Bin_Op_Plus | Bin_Op_Minus =>
            Left, Right : Expr_Access;
         when Num =>
            Val : Integer;
      end case;
      --  Variant part. Only one, at the end of the record
      --  definition, but can be nested
   end record;
end Variant_Record;
```

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

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

```
with Variant_Record; use Variant_Record;

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

Here is how you could write an evaluator for expressions:

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

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

   E : Expr := (Bin_Op_Plus,
                new Expr'(Bin_Op_Minus,
                          new Expr'(Num, 12), new Expr'(Num, 15)),
                new Expr'(Num, 3));
begin
   Put_Line (Integer'Image (Eval_Expr (E)));
end Main;
```

In other languages

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

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

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

8.6 Fixed-point types

8.6.1 Decimal fixed-point types

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

The syntax for a simple decimal fixed-point type is

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

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

Several attributes are useful for dealing with decimal types:

Attribute Name	Meaning
First	The first value of the type
Last	The last value of the type
Delta	The delta value of the type

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

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

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

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

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

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

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

In this example, the result of the operation $0.001 * 0.5$ is 0.0005. Since this value is not representable for the T3_D3 type because the delta value is 0.001, the actual value stored in variable A is zero. However, accuracy is preserved during the arithmetic operations if the target has sufficient precision, and the value displayed for C is 0.000500.

8.6.2 Fixed-point types

Ordinary fixed-point types are similar to decimal fixed-point types in that the values are, in effect, scaled integers. The difference between them is in the scale factor: for a decimal fixed-point type, the scaling, given explicitly by the type's `delta`, is always a power of ten.

In contrast, for an ordinary fixed-point type, the scaling is defined by the type's `small`, which is derived from the specified `delta` and, by default, is a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types.

Note: Ordinary fixed-point types can be thought of being closer to the actual representation on the machine, since hardware support for decimal fixed-point arithmetic is not widespread (rescalings by a power of ten), while ordinary fixed-point types make use of the available integer shift instructions.

The syntax for an ordinary fixed-point type is

```
type <type-name> is delta <delta-value> range <lower-bound> .. <upper-bound>;
```

By default the compiler will choose a scale factor, or `small`, that is a power of 2 no greater than <delta-value>.

For example, we may define a normalized range between -1.0 and 1.0 as following:

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

procedure Normalized_Fixed_Point_Type is
   D : constant := 2.0 ** (-31);
```

(continues on next page)

(continued from previous page)

```
   type TQ31 is delta D range -1.0 .. 1.0 - D;
begin
   Put_Line ("TQ31 requires " & Integer'Image (TQ31'Size) & " bits");
   Put_Line ("The delta    value of TQ31 is " & TQ31'Image (TQ31'Delta));
   Put_Line ("The minimum  value of TQ31 is " & TQ31'Image (TQ31'First));
   Put_Line ("The maximum  value of TQ31 is " & TQ31'Image (TQ31'Last));
end Normalized_Fixed_Point_Type;
```

In this example, we are defining a 32-bit fixed-point data type for our normalized range. When running the application, we notice that the upper bound is close to one, but not exact one. This is a typical effect of fixed-point data types — you can find more details in this discussion about the Q format[14]. We may also rewrite this code with an exact type definition:

```
procedure Normalized_Adapted_Fixed_Point_Type is
   type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);
begin
   null;
end Normalized_Adapted_Fixed_Point_Type;
```

We may also use any other range. For example:

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

procedure Custom_Fixed_Point_Range is
   type T_Inv_Trig is delta 2.0 ** (-15) * Pi range -Pi / 2.0 .. Pi / 2.0;
begin
   Put_Line ("T_Inv_Trig requires " & Integer'Image (T_Inv_Trig'Size)
             & " bits");
   Put_Line ("The delta    value of T_Inv_Trig is "
             & T_Inv_Trig'Image (T_Inv_Trig'Delta));
   Put_Line ("The minimum  value of T_Inv_Trig is "
             & T_Inv_Trig'Image (T_Inv_Trig'First));
   Put_Line ("The maximum  value of T_Inv_Trig is "
             & T_Inv_Trig'Image (T_Inv_Trig'Last));
end Custom_Fixed_Point_Range;
```

In this example, we are defining a 16-bit type called `T_Inv_Trig`, which has a range from $-\pi/2$ to $\pi/2$.

All standard operations are available for fixed-point types. For example:

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

procedure Fixed_Point_Op is
   type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);

   A, B, R : TQ31;
begin
   A := 0.25;
   B := 0.50;
   R := A + B;
   Put_Line ("R is " & TQ31'Image (R));
end Fixed_Point_Op;
```

As expected, R contains 0.75 after the addition of A and B.

In fact the language is more general than these examples imply, since in practice it is typical to need to multiply or divide values from different fixed-point types, and obtain a result that may be of a third fixed-point type. The details are outside the scope of this introductory course.

[14] https://en.wikipedia.org/wiki/Q_(number_format)

It is also worth noting, although again the details are outside the scope of this course, that you can explicitly specify a value for an ordinary fixed-point type's `small`. This allows non-binary scaling, for example:

```
type Angle is delta 1.0/3600.0 range 0.0 .. 360.0 - 1.0/3600.0;
for Angle'Small use Angle'Delta;
```

8.7 Character types

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

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

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

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

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

   M := 'a';

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

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

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

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

CHAPTER

NINE

PRIVACY

One of the main principles of modular programming, as well as object oriented programming, is encapsulation[15].

Encapsulation, briefly, is the concept that the implementer of a piece of software will distinguish between the code's public interface and its private implementation.

This is not only applicable to software libraries but wherever abstraction is used.

In Ada, the granularity of encapsulation is a bit different from most object-oriented languages, because privacy is generally specified at the package level.

9.1 Basic encapsulation

```
package Encapsulate is
   procedure Hello;

private

   procedure Hello2;
   --  Not visible from external units
end Encapsulate;
```

```
with Encapsulate;

procedure Main is
begin
   Encapsulate.Hello;
   Encapsulate.Hello2;
   --  Invalid: Hello2 is not visible
end Main;
```

9.2 Abstract data types

With this high-level granularity, it might not seem obvious how to hide the implementation details of a type. Here is how it can be done in Ada:

```
package Stacks is
   type Stack is private;
   --  Declare a private type: You cannot depend on its
   --  implementation. You can only assign and test for
   --  equality.
```

(continues on next page)

[15] https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

(continued from previous page)

```
   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private

   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is record
      Top     : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

In the above example, we define a stack type in the public part (known as the *visible part* of the package spec in Ada), but the exact representation of that type is private.

Then, in the private part, we define the representation of that type. We can also declare other types that will be used as *helpers* for our main public type. This is useful since declaring helper types is common in Ada.

A few words about terminology:

- The Stack type as viewed from the public part is called the partial view of the type. This is what clients have access to.
- The Stack type as viewed from the private part or the body of the package is called the full view of the type. This is what implementers have access to.

From the point of view of the client (the *with*'ing unit), only the public (visible) part is important, and the private part could as well not exist. It makes it very easy to read linearly the part of the package that is important for you.

```
--  No need to read the private part to use the package
package Stacks is
   type Stack is private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private
   ...
end Stacks;
```

Here is how the `Stacks` package would be used:

```
--  Example of use
with Stacks; use Stacks;

procedure Test_Stack is
   S : Stack;
   Res : Integer;
begin
   Push (S, 5);
   Push (S, 7);
   Pop (S, Res);
end Test_Stack;
```

9.3 Limited types

Ada's *limited type* facility allows you to declare a type for which assignment and comparison operations are not automatically provided.

```
package Stacks is
   type Stack is limited private;
   --  Limited type. Cannot assign nor compare.

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is limited record
      Top     : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

```
with Stacks; use Stacks;

procedure Main is
   S, S2 : Stack;
begin
   S := S2;
   --  Illegal: S is limited.
end Main;
```

This is useful because, for example, for some data types the built-in assignment operation might be incorrect (for example when a deep copy is required).

Ada does allow you to overload the comparison operators = and /= for limited types (and to override the built-in declarations for non-limited types).

Ada also allows you to implement special semantics for assignment via controlled types[16]. However, in some cases assignment is simply inappropriate; one example is the `File_Type` from the `Ada.Text_IO` package, which is declared as a limited type and thus attempts to assign one file to another would be detected as illegal.

9.4 Child packages & privacy

We've seen previously (in the *child packages section* (page 26)) that packages can have child packages. Privacy plays an important role in child packages. This section discusses some of the privacy rules that apply to child packages.

Although the private part of a package `P` is meant to encapsulate information, certain parts of a child package `P.C` can have access to this private part of `P`. In those cases, information from the private part of `P` can then be used as if it were declared in the public part of its specification. To be more specific, the body of `P.C` and the private part of the specification of `P.C` have access to the private part of `P`. However, the public part of the specification of `P.C` only has access to the public part of `P`'s specification. The following table summarizes this:

[16] https://www.adaic.org/resources/add_content/standards/12rm/html/RM-7-6.html

Part of a child package	Access to the private part of its parent's specification
Specification: public part	
Specification: private part	X
Body	X

The rest of this section shows examples of how this access to private information actually works for child packages

Let's first look at an example where the body of a child package `P.C` has access to the private part of the specification of its parent `P`. We've seen, in a previous source-code example, that the `Hello2` procedure declared in the private part of the `Encapsulate` package cannot be used in the `Main` procedure, since it's not visible there. This limitation doesn't apply, however, for parts of the child packages of the `Encapsulate` package. In fact, the body of its child package `Encapsulate.Child` has access to the `Hello2` procedure and can call it there, as you can see in the implementation of the `Hello3` procedure of the `Child` package:

```
package Encapsulate is
   procedure Hello;

private

   procedure Hello2;
   --  Not visible from external units
   --  But visible in child packages
end Encapsulate;
```

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

package body Encapsulate is

   procedure Hello is
   begin
      Put_Line ("Hello");
   end Hello;

   procedure Hello2 is
   begin
      Put_Line ("Hello #2");
   end Hello2;

end Encapsulate;
```

```
package Encapsulate.Child is

   procedure Hello3;

end Encapsulate.Child;
```

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

package body Encapsulate.Child is

   procedure Hello3 is
   begin
      --  Using private procedure Hello2 from the parent package
      Hello2;
      Put_Line ("Hello #3");
   end Hello3;

end Encapsulate.Child;
```

```
with Encapsulate.Child;

procedure Main is
begin
   Encapsulate.Child.Hello3;
end Main;
```

The same mechanism applies to types declared in the private part of a parent package. For instance, the body of a child package can access components of a record declared in the private part of its parent package. Let's look at an example:

```
package My_Types is

   type Priv_Rec is private;

private

   type Priv_Rec is record
     Number : Integer := 42;
   end record;

end My_Types;
```

```
package My_Types.Ops is

   procedure Display (E : Priv_Rec);

end My_Types.Ops;
```

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

package body My_Types.Ops is

   procedure Display (E : Priv_Rec) is
   begin
      Put_Line ("Priv_Rec.Number: " & Integer'Image (E.Number));
   end Display;

end My_Types.Ops;
```

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

with My_Types;     use My_Types;
with My_Types.Ops; use My_Types.Ops;

procedure Main is
   E : Priv_Rec;
begin
   Put_Line ("Presenting information:");

   --  The following line would trigger a compilation error here:
   --  Put_Line ("Priv_Rec.Number: " & Integer'Image (E.Number));

   Display (E);
end Main;
```

In this example, we don't have access to the `Number` component of the record type `Priv_Rec` in the `Main` procedure. You can see this in the call to `Put_Line` that has been commented-out in the implementation of `Main`. Trying to access the `Number` component there would trigger a compilation error. But we do have access to this component in the body of the `My_Types.Ops` package, since it's a child package of the `My_Types` package. Therefore, `Ops`'s body has access to

the declaration of the `Priv_Rec` type — which is in the private part of its parent, the `My_Types` package. For this reason, the same call to `Put_Line` that would trigger a compilation error in the `Main` procedure works fine in the `Display` procedure of the `My_Types.Ops` package.

This kind of privacy rules for child packages allows for extending the functionality of a parent package and, at the same time, retain its encapsulation.

As we mentioned previously, in addition to the package body, the private part of the specification of a child package `P.C` also has access to the private part of the specification of its parent `P`. Let's look at an example where we declare an object of private type `Priv_Rec` in the private part of the child package `My_Types.Child` *and* initialize the `Number` component of the `Priv_Rec` record directly:

```
package My_Types.Child is

private

   E : Priv_Rec := (Number => 99);

end My_Types.Ops;
```

As expected, we wouldn't be able to initialize this component if we moved this declaration to the public (visible) part of the same child package:

```
package My_Types.Child is

   E : Priv_Rec := (Number => 99);

end My_Types.Ops;
```

The declaration above triggers a compilation error, since type `Priv_Rec` is private. Because the public part of `My_Types.Child` is also visible outside the child package, Ada cannot allow accessing private information in this part of the specification.

CHAPTER

TEN

GENERICS

10.1 Introduction

Generics are used for metaprogramming in Ada. They are useful for abstract algorithms that share common properties with each other.

Either a subprogram or a package can be generic. A generic is declared by using the keyword `generic`. For example:

```
generic
   type T is private;
   --  Declaration of formal types and objects
--  Below, we could use one of the following:
--  <procedure | function | package>
procedure Operator (Dummy : in out T);
```

```
procedure Operator (Dummy : in out T) is
begin
   null;
end Operator;
```

10.2 Formal type declaration

Formal types are abstractions of a specific type. For example, we may want to create an algorithm that works on any integer type, or even on any type at all, whether a numeric type or not. The following example declares a formal type T for the `Set` procedure.

```
generic
   type T is private;
   --  T is a formal type that indicates that any type can be used,
   --  possibly a numeric type or possibly even a record type.
procedure Set (Dummy : T);
```

```
procedure Set (Dummy : T) is
begin
   null;
end Set;
```

The declaration of T as `private` indicates that you can map any type to it. But you can also restrict the declaration to allow only some types to be mapped to that formal type. Here are some examples:

Formal Type	Format
Any type	`type T is private;`
Any discrete type	`type T is (<>);`
Any floating-point type	`type T is digits <>;`

10.3 Formal object declaration

Formal objects are similar to subprogram parameters. They can reference formal types declared in the formal specification. For example:

```
generic
   type T is private;
   X : in out T;
   --  X can be used in the Set procedure
procedure Set (E : T);
```

```
procedure Set (E : T) is
   pragma Unreferenced (E, X);
begin
   null;
end Set;
```

Formal objects can be either input parameters or specified using the `in out` mode.

10.4 Generic body definition

We don't repeat the `generic` keyword for the body declaration of a generic subprogram or package. Instead, we start with the actual declaration and use the generic types and objects we declared. For example:

```
generic
   type T is private;
   X : in out T;
procedure Set (E : T);
```

```
procedure Set (E : T) is
--  Body definition: "generic" keyword is not used
begin
   X := E;
end Set;
```

10.5 Generic instantiation

Generic subprograms or packages can't be used directly. Instead, they need to be instantiated, which we do using the new keyword, as shown in the following example:

```
generic
   type T is private;
   X : in out T;
   --  X can be used in the Set procedure
procedure Set (E : T);
```

```
procedure Set (E : T) is
begin
   X := E;
end Set;
```

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

procedure Show_Generic_Instantiation is

   Main    : Integer := 0;
   Current : Integer;

   procedure Set_Main is new Set (T => Integer,
                                  X => Main);
   --  Here, we map the formal parameters to actual types and objects.
   --
   --  The same approach can be used to instantiate functions or
   --  packages, e.g.:
   --  function Get_Main is new ...
   --  package Integer_Queue is new ...

begin
   Current := 10;

   Set_Main (Current);
   Put_Line ("Value of Main is " & Integer'Image (Main));
end Show_Generic_Instantiation;
```

In the example above, we instantiate the procedure `Set` by mapping the formal parameters `T` and `X` to actual existing elements, in this case the `Integer` type and the `Main` variable.

10.6 Generic packages

The previous examples focused on generic subprograms. In this section, we look at generic packages. The syntax is similar to that used for generic subprograms: we start with the `generic` keyword and continue with formal declarations. The only difference is that `package` is specified instead of a subprogram keyword.

Here's an example:

```
generic
   type T is private;
package Element is

   procedure Set (E : T);
   procedure Reset;
   function Get return T;
   function Is_Valid return Boolean;

   Invalid_Element : exception;

private
   Value : T;
   Valid : Boolean := False;
end Element;
```

```
package body Element is

   procedure Set (E : T) is
   begin
      Value := E;
      Valid := True;
   end Set;

   procedure Reset is
   begin
      Valid := False;
   end Reset;

   function Get return T is
   begin
      if not Valid then
         raise Invalid_Element;
      end if;
      return Value;
   end Get;

   function Is_Valid return Boolean is (Valid);
end Element;
```

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

procedure Show_Generic_Package is

   package I is new Element (T => Integer);

   procedure Display_Initialized is
   begin
      if I.Is_Valid then
         Put_Line ("Value is initialized");
      else
         Put_Line ("Value is not initialized");
      end if;
   end Display_Initialized;

begin
   Display_Initialized;

   Put_Line ("Initializing...");
   I.Set (5);
   Display_Initialized;
   Put_Line ("Value is now set to " & Integer'Image (I.Get));

   Put_Line ("Reseting...");
   I.Reset;
   Display_Initialized;

end Show_Generic_Package;
```

In the example above, we created a simple container named `Element`, with just one single element. This container tracks whether the element has been initialized or not.

After writing package definition, we create the instance `I` of the `Element`. We use the instance by calling the package subprograms (`Set`, `Reset`, and `Get`).

10.7 Formal subprograms

In addition to formal types and objects, we can also declare formal subprograms or packages. This course only describes formal subprograms; formal packages are discussed in the advanced course.

We use the `with` keyword to declare a formal subprogram. In the example below, we declare a formal function (`Comparison`) to be used by the generic procedure `Check`.

```
generic
   Description : String;
   type T is private;
   with function Comparison (X, Y : T) return Boolean;
procedure Check (X, Y : T);
```

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

procedure Check (X, Y : T) is
   Result : Boolean;
begin
   Result := Comparison (X, Y);
   if Result then
      Put_Line ("Comparison (" & Description &
                ") between arguments is OK!");
   else
      Put_Line ("Comparison (" & Description &
                ") between arguments is not OK!");
   end if;
end Check;
```

```
with Check;

procedure Show_Formal_Subprogram is

   A, B : Integer;

   procedure Check_Is_Equal is new Check (Description => "equality",
                                          T           => Integer,
                                          Comparison  => Standard."=");
   --  Here, we are mapping the standard equality operator for Integer
   --  types to the Comparison formal function
begin
   A := 0;
   B := 1;
   Check_Is_Equal (A, B);
end Show_Formal_Subprogram;
```

10.8 Example: I/O instances

Ada offers generic I/O packages that can be instatianted for standard and derived types. One example is the generic `Float_IO` package, which provides procedures such as `Put` and `Get`. In fact, `Float_Text_IO` — available from the standard library — is an instance of the `Float_IO` package, and it's defined as:

```
with Ada.Text_IO;

package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO (Float);
```

You can use it directly with any object of floating-point type. For example:

```
with Ada.Float_Text_IO;

procedure Show_Float_Text_IO is
   X : constant Float := 2.5;

   use Ada.Float_Text_IO;
begin
   Put (X);
end Show_Float_Text_IO;
```

Instantiating generic I/O packages can be useful for derived types. For example, let's create a new type `Price` that must be displayed with two decimal digits after the point, and no exponent.

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

procedure Show_Float_IO_Inst is

   type Price is digits 3;

   package Price_IO is new Ada.Text_IO.Float_IO (Price);

   P : Price;
begin
   --  Set to zero => don't display exponent
   Price_IO.Default_Exp  := 0;

   P := 2.5;
   Price_IO.Put (P);
   New_Line;

   P := 5.75;
   Price_IO.Put (P);
   New_Line;
end Show_Float_IO_Inst;
```

By adjusting `Default_Exp` from the `Price_IO` instance to *remove* the exponent, we can control how variables of `Price` type are displayed. Just as a side note, we could also have written:

```
-- [...]

   type Price is new Float;

   package Price_IO is new Ada.Text_IO.Float_IO (Price);

begin
   Price_IO.Default_Aft  := 2;
   Price_IO.Default_Exp  := 0;
```

In this case, we're ajusting `Default_Aft`, too, to get two decimal digits after the point when calling `Put`.

In addition to the generic `Float_IO` package, the following generic packages are available from `Ada.Text_IO`:

- `Enumeration_IO` for enumeration types;
- `Integer_IO` for integer types;
- `Modular_IO` for modular types;
- `Fixed_IO` for fixed-point types;
- `Decimal_IO` for decimal types.

In fact, we could rewrite the example above using decimal types:

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

procedure Show_Decimal_IO_Inst is

   type Price is delta 10.0 ** (-2) digits 12;

   package Price_IO is new Ada.Text_IO.Decimal_IO (Price);

   P : Price;
begin
   Price_IO.Default_Exp  := 0;

   P := 2.5;
   Price_IO.Put (P);
   New_Line;

   P := 5.75;
   Price_IO.Put (P);
   New_Line;
end Show_Decimal_IO_Inst;
```

10.9 Example: ADTs

An important application of generics is to model abstract data types (ADTs). In fact, Ada includes a library with numerous ADTs using generics: `Ada.Containers` (described in the *containers section* (page 139)).

A typical example of an ADT is a stack:

```
generic
   Max : Positive;
   type T is private;
package Stacks is

   type Stack is limited private;

   Stack_Underflow, Stack_Overflow : exception;

   function Is_Empty (S : Stack) return Boolean;

   function Pop (S : in out Stack) return T;

   procedure Push (S : in out Stack; V : T);

private

   type Stack_Array is array (Natural range <>) of T;

   Min : constant := 1;

   type Stack is record
      Container : Stack_Array (Min .. Max);
      Top       : Natural := Min - 1;
   end record;

end Stacks;
```

```
package body Stacks is

   function Is_Empty (S : Stack) return Boolean is
     (S.Top < S.Container'First);

   function Is_Full (S : Stack) return Boolean is
     (S.Top >= S.Container'Last);

   function Pop (S : in out Stack) return T is
   begin
      if Is_Empty (S) then
         raise Stack_Underflow;
      else
         return X : T do
            X     := S.Container (S.Top);
            S.Top := S.Top - 1;
         end return;
      end if;
   end Pop;

   procedure Push (S : in out Stack; V : T) is
   begin
      if Is_Full (S) then
         raise Stack_Overflow;
      else
         S.Top               := S.Top + 1;
         S.Container (S.Top) := V;
      end if;
   end Push;

end Stacks;
```

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

procedure Show_Stack is

   package Integer_Stacks is new Stacks (Max => 10,
                                         T   => Integer);
   use Integer_Stacks;

   Values : Integer_Stacks.Stack;

begin
   Push (Values, 10);
   Push (Values, 20);

   Put_Line ("Last value was " & Integer'Image (Pop (Values)));
end Show_Stack;
```

In this example, we first create a generic stack package (`Stacks`) and then instantiate it to create a stack of up to 10 integer values.

10.10 Example: Swap

Let's look at a simple procedure that swaps variables of type `Color`:

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

(continues on next page)

(continued from previous page)

```
   procedure Swap_Colors (X, Y : in out Color);
end Colors;
```

```
package body Colors is

   procedure Swap_Colors (X, Y : in out Color) is
      Tmp : constant Color := X;
   begin
      X := Y;
      Y := Tmp;
   end Swap_Colors;

end Colors;
```

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

procedure Test_Non_Generic_Swap_Colors is
   A, B, C : Color;
begin
   A := Blue;
   B := White;
   C := Red;

   Put_Line ("Value of A is " & Color'Image (A));
   Put_Line ("Value of B is " & Color'Image (B));
   Put_Line ("Value of C is " & Color'Image (C));

   New_Line;
   Put_Line ("Swapping A and C...");
   New_Line;
   Swap_Colors (A, C);

   Put_Line ("Value of A is " & Color'Image (A));
   Put_Line ("Value of B is " & Color'Image (B));
   Put_Line ("Value of C is " & Color'Image (C));
end Test_Non_Generic_Swap_Colors;
```

In this example, `Swap_Colors` can only be used for the `Color` type. However, this algorithm can theoretically be used for any type, whether an enumeration type or a complex record type with many elements. The algorithm itself is the same: it's only the type that differs. If, for example, we want to swap variables of `Integer` type, we don't want to duplicate the implementation. Therefore, such an algorithm is a perfect candidate for abstraction using generics.

In the example below, we create a generic version of `Swap_Colors` and name it `Generic_Swap`. This generic version can operate on any type due to the declaration of formal type `T`.

```
generic
   type T is private;
procedure Generic_Swap (X, Y : in out T);
```

```
procedure Generic_Swap (X, Y : in out T) is
   Tmp : constant T := X;
begin
   X := Y;
   Y := Tmp;
end Generic_Swap;
```

```
with Generic_Swap;

package Colors is

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

   procedure Swap_Colors is new Generic_Swap (T => Color);

end Colors;
```

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

procedure Test_Swap_Colors is
   A, B, C : Color;
begin
   A := Blue;
   B := White;
   C := Red;

   Put_Line ("Value of A is " & Color'Image (A));
   Put_Line ("Value of B is " & Color'Image (B));
   Put_Line ("Value of C is " & Color'Image (C));

   New_Line;
   Put_Line ("Swapping A and C...");
   New_Line;
   Swap_Colors (A, C);

   Put_Line ("Value of A is " & Color'Image (A));
   Put_Line ("Value of B is " & Color'Image (B));
   Put_Line ("Value of C is " & Color'Image (C));
end Test_Swap_Colors;
```

As we can see in the example, we can create the same `Swap_Colors` procedure as we had in the non-generic version of the algorithm by declaring it as an instance of the generic `Generic_Swap` procedure. We specify that the generic T type will be mapped to the `Color` type by passing it as an argument to the `Generic_Swap` instantiation,

10.11 Example: Reversing

The previous example, with an algorithm to swap two values, is one of the simplest examples of using generics. Next we study an algorithm for reversing elements of an array. First, let's start with a non-generic version of the algorithm, one that works specifically for the `Color` type:

```
package Colors is

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

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

   procedure Reverse_Color_Array (X : in out Color_Array);

end Colors;
```

```
package body Colors is

   procedure Reverse_Color_Array (X : in out Color_Array) is
```

(continues on next page)

(continued from previous page)

```
   begin
      for I in X'First .. (X'Last + X'First) / 2 loop
         declare
            Tmp     : Color;
            X_Left  : Color renames X (I);
            X_Right : Color renames X (X'Last + X'First - I);
         begin
            Tmp     := X_Left;
            X_Left  := X_Right;
            X_Right := Tmp;
         end;
      end loop;
   end Reverse_Color_Array;

end Colors;
```

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

procedure Test_Non_Generic_Reverse_Colors is

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

begin
   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

   New_Line;
   Put_Line ("Reversing My_Color...");
   New_Line;
   Reverse_Color_Array (My_Colors);

   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

end Test_Non_Generic_Reverse_Colors;
```

The procedure `Reverse_Color_Array` takes an array of colors, starts by swapping the first and last elements of the array, and continues doing that with successive elements until it reaches the middle of array. At that point, the entire array has been reversed, as we see from the output of the test program.

To abstract this procedure, we declare formal types for three components of the algorithm:

- the elements of the array (`Color` type in the example)
- the range used for the array (`Integer` range in the example)
- the actual array type (`Color_Array` type in the example)

This is a generic version of the algorithm:

```
generic
   type T is private;
   type Index is range <>;
   type Array_T is array (Index range <>) of T;
procedure Generic_Reverse_Array (X : in out Array_T);
```

```
procedure Generic_Reverse_Array (X : in out Array_T) is
begin
```

(continues on next page)

(continued from previous page)

```
   for I in X'First .. (X'Last + X'First) / 2 loop
      declare
         Tmp     : T;
         X_Left  : T renames X (I);
         X_Right : T renames X (X'Last + X'First - I);
      begin
         Tmp     := X_Left;
         X_Left  := X_Right;
         X_Right := Tmp;
      end;
   end loop;
end Generic_Reverse_Array;
```

```
with Generic_Reverse_Array;

package Colors is

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

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

   procedure Reverse_Color_Array is new Generic_Reverse_Array
     (T => Color, Index => Integer, Array_T => Color_Array);

end Colors;
```

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

procedure Test_Reverse_Colors is

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

begin
   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

   New_Line;
   Put_Line ("Reversing My_Color...");
   New_Line;
   Reverse_Color_Array (My_Colors);

   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

end Test_Reverse_Colors;
```

As mentioned above, we're abstracting three components of the algorithm:

- the `T` type abstracts the elements of the array
- the `Index` type abstracts the range used for the array
- the `Array_T` type abstracts the array type and uses the formal declarations of the `T` and `Index` types.

10.12 Example: Test application

In the previous example we've focused only on abstracting the reversing algorithm itself. However, we could have decided to also abstract our small test application. This could be useful if we, for example, decide to test other procedures that change elements of an array.

In order to do this, we again have to choose the elements to abstract. We therefore declare the following formal parameters:

- S: the string containing the array name
- a function `Image` that converts an element of type T to a string
- a procedure `Test` that performs some operation on the array

Note that `Image` and `Test` are examples of formal subprograms and S is an example of a formal object.

Here is a version of the test application making use of the generic `Perform_Test` procedure:

```
generic
   type T is private;
   type Index is range <>;
   type Array_T is array (Index range <>) of T;
procedure Generic_Reverse_Array (X : in out Array_T);
```

```
procedure Generic_Reverse_Array (X : in out Array_T) is
begin
   for I in X'First .. (X'Last + X'First) / 2 loop
      declare
         Tmp     : T;
         X_Left  : T renames X (I);
         X_Right : T renames X (X'Last + X'First - I);
      begin
         Tmp     := X_Left;
         X_Left  := X_Right;
         X_Right := Tmp;
      end;
   end loop;
end Generic_Reverse_Array;
```

```
generic
   type T is private;
   type Index is range <>;
   type Array_T is array (Index range <>) of T;
   S : String;
   with function Image (E : T) return String is <>;
   with procedure Test (X : in out Array_T);
procedure Perform_Test (X : in out Array_T);
```

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

procedure Perform_Test (X : in out Array_T) is
begin
   for C of X loop
      Put_Line (S & ": " & Image (C));
   end loop;

   New_Line;
   Put_Line ("Testing " & S & "...");
   New_Line;
   Test (X);
```

(continues on next page)

(continued from previous page)

```
   for C of X loop
      Put_Line (S & ": " & Image (C));
   end loop;
end Perform_Test;
```

```
with Generic_Reverse_Array;

package Colors is

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

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

   procedure Reverse_Color_Array is new Generic_Reverse_Array
     (T => Color, Index => Integer, Array_T => Color_Array);

end Colors;
```

```
with Colors;       use Colors;
with Perform_Test;

procedure Test_Reverse_Colors is

   procedure Perform_Test_Reverse_Color_Array is new
     Perform_Test (T       => Color,
                   Index   => Integer,
                   Array_T => Color_Array,
                   S       => "My_Color",
                   Image   => Color'Image,
                   Test    => Reverse_Color_Array);

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

begin
   Perform_Test_Reverse_Color_Array (My_Colors);
end Test_Reverse_Colors;
```

In this example, we create the procedure `Perform_Test_Reverse_Color_Array` as an instance of the generic procedure (`Perform_Test`). Note that:

- For the formal `Image` function, we use the `'Image` attribute of the `Color` type
- For the formal `Test` procedure, we reference the `Reverse_Array` procedure from the package.

CHAPTER

ELEVEN

EXCEPTIONS

Ada uses exceptions for error handling. Unlike many other languages, Ada speaks about *raising*, not *throwing*, an exception and *handling*, not *catching*, an exception.

11.1 Exception declaration

Ada exceptions are not types, but instead objects, which may be peculiar to you if you're used to the way Java or Python support exceptions. Here's how you declare an exception:

```
package Exceptions is
    My_Except : exception;
    --  Like an object. *NOT* a type !
end Exceptions;
```

Even though they're objects, you're going to use each declared exception object as a "kind" or "family" of exceptions. Ada does not require that a subprogram declare every exception it can potentially raise.

11.2 Raising an exception

To raise an exception of our newly declared exception kind, do the following:

```
with Exceptions; use Exceptions;

procedure Main is
begin
   raise My_Except;
   --  Execution of current control flow abandoned; an exception of kind
   --  "My_Except" will bubble up until it is caught.

   raise My_Except with "My exception message";
   --  Execution of current control flow abandoned; an exception of
   --  kind "My_Except" with associated string will bubble up until
   --  it is caught.
end Main;
```

11.3 Handling an exception

Next, we address how to handle exceptions that were raised by us or libraries that we call. The neat thing in Ada is that you can add an exception handler to any statement block as follows:

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

procedure Open_File is
   File : File_Type;
begin
   --  Block (sequence of statements)
   begin
      Open (File, In_File, "input.txt");
   exception
      when E : Name_Error =>
      --       ^ Exception to be handled
         Put ("Cannot open input file : ");
         Put_Line (Exception_Message (E));
         raise;
         --  Reraise current occurence
   end;
end Open_File;
```

In the example above, we're using the `Exception_Message` function from the `Ada.Exceptions` package. This function returns the message associated with the exception as a string.

You don't need to introduce a block just to handle an exception: you can add it to the statements block of your current subprogram:

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

procedure Open_File is
   File : File_Type;
begin
   Open (File, In_File, "input.txt");
--  Exception block can be added to any block
exception
   when Name_Error =>
      Put ("Cannot open input file");
end Open_File;
```

Attention: Exception handlers have an important restriction that you need to be careful about: Exceptions raised in the declarative section are not caught by the handlers of that block. So for example, in the following code, the exception will not be caught.

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

procedure Be_Careful is
   function Dangerous return Integer is
   begin
      raise Constraint_Error;
      return 42;
   end Dangerous;

begin
   declare
      A : Integer := Dangerous;
   begin
      Put_Line (Integer'Image (A));
   exception
      when Constraint_Error => Put_Line ("error!");
   end;
end Be_Careful;
```

This is also the case for the top-level exception block that is part of the current subprogram.

11.4 Predefined exceptions

Ada has a very small number of predefined exceptions:

- `Constraint_Error` is the main one you might see. It's raised:
  - When bounds don't match or, in general, any violation of constraints.
  - In case of overflow
  - In case of null dereferences
  - In case of division by 0
- `Program_Error` might appear, but probably less often. It's raised in more arcane situations, such as for order of elaboration issues and some cases of detectable erroneous execution.
- `Storage_Error` will happen because of memory issues, such as:
  - Not enough memory (allocator)
  - Not enough stack
- `Tasking_Error` will happen with task related errors, such as any error happening during task activation.

You should not reuse predefined exceptions. If you do then, it won't be obvious when one is raised that it is because something went wrong in a built-in language operation.

CHAPTER

TWELVE

TASKING

Tasks and protected objects allow the implementation of concurrency in Ada. The following sections explain these concepts in more details.

12.1 Tasks

A task can be thought as an application that runs *concurrently* with the main application. In other programming languages, a task can be called a thread[17], and tasking can be called multithreading[18].

Tasks may synchronize with the main application but may also process information completely independent from the main application. Here we show how this is accomplished.

12.1.1 Simple task

Tasks are declared using the keyword `task`. The task implementation is specified in a `task body` block. For example:

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

procedure Show_Simple_Task is
   task T;

   task body T is
   begin
      Put_Line ("In task T");
   end T;
begin
   Put_Line ("In main");
end Show_Simple_Task;
```

Here, we're declaring and implementing the task T. As soon as the main application starts, task T starts automatically — it's not necessary to manually start this task. By running the application above, we can see that both calls to `Put_Line` are performed.

Note that:

- The main application is itself a task (the main task).
  - In this example, the subprogram `Show_Simple_Task` is the main task of the application.
- Task T is a subtask.
  - Each subtask has a master task.
  - Therefore the main task is also the master task of task T.

[17] https://en.wikipedia.org/wiki/Thread_(computing)
[18] https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading

- The number of tasks is not limited to one: we could include a task T2 in the example above.
  - This task also starts automatically and runs *concurrently* with both task T and the main task. For example:

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

procedure Show_Simple_Tasks is
   task T;
   task T2;

   task body T is
   begin
      Put_Line ("In task T");
   end T;

   task body T2 is
   begin
      Put_Line ("In task T2");
   end T2;

begin
   Put_Line ("In main");
end Show_Simple_Tasks;
```

12.1.2 Simple synchronization

As we've just seen, as soon as the main task starts, its subtasks also start automatically. The main task continues its processing until it has nothing more to do. At that point, however, it will not terminate. Instead, the task waits until its subtasks have finished before it allows itself to terminate. In other words, this waiting process provides synchronization between the main task and its subtasks. After this synchronization, the main task will terminate. For example:

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

procedure Show_Simple_Sync is
   task T;
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end T;
begin
   null;
   --  Will wait here until all tasks have terminated
end Show_Simple_Sync;
```

The same mechanism is used for other subprograms that contain subtasks: the subprogram's master task will wait for its subtasks to finish. So this mechanism is not limited to the main application and also applies to any subprogram called by the main application or its subprograms.

Synchronization also occurs if we move the task to a separate package. In the example below, we declare a task T in the package `Simple_Sync_Pkg`.

```
package Simple_Sync_Pkg is
   task T;
end Simple_Sync_Pkg;
```

This is the corresponding package body:

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

package body Simple_Sync_Pkg is
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end T;
end Simple_Sync_Pkg;
```

Because the package is `with`'ed by the main procedure, the task T defined in the package is part of the main task. For example:

```
with Simple_Sync_Pkg;

procedure Test_Simple_Sync_Pkg is
begin
   null;
   --  Will wait here until all tasks have terminated
end Test_Simple_Sync_Pkg;
```

Again, as soon as the main task reaches its end, it synchronizes with task T from `Simple_Sync_Pkg` before terminating.

12.1.3 Delay

We can introduce a delay by using the keyword `delay`. This puts the task to sleep for the length of time (in seconds) specified in the delay statement. For example:

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

procedure Show_Delay is

   task T;

   task body T is
   begin
      for I in 1 .. 5 loop
         Put_Line ("hello from task T");
         delay 1.0;
         --    ^ Wait 1.0 seconds
      end loop;
   end T;
begin
   delay 1.5;
   Put_Line ("hello from main");
end Show_Delay;
```

In this example, we're making the task T wait one second after each time it displays the "hello" message. In addition, the main task is waiting 1.5 seconds before displaying its own "hello" message

12.1.4 Synchronization: rendez-vous

The only type of synchronization we've seen so far is the one that happens automatically at the end of the main task. You can also define custom synchronization points using the keyword `entry`. An *entry* can be viewed as a special kind of subprogram, which is called by the master task using a similar syntax, as we will see later.

In the task definition, you define which part of the task will accept the entries by using the keyword `accept`. A task proceeds until it reaches an `accept` statement and then waits for the master task to synchronize with it. Specifically,

- The subtask waits at that point (in the `accept` statement), ready to accept a call to the corresponding entry from the master task.
- The master task calls the task entry, in a manner similar to a procedure call, to synchronize with the subtask.

This synchronization between tasks is called *rendez-vous*. Let's see an example:

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

procedure Show_Rendezvous is

   task T is
      entry Start;
   end T;

   task body T is
   begin
      accept Start; -- Waiting for somebody to call the entry
      Put_Line ("In T");
   end T;

begin
   Put_Line ("In Main");
   T.Start; --  Calling T's entry
end Show_Rendezvous;
```

In this example, we declare an entry `Start` for task T. In the task body, we implement this entry using `accept Start`. When task T reaches this point, it waits for the master task. This synchronization occurs in the `T.Start` statement. After the synchronization completes, the main task and task T again run concurrently until they synchronize one final time when the main task finishes.

An entry may be used to perform more than a simple task synchronization: it also may perform multiple statements during the time both tasks are synchronized. We do this with a `do ... end` block. For the previous example, we would simply write `accept Start do <statements>; end;`. We use this kind of block in the next example.

12.1.5 Select loop

There's no limit to the number of times an entry can be accepted. We could even create an infinite loop in the task and accept calls to the same entry over and over again. An infinite loop, however, prevents the subtask from finishing, so it blocks the master task when it reaches the end of its processing. Therefore, a loop containing `accept` statements in a task body is normally used in conjunction with a `select ... or terminate` statement. In simple terms, this statement allows the master task to automatically terminate the subtask when the master task finishes. For example:

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

procedure Show_Rendezvous_Loop is

   task T is
      entry Reset;
      entry Increment;
   end T;

   task body T is
      Cnt : Integer := 0;
```

(continues on next page)

(continued from previous page)

```
   begin
      loop
         select
            accept Reset do
               Cnt := 0;
            end Reset;
            Put_Line ("Reset");
         or
            accept Increment do
               Cnt := Cnt + 1;
            end Increment;
            Put_Line ("In T's loop (" & Integer'Image (Cnt) & ")");
         or
            terminate;
         end select;
      end loop;
   end T;

begin
   Put_Line ("In Main");

   for I in 1 .. 4 loop
      T.Increment; --  Calling T's entry multiple times
   end loop;

   T.Reset;
   for I in 1 .. 4 loop
      T.Increment; --  Calling T's entry multiple times
   end loop;

end Show_Rendezvous_Loop;
```

In this example, the task body implements an infinite loop that accepts calls to the `Reset` and `Increment` entry. We make the following observations:

- The `accept E do ... end` block is used to increment a counter.
  - As long as task T is performing the `do ... end` block, the main task waits for the block to complete.
- The main task is calling the `Increment` entry multiple times in the loop from `1 .. 4`. It is also calling the `Reset` entry before and the loop.
  - Because task T contains an infinite loop, it always accepts calls to the `Reset` and `Increment` entries.
  - When the main task finishes, it checks the status of the T task. Even though task T could accept new calls to the `Reset` or `Increment` entries, the master task is allowed to terminate task T due to the `or terminate` part of the `select` statement.

12.1.6 Cycling tasks

In a previous example, we saw how to delay a task a specified time by using the `delay` keyword. However, using delay statements in a loop is not enough to guarantee regular intervals between those delay statements. For example, we may have a call to a computationally intensive procedure between executions of successive delay statements:

```
while True loop
   delay 1.0;
   --    ^ Wait 1.0 seconds
```

(continues on next page)

(continued from previous page)

```
   Computational_Intensive_App;
end loop;
```

In this case, we can't guarantee that exactly 10 seconds have elapsed after 10 calls to the delay statement because a time drift may be introduced by the `Computational_Intensive_App` procedure. In many cases, this time drift is not relevant, so using the `delay` keyword is good enough.

However, there are situations where a time drift isn't acceptable. In those cases, we need to use the `delay until` statement, which accepts a precise time for the end of the delay, allowing us to define a regular interval. This is useful, for example, in real-time applications.

We will soon see an example of how this time drift may be introduced and how the `delay until` statement circumvents the problem. But before we do that, we look at a package containing a procedure allowing us to measure the elapsed time (`Show_Elapsed_Time`) and a dummy `Computational_Intensive_App` procedure which is simulated by using a simple delay. This is the package specification:

```
with Ada.Real_Time; use Ada.Real_Time;

package Delay_Aux_Pkg is

   function Get_Start_Time return Time
     with Inline;

   procedure Show_Elapsed_Time
     with Inline;

   procedure Computational_Intensive_App;
private
   Start_Time   : Time := Clock;

   function Get_Start_Time return Time is (Start_Time);

end Delay_Aux_Pkg;
```

And this is the package body:

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

package body Delay_Aux_Pkg is

   procedure Show_Elapsed_Time is
      Now_Time     : Time;
      Elapsed_Time : Time_Span;
   begin
      Now_Time     := Clock;
      Elapsed_Time := Now_Time - Start_Time;
      Put_Line ("Elapsed time "
                & Duration'Image (To_Duration (Elapsed_Time))
                & " seconds");
   end Show_Elapsed_Time;

   procedure Computational_Intensive_App is
   begin
      delay 0.5;
   end Computational_Intensive_App;

end Delay_Aux_Pkg;
```

Using this auxiliary package, we're now ready to write our time-drifting application:

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

with Delay_Aux_Pkg;

procedure Show_Time_Drifting_Task is
   package Aux renames Delay_Aux_Pkg;

   task T;

   task body T is
      Cnt   : Integer := 1;
   begin
      for I in 1 .. 5 loop
         delay 1.0;

         Aux.Show_Elapsed_Time;
         Aux.Computational_Intensive_App;

         Put_Line ("Cycle # " & Integer'Image (Cnt));
         Cnt  := Cnt + 1;
      end loop;
      Put_Line ("Finished time-drifting loop");
   end T;

begin
   null;
end Show_Time_Drifting_Task;
```

We can see by running the application that we already have a time difference of about four seconds after three iterations of the loop due to the drift introduced by `Computational_Intensive_App`. Using the `delay until` statement, however, we're able to avoid this time drift and have a regular interval of exactly one second:

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

with Delay_Aux_Pkg;

procedure Show_Cycling_Task is
   package Aux renames Delay_Aux_Pkg;

   task T;

   task body T is
      Cycle : constant Time_Span := Milliseconds (1000);
      Next  : Time := Aux.Get_Start_Time + Cycle;

      Cnt   : Integer := 1;
   begin
      for I in 1 .. 5 loop
         delay until Next;

         Aux.Show_Elapsed_Time;
         Aux.Computational_Intensive_App;

         --  Calculate next execution time using a
         --  cycle of one second
         Next := Next + Cycle;

         Put_Line ("Cycle # " & Integer'Image (Cnt));
         Cnt  := Cnt + 1;
```

(continues on next page)

(continued from previous page)

```
      end loop;
      Put_Line ("Finished cycling");
   end T;

begin
   null;
end Show_Cycling_Task;
```

Now, as we can see by running the application, the `delay until` statement ensures that the `Computational_Intensive_App` doesn't disturb the regular interval of one second between iterations.

12.2 Protected objects

When multiple tasks are accessing shared data, corruption of that data may occur. For example, data may be inconsistent if one task overwrites parts of the information that's being read by another task at the same time. In order to avoid these kinds of problems and ensure information is accessed in a coordinated way, we use *protected objects*.

Protected objects encapsulate data and provide access to that data by means of *protected operations*, which may be subprograms or protected entries. Using protected objects ensures that data is not corrupted by race conditions or other simultaneous access.

Important

Protected objects can be implemented using Ada tasks. In fact, this was the *only* possible way of implementing them in Ada 83 (the first version of the Ada language). However, the use of protected objects is much simpler than using similar mechanisms implemented using only tasks. Therefore, you should use protected objects when your main goal is only to protect data.

12.2.1 Simple object

You declare a protected object with the `protected` keyword. The syntax is similar to that used for packages: you can declare operations (e.g., procedures and functions) in the public part and data in the private part. The corresponding implementation of the operations is included in the `protected body` of the object. For example:

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

procedure Show_Protected_Objects is

   protected Obj is
      --  Operations go here (only subprograms)
      procedure Set (V : Integer);
      function Get return Integer;
   private
      --  Data goes here
      Local : Integer := 0;
   end Obj;

   protected body Obj is
      --  procedures can modify the data
      procedure Set (V : Integer) is
      begin
```

(continues on next page)

(continued from previous page)

```
         Local := V;
      end Set;

      --  functions cannot modify the data
      function Get return Integer is
      begin
         return Local;
      end Get;
   end Obj;

begin
   Obj.Set (5);
   Put_Line ("Number is: " & Integer'Image (Obj.Get));
end Show_Protected_Objects;
```

In this example, we define two operations for `Obj`: `Set` and `Get`. The implementation of these operations is in the `Obj` body. The syntax used for writing these operations is the same as that for normal procedures and functions. The implementation of protected objects is straightforward — we simply access and update `Local` in these subprograms. To call these operations in the main application, we use prefixed notation, e.g., `Obj.Get`.

12.2.2 Entries

In addition to protected procedures and functions, you can also define protected entry points. Do this using the `entry` keyword. Protected entry points allow you to define barriers using the `when` keyword. Barriers are conditions that must be fulfilled before the entry can start performing its actual processing — we speak of *releasing* the barrier when the condition is fulfilled.

The previous example used procedures and functions to define operations on the protected objects. However, doing so permits reading protected information (via `Obj.Get`) before it's set (via `Obj.Set`). To allow that to be a defined operation, we specified a default value (0). Instead, by rewriting `Obj.Get` using an *entry* instead of a function, we implement a barrier, ensuring no task can read the information before it's been set.

The following example implements the barrier for the `Obj.Get` operation. It also contains two concurrent subprograms (main task and task T) that try to access the protected object.

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

procedure Show_Protected_Objects_Entries is

   protected Obj is
      procedure Set (V : Integer);
      entry Get (V : out Integer);
   private
      Local  : Integer;
      Is_Set : Boolean := False;
   end Obj;

   protected body Obj is
      procedure Set (V : Integer) is
      begin
         Local := V;
         Is_Set := True;
      end Set;

      entry Get (V : out Integer)
        when Is_Set is
         --  Entry is blocked until the condition is true.
```

(continues on next page)

(continued from previous page)

```
         --  The barrier is evaluated at call of entries and at exits of
         --  procedures and entries.
         --  The calling task sleeps until the barrier is released.
      begin
         V := Local;
         Is_Set := False;
      end Get;
   end Obj;

   N : Integer := 0;

   task T;

   task body T is
   begin
      Put_Line ("Task T will delay for 4 seconds...");
      delay 4.0;
      Put_Line ("Task T will set Obj...");
      Obj.Set (5);
      Put_Line ("Task T has just set Obj...");
   end T;
begin
   Put_Line ("Main application will get Obj...");
   Obj.Get (N);
   Put_Line ("Main application has just retrieved Obj...");
   Put_Line ("Number is: " & Integer'Image (N));

end Show_Protected_Objects_Entries;
```

As we see by running it, the main application waits until the protected object is set (by the call to `Obj.Set` in task `T`) before it reads the information (via `Obj.Get`). Because a 4-second delay has been added in task `T`, the main application is also delayed by 4 seconds. Only after this delay does task `T` set the object and release the barrier in `Obj.Get` so that the main application can then resume processing (after the information is retrieved from the protected object).

12.3 Task and protected types

In the previous examples, we defined single tasks and protected objects. We can, however, generalize tasks and protected objects using type definitions. This allows us, for example, to create multiple tasks based on just a single task type.

12.3.1 Task types

A task type is a generalization of a task. The declaration is similar to simple tasks: you replace `task` with `task type`. The difference between simple tasks and task types is that task types don't create actual tasks that automatically start. Instead, a task declaration is needed. This is exactly the way normal variables and types work: objects are only created by variable definitions, not type definitions.

To illustrate this, we repeat our first example:

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

procedure Show_Simple_Task is
   task T;
```

(continues on next page)

(continued from previous page)

```
   task body T is
   begin
      Put_Line ("In task T");
   end T;
begin
   Put_Line ("In main");
end Show_Simple_Task;
```

We now rewrite it by replacing `task T` with `task type TT`. We declare a task (`A_Task`) based on the task type `TT` after its definition:

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

procedure Show_Simple_Task_Type is
   task type TT;

   task body TT is
   begin
      Put_Line ("In task type TT");
   end TT;

   A_Task : TT;
begin
   Put_Line ("In main");
end Show_Simple_Task_Type;
```

We can extend this example and create an array of tasks. Since we're using the same syntax as for variable declarations, we use a similar syntax for task types: `array (<>) of Task_Type`. Also, we can pass information to the individual tasks by defining a `Start` entry. Here's the updated example:

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

procedure Show_Task_Type_Array is
   task type TT is
      entry Start (N : Integer);
   end TT;

   task body TT is
      Task_N : Integer;
   begin
      accept Start (N : Integer) do
         Task_N := N;
      end Start;
      Put_Line ("In task T: " & Integer'Image (Task_N));
   end TT;

   My_Tasks : array (1 .. 5) of TT;
begin
   Put_Line ("In main");

   for I in My_Tasks'Range loop
      My_Tasks (I).Start (I);
   end loop;
end Show_Task_Type_Array;
```

In this example, we're declaring five tasks in the array `My_Tasks`. We pass the array index to the individual tasks in the entry point (`Start`). After the synchronization between the individual subtasks and the main task, each subtask calls `Put_Line` concurrently.

12.3.2 Protected types

A protected type is a generalization of a protected object. The declaration is similar to that for protected objects: you replace `protected` with `protected type`. Like task types, protected types require an object declaration to create actual objects. Again, this is similar to variable declarations and allows for creating arrays (or other composite objects) of protected objects.

We can reuse a previous example and rewrite it to use a protected type:

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

procedure Show_Protected_Object_Type is

   protected type Obj_Type is
      procedure Set (V : Integer);
      function Get return Integer;
   private
      Local : Integer := 0;
   end Obj_Type;

   protected body Obj_Type is
      procedure Set (V : Integer) is
      begin
         Local := V;
      end Set;

      function Get return Integer is
      begin
         return Local;
      end Get;
   end Obj_Type;

   Obj : Obj_Type;
begin
   Obj.Set (5);
   Put_Line ("Number is: " & Integer'Image (Obj.Get));
end Show_Protected_Object_Type;
```

In this example, instead of directly defining the protected object `Obj`, we first define a protected type `Obj_Type` and then declare `Obj` as an object of that protected type. Note that the main application hasn't changed: we still use `Obj.Set` and `Obj.Get` to access the protected object, just like in the original example.

CHAPTER

THIRTEEN

DESIGN BY CONTRACTS

Contracts are used in programming to codify expectations. Parameter modes of a subprogram can be viewed as a simple form of contracts. When the specification of subprogram `Op` declares a parameter using `in` mode, the caller of `Op` knows that the `in` argument won't be changed by `Op`. In other words, the caller expects that `Op` doesn't modify the argument it's providing, but just reads the information stored in the argument. Constraints and subtypes are other examples of contracts. In general, these specifications improve the consistency of the application.

Design-by-contract programming refers to techniques that include pre- and postconditions, subtype predicates, and type invariants. We study those topics in this chapter.

13.1 Pre- and postconditions

Pre- and postconditions provide expectations regarding input and output parameters of subprograms and return value of functions. If we say that certain requirements must be met before calling a subprogram `Op`, those are preconditions. Similarly, if certain requirements must be met after a call to the subprogram `Op`, those are postconditions. We can think of preconditions and postconditions as promises between the subprogram caller and the callee: a precondition is a promise from the caller to the callee, and a postcondition is a promise in the other direction.

Pre- and postconditions are specified using an aspect clause in the subprogram declaration. A `with Pre => <condition>` clause specifies a precondition and a `with Post => <condition>` clause specifies a postcondition.

The following code shows an example of preconditions:

```
procedure Show_Simple_Precondition is

   procedure DB_Entry (Name : String; Age  : Natural)
     with Pre => Name'Length > 0
   is
   begin
      --  Missing implementation
      null;
   end DB_Entry;
begin
   DB_Entry ("John", 30);
   DB_Entry ("",     21);  --  Precondition will fail!
end Show_Simple_Precondition;
```

In this example, we want to prevent the name field in our database from containing an empty string. We implement this requirement by using a precondition requiring that the length of the string used for the `Name` parameter of the `DB_Entry` procedure is greater than zero. If the `DB_Entry` procedure is called with an empty string for the `Name` parameter, the call will fail because the precondition is not met.

In the GNAT toolchain

GNAT handles pre- and postconditions by generating runtime assertions for them. By default, however, assertions aren't enabled. Therefore, in order to check pre- and postconditions at runtime, you need to enable assertions by using the *-gnata* switch.

Before we get to our next example, let's briefly discuss quantified expressions, which are quite useful in concisely writing pre- and postconditions. Quantified expressions return a Boolean value indicating whether elements of an array or container match the expected condition. They have the form: `(for all I in A'Range => <condition on A(I)>`, where `A` is an array and `I` is an index. Quantified expressions using `for all` check whether the condition is true for every element. For example:

```
(for all I in A'Range => A (I) = 0)
```

This quantified expression is only true when all elements of the array `A` have a value of zero.

Another kind of quantified expressions uses `for some`. The form looks similar: `(for some I in A'Range => <condition on A(I)>`. However, in this case the qualified expression tests whether the condition is true only on *some* elements (hence the name) instead of all elements.

We illustrate postconditions using the following example:

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

procedure Show_Simple_Postcondition is

   type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;

   type Int_8_Array is array (Integer range <>) of Int_8;

   function Square (A : Int_8) return Int_8 is
     (A * A)
     with Post => (if abs A in 0 | 1 then Square'Result = abs A
                   else Square'Result > A);

   procedure Square (A : in out Int_8_Array)
     with Post => (for all I in A'Range =>
                     A (I) = A'Old (I) * A'Old (I))
   is
   begin
      for V of A loop
         V := Square (V);
      end loop;
   end Square;

   V : Int_8_Array := (-2, -1, 0, 1, 10, 11);
begin
   for E of V loop
      Put_Line ("Original: " & Int_8'Image (E));
   end loop;
   New_Line;

   Square (V);
   for E of V loop
      Put_Line ("Square:   " & Int_8'Image (E));
   end loop;
end Show_Simple_Postcondition;
```

We declare a signed 8-bit type `Int_8` and an array of that type (`Int_8_Array`). We want to ensure each element of the array is squared after calling the procedure `Square` for an object of the

`Int_8_Array` type. We do this with a postcondition using a `for all` expression. This postcondition also uses the `'Old` attribute to refer to the original value of the parameter (before the call).

We also want to ensure that the result of calls to the `Square` function for the `Int_8` type are greater than the input to that call. To do that, we write a postcondition using the `'Result` attribute of the function and comparing it to the input value.

We can use both pre- and postconditions in the declaration of a single subprogram. For example:

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

procedure Show_Simple_Contract is

   type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;

   function Square (A : Int_8) return Int_8 is
     (A * A)
     with
          Pre  => (Integer'Size >= Int_8'Size * 2 and
                   Integer (A) * Integer (A) < Integer (Int_8'Last)),
          Post => (if abs A in 0 | 1 then Square'Result = abs A
                   else Square'Result > A);

   V : Int_8;
begin
   V := Square (11);
   Put_Line ("Square of 11 is " & Int_8'Image (V));

   V := Square (12);   --  Precondition will fail...
   Put_Line ("Square of 12 is " & Int_8'Image (V));
end Show_Simple_Contract;
```

In this example, we want to ensure that the input value of calls to the `Square` function for the `Int_8` type won't cause overflow in that function. We do this by converting the input value to the `Integer` type, which is used for the temporary calculation, and check if the result is in the appropriate range for the `Int_8` type. We have the same postcondition in this example as in the previous one.

13.2 Predicates

Predicates specify expectations regarding types. They're similar to pre- and postconditions, but apply to types instead of subprograms. Their conditions are checked for each object of a given type, which allows verifying that an object of type T is conformant to the requirements of its type.

There are two kinds of predicates: static and dynamic. In simple terms, static predicates are used to check objects at compile-time, while dynamic predicates are used for checks at run time. Normally, static predicates are used for scalar types and dynamic predicates for the more complex types.

Static and dynamic predicates are specified using the following clauses, respectively:

- `with Static_Predicate => <property>`
- `with Dynamic_Predicate => <property>`

Let's use the following example to illustrate dynamic predicates:

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Calendar;          use Ada.Calendar;
with Ada.Containers.Vectors;

procedure Show_Dynamic_Predicate_Courses is
```

(continues on next page)

(continued from previous page)

```
   package Courses is
      type Course_Container is private;

      type Course is record
         Name       : Unbounded_String;
         Start_Date : Time;
         End_Date   : Time;
      end record
        with Dynamic_Predicate => Course.Start_Date <= Course.End_Date;

      procedure Add (CC : in out Course_Container; C : Course);
   private
      package Course_Vectors is new Ada.Containers.Vectors
        (Index_Type   => Natural,
         Element_Type => Course);

      type Course_Container is record
         V : Course_Vectors.Vector;
      end record;
   end Courses;

   package body Courses is
      procedure Add (CC : in out Course_Container; C : Course) is
      begin
         CC.V.Append (C);
      end Add;
   end Courses;

   use Courses;

   CC : Course_Container;
begin
   Add (CC,
        Course'(
          Name       => To_Unbounded_String ("Intro to Photography"),
          Start_Date => Time_Of (2018, 5, 1),
          End_Date   => Time_Of (2018, 5, 10)));

   --  This should trigger an error in the dynamic predicate check
   Add (CC,
        Course'(
          Name       => To_Unbounded_String ("Intro to Video Recording"),
          Start_Date => Time_Of (2019, 5, 1),
          End_Date   => Time_Of (2018, 5, 10)));

end Show_Dynamic_Predicate_Courses;
```

In this example, the package `Courses` defines a type `Course` and a type `Course_Container`, an object of which contains all courses. We want to ensure that the dates of each course are consistent, specifically that the start date is no later than the end date. To enforce this rule, we declare a dynamic predicate for the `Course` type that performs the check for each object. The predicate uses the type name where a variable of that type would normally be used: this is a reference to the instance of the object being tested.

Note that the example above makes use of unbounded strings and dates. Both types are available in Ada's standard library. Please refer to the following sections for more information about:

- the unbounded string type (`Unbounded_String`): *Unbounded Strings* (page 170) section;
- dates and times: *Dates & Times* (page 159) section.

Static predicates, as mentioned above, are mostly used for scalar types and checked during compilation. They're particularly useful for representing non-contiguous elements of an enumeration.

A classic example is a list of week days:

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

We can easily create a sub-list of work days in the week by specifying a `subtype` with a range based on `Week`. For example:

```
subtype Work_Week is Week range Mon .. Fri;
```

Ranges in Ada can only be specified as contiguous lists: they don't allow us to pick specific days. However, we may want to create a list containing just the first, middle and last day of the work week. To do that, we use a static predicate:

```
subtype Check_Days is Work_Week
  with Static_Predicate => Check_Days in Mon | Wed | Fri;
```

Let's look at a complete example:

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

procedure Show_Predicates is

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

   subtype Work_Week is Week range Mon .. Fri;

   subtype Test_Days is Work_Week
     with Static_Predicate => Test_Days in Mon | Wed | Fri;

   type Tests_Week is array (Week) of Natural
     with Dynamic_Predicate =>
       (for all I in Tests_Week'Range =>
          (case I is
               when Test_Days => Tests_Week (I) > 0,
               when others    => Tests_Week (I) = 0));

   Num_Tests : Tests_Week :=
                 (Mon => 3, Tue => 0,
                  Wed => 4, Thu => 0,
                  Fri => 2, Sat => 0, Sun => 0);

   procedure Display_Tests (N : Tests_Week) is
   begin
      for I in Test_Days loop
         Put_Line ("# tests on " & Test_Days'Image (I)
                   & " => "      & Integer'Image (N (I)));
      end loop;
   end Display_Tests;

begin
   Display_Tests (Num_Tests);

   --  Assigning non-conformant values to individual elements of
   --  the Tests_Week type does not trigger a predicate check:
   Num_Tests (Tue) := 2;

   --  However, assignments with the "complete" Tests_Week type
   --  trigger a predicate check. For example:
   --
   --  Num_Tests := (others => 0);

   --  Also, calling any subprogram with parameters of Tests_Week
```

(continues on next page)

(continued from previous page)

```
   --  type triggers a predicate check.
   --  Therefore, the following line will fail:
   Display_Tests (Num_Tests);
end Show_Predicates;
```

Here we have an application that wants to perform tests only on three days of the work week. These days are specified in the `Test_Days` subtype. We want to track the number of tests that occur each day. We declare the type `Tests_Week` as an array, an object of which will contain the number of tests done each day. According to our requirements, these tests should happen only in the aforementioned three days; on other days, no tests should be performed. This requirement is implemented with a dynamic predicate of the type `Tests_Week`. Finally, the actual information about these tests is stored in the array `Num_Tests`, which is an instance of the `Tests_Week` type.

The dynamic predicate of the `Tests_Week` type is verified during the initialization of `Num_Tests`. If we have a non-conformant value there, the check will fail. However, as we can see in our example, individual assignments to elements of the array do not trigger a check. We can't check for consistency at this point because the initialization of the a complex data structure (such as arrays or records) may not be performed with a single assignment. However, as soon as the object is passed as an argument to a subprogram, the dynamic predicate is checked because the subprogram requires the object to be consistent. This happens in the last call to `Display_Tests` in our example. Here, the predicate check fails because the previous assignment has a non-conformant value.

13.3 Type invariants

Type invariants are another way of specifying expectations regarding types. While predicates are used for *non-private* types, type invariants are used exclusively to define expectations about private types. If a type T from a package P has a type invariant, the results of operations on objects of type T are always consistent with that invariant.

Type invariants are specified with a `with Type_Invariant => <property>` clause. Like predicates, the *property* defines a condition that allows us to check if an object of type T is conformant to its requirements. In this sense, type invariants can be viewed as a sort of predicate for private types. However, there are some differences in terms of checks. The following table summarizes the differences:

Element	Subprogram parameter checks	Assignment checks
Predicates	On all `in` and `out` parameters	On assignments and explicit initializations
Type invariants	On `out` parameters returned from subprograms declared in the same public scope	On all initializations

We could rewrite our previous example and replace dynamic predicates by type invariants. It would look like this:

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Calendar;          use Ada.Calendar;
with Ada.Containers.Vectors;

procedure Show_Type_Invariant is

   package Courses is
      type Course is private
        with Type_Invariant => Check (Course);
```

(continues on next page)

(continued from previous page)

```
      type Course_Container is private;

      procedure Add (CC : in out Course_Container; C : Course);

      function Init
        (Name : String; Start_Date, End_Date : Time) return Course;

      function Check (C : Course) return Boolean;

   private
      type Course is record
         Name       : Unbounded_String;
         Start_Date : Time;
         End_Date   : Time;
      end record;

      function Check (C          : Course) return Boolean is
        (C.Start_Date <= C.End_Date);

      package Course_Vectors is new Ada.Containers.Vectors
        (Index_Type   => Natural,
         Element_Type => Course);

      type Course_Container is record
         V : Course_Vectors.Vector;
      end record;
   end Courses;

   package body Courses is
      procedure Add (CC : in out Course_Container; C : Course) is
      begin
         CC.V.Append (C);
      end Add;

      function Init
        (Name : String; Start_Date, End_Date : Time) return Course is
      begin
         return Course'(Name       => To_Unbounded_String (Name),
                        Start_Date => Start_Date,
                        End_Date   => End_Date);
      end Init;
   end Courses;

   use Courses;

   CC : Course_Container;
begin
   Add (CC,
        Init (Name       => "Intro to Photography",
              Start_Date => Time_Of (2018, 5, 1),
              End_Date   => Time_Of (2018, 5, 10)));

   --  This should trigger an error in the type-invariant check
   Add (CC,
        Init (Name       => "Intro to Video Recording",
              Start_Date => Time_Of (2019, 5, 1),
              End_Date   => Time_Of (2018, 5, 10)));
end Show_Type_Invariant;
```

The major difference is that the `Course` type was a visible (public) type of the `Courses` package in the previous example, but in this example is a private type.

CHAPTER

FOURTEEN

INTERFACING WITH C

Ada allows us to interface with code in many languages, including C and C++. This section discusses how to interface with C.

14.1 Multi-language project

By default, when using **gprbuild** we only compile Ada source files. To compile C files as well, we need to modify the project file used by **gprbuild**. We use the `Languages` entry, as in the following example:

```
project Multilang is

   for Languages use ("ada", "c");

   for Source_Dirs use ("src");
   for Main use ("main.adb");
   for Object_Dir use "obj";

end Multilang;
```

14.2 Type convention

To interface with data types declared in a C application, you specify the `Convention` aspect on the corresponding Ada type declaration. In the following example, we interface with the `C_Enum` enumeration declared in a C source file:

```
procedure Show_C_Enum is

   type C_Enum is (A, B, C) with Convention => C;
   --  Use C convention for C_Enum
begin
   null;
end Show_C_Enum;
```

To interface with C's built-in types, we use the `Interfaces.C` package, which contains most of the type definitions we need. For example:

```
with Interfaces.C; use Interfaces.C;

procedure Show_C_Struct is

   type c_struct is record
      a : int;
```

(continues on next page)

(continued from previous page)

```
      b : long;
      c : unsigned;
      d : double;
   end record
     with Convention => C;

begin
   null;
end Show_C_Struct;
```

Here, we're interfacing with a C struct (`C_Struct`) and using the corresponding data types in C (`int`, `long`, `unsigned` and `double`). This is the declaration in C:

```
struct c_struct
{
    int         a;
    long        b;
    unsigned    c;
    double      d;
};
```

14.3 Foreign subprograms

14.3.1 Calling C subprograms in Ada

We use a similar approach when interfacing with subprograms written in C. In this case, an additional aspect is required: `Import`. For example:

```
with Interfaces.C; use Interfaces.C;

procedure Show_C_Func is

   function my_func (a : int) return int
     with
       Import        => True,
       Convention    => C;
   --  Imports function 'my_func' from C.
   --  You can now call it from Ada.

begin
   null;
end Show_C_Func;
```

This code interfaces with the following declaration in the C header file:

```
int my_func (int a);
```

Here's the corresponding C definition:

```
#include "my_func.h"

int my_func (int a)
{
    return a * 2;
}
```

If you want, you can use a different subprogram name in the Ada code. For example, we could call the C function `Get_Value`:

```
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO;  use Ada.Text_IO;

procedure Show_C_Func is

   function Get_Value (a : int) return int
     with
       Import        => True,
       Convention    => C,
       External_Name => "my_func";

   --  Imports function 'my_func' from C and
   --  rename it to 'Get_Value'

   V : int;
begin
   V := Get_Value (2);
   Put_Line ("Result is " & int'Image (V));
end Show_C_Func;
```

14.3.2 Calling Ada subprograms in C

You can also call Ada subprograms from C applications. You do this with the `Export` aspect. For example:

```
with Interfaces.C; use Interfaces.C;

package C_API is

   function My_Func (a : int) return int
     with
       Export        => True,
       Convention    => C,
       External_Name => "my_func";

end C_API;
```

This is the corresponding body that implements that function:

```
package body C_API is

   function My_Func (a : int) return int is
   begin
      return a * 2;
   end My_Func;

end C_API;
```

On the C side, we do the same as we would if the function were written in C: simply declare it using the `extern` keyword. For example:

```
#include <stdio.h>

extern int my_func (int a);

int main (int argc, char **argv) {

  int v = my_func(2);
```

(continues on next page)

(continued from previous page)

```
  printf("Result is %d\n", v);

  return 0;
}
```

14.4 Foreign variables

14.4.1 Using C global variables in Ada

To use global variables from C code, we use the same method as subprograms: we specify the `Import` and `Convention` aspects for each variable we want to import.

Let's reuse an example from the previous section. We'll add a global variable (`func_cnt`) to count the number of times the function (`my_func`) is called:

```
/*% filename: test.h */

extern int func_cnt;

int my_func (int a);
```

The variable is declared in the C file and incremented in `my_func`:

```
#include "test.h"

int func_cnt = 0;

int my_func (int a)
{
  func_cnt++;

  return a * 2;
}
```

In the Ada application, we just reference the foreign variable:

```
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO;  use Ada.Text_IO;

procedure Show_C_Func is

   function my_func (a : int) return int
     with
       Import        => True,
       Convention    => C;

   V : int;

   func_cnt : int
     with
       Import        => True,
       Convention    => C;
   --  We can access the func_cnt variable from test.c

begin
   V := my_func (1);
   V := my_func (2);
```

(continues on next page)

(continued from previous page)

```
   V := my_func (3);
   Put_Line ("Result is " & int'Image (V));

   Put_Line ("Function was called " & int'Image (func_cnt) & " times");
end Show_C_Func;
```

As we see by running the application, the value of the counter is the number of times `my_func` was called.

We can use the `External_Name` aspect to give a different name for the variable in the Ada application in the same we do for subprograms.

14.4.2 Using Ada variables in C

You can also use variables declared in Ada files in C applications. In the same way as we did for subprograms, you do this with the `Export` aspect.

Let's reuse a past example and add a counter, as in the previous example, but this time have the counter incremented in Ada code:

```
with Interfaces.C; use Interfaces.C;

package C_API is

   func_cnt : int := 0
     with
       Export     => True,
       Convention => C;

   function My_Func (a : int) return int
     with
       Export        => True,
       Convention    => C,
       External_Name => "my_func";

end C_API;
```

The variable is then increment in `My_Func`:

```
--% filename: c_api.adb
package body C_API is

   function My_Func (a : int) return int is
   begin
      func_cnt := func_cnt + 1;
      return a * 2;
   end My_Func;

end C_API;
```

In the C application, we just need to declare the variable and use it:

```
#include <stdio.h>

extern int my_func (int a);

extern int func_cnt;

int main (int argc, char **argv) {
```

(continues on next page)

(continued from previous page)

```
  int v;

  v = my_func(1);
  v = my_func(2);
  v = my_func(3);

  printf("Result is %d\n", v);

  printf("Function was called %d times\n", func_cnt);

  return 0;
}
```

Again, by running the application, we see that the value from the counter is the number of times that `my_func` was called.

14.5 Generating bindings

In the examples above, we manually added aspects to our Ada code to correspond to the C source-code we're interfacing with. This is called creating a *binding*. We can automate this process by using the *Ada spec dump* compiler option: `-fdump-ada-spec`. We illustrate this by revisiting our previous example.

This was our C header file:

```
extern int func_cnt;

int my_func (int a);
```

To create Ada bindings, we'll call the compiler like this:

```
gcc -c -fdump-ada-spec -C ./test.h
```

The result is an Ada spec file called `test_h.ads`:

```
pragma Ada_2005;
pragma Style_Checks (Off);

with Interfaces.C; use Interfaces.C;

package test_h is

   func_cnt : aliased int;  -- ./test.h:3
   pragma Import (C, func_cnt, "func_cnt");

   function my_func (arg1 : int) return int;  -- ./test.h:5
   pragma Import (C, my_func, "my_func");

end test_h;
```

Now we simply refer to this `test_h` package in our Ada application:

```
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO;  use Ada.Text_IO;
with test_h;       use test_h;

procedure Show_C_Func is
```

(continues on next page)

(continued from previous page)

```
   V : int;
begin
   V := my_func (1);
   V := my_func (2);
   V := my_func (3);
   Put_Line ("Result is " & int'Image (V));

   Put_Line ("Function was called " & int'Image (func_cnt) & " times");
end Show_C_Func;
```

You can specify the name of the parent unit for the bindings you're creating as the operand to `fdump-ada-spec`:

```
gcc -c -fdump-ada-spec -fada-spec-parent=Ext_C_Code -C ./test.h
```

This creates the file `ext_c_code-test_h.ads`:

```
package Ext_C_Code.test_h is

   --  automatic generated bindings...

end Ext_C_Code.test_h;
```

14.5.1 Adapting bindings

The compiler does the best it can when creating bindings for a C header file. However, sometimes it has to guess about the translation and the generated bindings don't always match our expectations. For example, this can happen when creating bindings for functions that have pointers as arguments. In this case, the compiler may use `System.Address` as the type of one or more pointers. Although this approach works fine (as we'll see later), this is usually not how a human would interpret the C header file. The following example illustrates this issue.

Let's start with this C header file:

```
/*% filename: test.h */

struct test;

struct test * test_create(void);

void test_destroy(struct test *t);

void test_reset(struct test *t);

void test_set_name(struct test *t, char *name);

void test_set_address(struct test *t, char *address);

void test_display(const struct test *t);
```

And the corresponding C implementation:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "test.h"

struct test {
```

(continues on next page)

(continued from previous page)

```
  char name[80];
  char address[120];
};

static size_t
strlcpy(char *dst, const char *src, size_t dstsize)
{
  size_t len = strlen(src);
  if (dstsize) {
    size_t bl = (len < dstsize-1 ? len : dstsize-1);
    ((char*)memcpy(dst, src, bl))[bl] = 0;
  }
  return len;
}

struct test * test_create(void)
{
  return malloc (sizeof (struct test));
}

void test_destroy(struct test *t)
{
  if (t != NULL) {
    free(t);
  }
}

void test_reset(struct test *t)
{
  t->name[0]    = '\0';
  t->address[0] = '\0';
}

void test_set_name(struct test *t, char *name)
{
  strlcpy(t->name, name, sizeof(t->name));
}

void test_set_address(struct test *t, char *address)
{
  strlcpy(t->address, address, sizeof(t->address));
}

void test_display(const struct test *t)
{
  printf("Name:    %s\n", t->name);
  printf("Address: %s\n", t->address);
}
```

Next, we'll create our bindings:

```
gcc -c -fdump-ada-spec -C ./test.h
```

This creates the following specification in `test_h.ads`:

```
pragma Ada_2005;
pragma Style_Checks (Off);

with Interfaces.C; use Interfaces.C;
with System;
with Interfaces.C.Strings;
```

(continues on next page)

(continued from previous page)

```
package test_h is

   --  skipped empty struct test

   function test_create return System.Address;  -- ./test.h:5
   pragma Import (C, test_create, "test_create");

   procedure test_destroy (arg1 : System.Address);  -- ./test.h:7
   pragma Import (C, test_destroy, "test_destroy");

   procedure test_reset (arg1 : System.Address);  -- ./test.h:9
   pragma Import (C, test_reset, "test_reset");

   procedure test_set_name (arg1 : System.Address; arg2 : Interfaces.C.Strings.
 ↪chars_ptr);  -- ./test.h:11
   pragma Import (C, test_set_name, "test_set_name");

   procedure test_set_address (arg1 : System.Address; arg2 : Interfaces.C.Strings.
 ↪chars_ptr);  -- ./test.h:13
   pragma Import (C, test_set_address, "test_set_address");

   procedure test_display (arg1 : System.Address);  -- ./test.h:15
   pragma Import (C, test_display, "test_display");

end test_h;
```

As we can see, the binding generator completely ignores the declaration `struct test` and all references to the `test` struct are replaced by addresses (`System.Address`). Nevertheless, these bindings are good enough to allow us to create a test application in Ada:

```
with Interfaces.C;         use Interfaces.C;
with Interfaces.C.Strings; use Interfaces.C.Strings;
with Ada.Text_IO;          use Ada.Text_IO;
with test_h;               use test_h;

with System;

procedure Show_Automatic_C_Struct_Bindings is

   Name    : constant chars_ptr := New_String ("John Doe");
   Address : constant chars_ptr := New_String ("Small Town");

   T : System.Address := test_create;

begin
   test_reset (T);
   test_set_name (T, Name);
   test_set_address (T, Address);

   test_display (T);
   test_destroy (T);
end Show_Automatic_C_Struct_Bindings;
```

We can successfully bind our C code with Ada using the automatically-generated bindings, but they aren't ideal. Instead, we would prefer Ada bindings that match our (human) interpretation of the C header file. This requires manual analysis of the header file. The good news is that we can use the automatic generated bindings as a starting point and adapt them to our needs. For example, we can:

1. Define a `Test` type based on `System.Address` and use it in all relevant functions.

2. Remove the `test_` prefix in all operations on the `Test` type.

This is the resulting specification:

```
with Interfaces.C; use Interfaces.C;
with System;
with Interfaces.C.Strings;

package adapted_test_h is

   type Test is new System.Address;

   function Create return Test;
   pragma Import (C, Create, "test_create");

   procedure Destroy (T : Test);
   pragma Import (C, Destroy, "test_destroy");

   procedure Reset (T : Test);
   pragma Import (C, Reset, "test_reset");

   procedure Set_Name (T    : Test;
                       Name : Interfaces.C.Strings.chars_ptr);  -- ./test.h:11
   pragma Import (C, Set_Name, "test_set_name");

   procedure Set_Address (T       : Test;
                          Address : Interfaces.C.Strings.chars_ptr);
   pragma Import (C, Set_Address, "test_set_address");

   procedure Display (T : Test);  -- ./test.h:15
   pragma Import (C, Display, "test_display");

end adapted_test_h;
```

And this is the corresponding Ada body:

```
with Interfaces.C;         use Interfaces.C;
with Interfaces.C.Strings; use Interfaces.C.Strings;
with Ada.Text_IO;          use  Ada.Text_IO;
with adapted_test_h;       use  adapted_test_h;

with System;

procedure Show_Adapted_C_Struct_Bindings is

   Name    : constant chars_ptr := New_String ("John Doe");
   Address : constant chars_ptr := New_String ("Small Town");

   T : Test := Create;

begin
   Reset (T);
   Set_Name (T, Name);
   Set_Address (T, Address);

   Display (T);
   Destroy (T);
end Show_Adapted_C_Struct_Bindings;
```

Now we can use the `Test` type and its operations in a clean, readable way.

CHAPTER

FIFTEEN

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a large and ill-defined concept in programming languages and one that tends to encompass many different meanings because different languages often implement their own vision of it, with similarities and differences from the implementations in other languages.

However, one model mostly "won" the battle of what object-oriented means, if only by sheer popularity. It's the model used in the Java programming language, which is very similar to the one used by C++. Here are some defining characteristics:

- Type derivation and extension: Most object oriented languages allow the user to add fields to derived types.
- Subtyping: Objects of a type derived from a base type can, in some instances, be substituted for objects of the base type.
- Runtime polymorphism: Calling a subprogram, usually called a *method*, attached to an object type can dispatch at runtime depending on the exact type of the object.
- Encapsulation: Objects can hide some of their data.
- Extensibility: People from the "outside" of your package, or even your whole library, can derive from your object types and define their own behaviors.

Ada dates from before object-oriented programming was as popular as it is today. Some of the mechanisms and concepts from the above list were in the earliest version of Ada even before what we would call OOP was added:

- As we saw, encapsulation is not implemented at the type level in Ada, but instead at the package level.
- Subtyping can be implemented using, well, subtypes, which have a full and permissive static substitability model. The substitution will fail at runtime if the dynamic constraints of the subtype are not fulfilled.
- Runtime polymorphism can be implemented using variant records.

However, this lists leaves out type extensions, if you don't consider variant records, and extensibility.

The 1995 revision of Ada added a feature filling the gaps, which allowed people to program following the object-oriented paradigm in an easier fashion. This feature is called *tagged types*.

Note: It's possible to program in Ada without ever creating tagged types. If that's your prefered style of programming or you have no specific use for tagged types, feel free to not use them, as is the case for many features of Ada.

However, they can be the best way to express solutions to certain problems and they may be the best way to solve your problem. If that's the case, read on!

15.1 Derived types

Before presenting tagged types, we should discuss a topic we have brushed on, but not really covered, up to now:

You can create one or more new types from every type in Ada. Type derivation is built into the language.

```
package Newtypes is
   type Point is record
       X, Y : Integer;
   end record;

   type New_Point is new Point;
end Newtypes;
```

Type derivation is useful to enforce strong typing because the type system treats the two types as incompatible.

But the benefits are not limited to that: you can inherit things from the type you derive from. You not only inherit the representation of the data, but you can also inherit behavior.

When you inherit a type you also inherit what are called *primitive operations*. A primitive operation (or just a *primitive*) is a subprogram attached to a type. Ada defines primitives as subprograms defined in the same scope as the type.

Attention: A subprogram will only become a primitive of the type if:

1. The subprogram is declared in the same scope as the type and
2. The type and the subprogram are declared in a package

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

procedure Primitives is
  package Week is
    type Days is (Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday, Sunday);

     --  Print day is a primitive of the type Days
    procedure Print_Day (D : Days);
  end Week;

  package body Week is
    procedure Print_Day (D : Days) is
    begin
       Put_Line (Days'Image (D));
    end Print_Day;
  end Week;

  use Week;
  type Weekend_Days is new Days range Saturday .. Sunday;

  --  A procedure Print_Day is automatically inherited here. It is as if
  --  the procedure
  --
  --  procedure Print_Day (D : Weekend_Days);
  --
  --  has been declared with the same body
```

(continues on next page)

(continued from previous page)

```
  Sat : Weekend_Days := Saturday;
begin
   Print_Day (Sat);
end Primitives;
```

This kind of inheritance can be very useful, and is not limited to record types (you can use it on discrete types, as in the example above), but it's only superficially similar to object-oriented inheritance:

- Records can't be extended using this mechanism alone. You also can't specify a new representation for the new type: it will **always** have the same representation as the base type.
- There's no facility for dynamic dispatch or polymorphism. Objects are of a fixed, static type.

There are other differences, but it's not useful to list them all here. Just remember that this is a kind of inheritance you can use if you only want to statically inherit behavior without duplicating code or using composition, but a kind you can't use if you want any dynamic features that are usually associated with OOP.

15.2 Tagged types

The 1995 revision of the Ada language introduced tagged types to fullfil the need for an unified solution that allows programming in an object-oriented style similar to the one described at the beginning of this chapter.

Tagged types are very similar to normal records except that some functionality is added:

- Types have a *tag*, stored inside each object, that identifies the runtime type[19] of that object.
- Primitives can dispatch. A primitive on a tagged type is what you would call a *method* in Java or C++. If you derive a base type and override a primitive of it, you can often call it on an object with the result that which primitive is called depends on the exact runtime type of the object.
- Subtyping rules are introduced allowing a tagged type derived from a base type to be statically compatible with the base type.

Let's see our first tagged type declarations:

```
package P is
   type My_Class is tagged null record;
   --  Just like a regular record, but with tagged qualifier

   --  Methods are outside of the type definition:

   procedure Foo (Self : in out My_Class);
   --  If you define a procedure taking a My_Class argument
   --  in the same package, it will be a method.

   --  Here's how you derive a tagged type:

   type Derived is new My_Class with record
       A : Integer;
       --  You can add fields in derived types.
   end record;

   overriding procedure Foo (Self : in out Derived);
   --  The "overriding" qualifier is optional, but if it is present,
```

(continues on next page)

[19] https://en.wikipedia.org/wiki/Run-time_type_information

(continued from previous page)

```
   --  it must be valid.
end P;
```

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

package body P is
   procedure Foo (Self : in out My_Class) is
   begin
      Put_Line ("In My_Class.Foo");
   end Foo;

   procedure Foo (Self : in out Derived) is
   begin
      Put_Line ("In Derived.Foo, A = " & Integer'Image (Self.A));
   end Foo;
end P;
```

15.3 Classwide types

To remain consistent with the rest of the language, a new notation needed to be introduced to say "This object is of this type or any descendent derives tagged type".

In Ada, we call this the *classwide type*. It's used in OOP as soon as you need polymorphism. For example, you can't do the following:

```
with P; use P;

procedure Main is

   O1 : My_Class;
   --  Declaring an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declaring an object of type Derived

   O3 : My_Class := O2;
   --  INVALID: Trying to assign a value of type derived to a variable of
   --  type My_Class.
begin
   null;
end Main;
```

This is because an object of a type T is exactly of the type T, whether T is tagged or not. What you want to say as a programmer is "I want O3 to be able to hold an object of type `My_Class` or any type descending from `My_Class`". Here's how you do that:

```
with P; use P;

procedure Main is
   O1 : My_Class;
   --  Declare an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declare an object of type Derived

   O3 : My_Class'Class := O2;
   --  Now valid: My_Class'Class designates the classwide type for
```

(continues on next page)

(continued from previous page)

```
   --  My_Class, which is the set of all types descending from My_Class
   --  (including My_Class).
begin
   null;
end Main;
```

Attention: Because an object of a classwide type can be the size of any descendent of its base type, it has an unknown size. It's therefore an indefinite type, with the expected restrictions:

- It can't be stored as a field/component of a record
- An object of a classwide type needs to be initialized immediately (you can't specify the constraints of such a type in any way other than by initializing it).

15.4 Dispatching operations

We saw that you can override operations in types derived from another tagged type. The eventual goal of OOP is to make a dispatching call: a call to a primitive (method) that depends on the exact type of the object.

But, if you think carefully about it, a variable of type `My_Class` always contains an object of exactly that type. If you want to have a variable that can contain a `My_Class` or any derived type, it has to be of type `My_Class'Class`.

In other words, to make a dispatching call, you must first have an object that can be either of a type or any type derived from this type, namely an object of a classwide type.

```
with P; use P;

procedure Main is
   O1 : My_Class;
   --  Declare an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declare an object of type Derived

   O3 : My_Class'Class := O2;

   O4 : My_Class'Class := O1;
begin
   Foo (O1);
   --  Non dispatching: Calls My_Class.Foo
   Foo (O2);
   --  Non dispatching: Calls Derived.Foo
   Foo (O3);
   --  Dispatching: Calls Derived.Foo
   Foo (O4);
   --  Dispatching: Calls My_Class.Foo
end Main;
```

Attention: You can convert an object of type `Derived` to an object of type `My_Class`. This is called a *view conversion* in Ada parlance and is useful, for example, if you want to call a parent method.

In that case, the object really is converted to a `My_Class` object, which means its tag is changed. Since tagged objects are always passed by reference, you can use this kind of conversion to

modify the state of an object: changes to converted object will affect the original one.

```
with P; use P;

procedure Main is
   O1 : Derived := (A => 12);
   --  Declare an object of type Derived

   O2 : My_Class := My_Class (O1);

   O3 : My_Class'Class := O2;
begin
   Foo (O1);
   --  Non dispatching: Calls Derived.Foo
   Foo (O2);
   --  Non dispatching: Calls My_Class.Foo

   Foo (O3);
   --  Dispatching: Calls My_Class.Foo
end Main;
```

15.5 Dot notation

You can also call primitives of tagged types with a notation that's more familiar to object oriented programmers. Given the Foo primitive above, you can also write the above program this way:

```
with P; use P;

procedure Main is
   O1 : My_Class;
   --  Declare an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declare an object of type Derived

   O3 : My_Class'Class := O2;

   O4 : My_Class'Class := O1;
begin
   O1.Foo;
   --  Non dispatching: Calls My_Class.Foo
   O2.Foo;
   --  Non dispatching: Calls Derived.Foo
   O3.Foo;
   --  Dispatching: Calls Derived.Foo
   O4.Foo;
   --  Dispatching: Calls My_Class.Foo
end Main;
```

If the dispatching parameter of a primitive is the first parameter, which is the case in our examples, you can call the primitive using the dot notation. Any remaining parameter are passed normally:

```
with P; use P;

procedure Main is
   package Extend is
      type D2 is new Derived with null record;
```

(continues on next page)

(continued from previous page)

```
      procedure Bar (Self : in out D2; Val : Integer);
   end Extend;

   package body Extend is
      procedure Bar (Self : in out D2; Val : Integer) is
      begin
         Self.A := Self.A + Val;
      end Bar;
   end Extend;

   use Extend;

   Obj : D2 := (A => 15);
begin
   Obj.Bar (2);
   Obj.Foo;
end Main;
```

15.6 Private & Limited

We've seen previously (in the *Privacy* (page 75) chapter) that types can be declared limited or private. These encapsulation techniques can also be applied to tagged types, as we'll see in this section.

This is an example of a tagged private type:

```
package P is
   type T is tagged private;
private
   type T is tagged record
       E : Integer;
   end record;
end P;
```

This is an example of a tagged limited type:

```
package P is
   type T is tagged limited record
       E : Integer;
   end record;
end P;
```

Naturally, you can combine both *limited* and *private* types and declare a tagged limited private type:

```
package P is
   type T is tagged limited private;

   procedure Init (A : in out T);
private
   type T is tagged limited record
       E : Integer;
   end record;
end P;
```

```
package body P is

   procedure Init (A : in out T) is
   begin
      A.E := 0;
```

(continues on next page)

(continued from previous page)

```
   end Init;

end P;
```

```
with P; use P;

procedure Main is
  T1, T2 : T;
begin
  T1.Init;
  T2.Init;

  --  The following line doesn't work because type T is private:
  --  T1.E := 0;

  --  The following line doesn't work because type T is limited:
  --  T2 := T1;
end Main;
```

Note that the code in the `Main` procedure above presents two assignments that trigger compilation errors because type `T` is limited private. In fact, you cannot:

- assign to `T1.E` directly because type `T` is private;
- assign `T1` to `T2` because type `T` is limited.

In this case, there's no distinction between tagged and non-tagged types: these compilation errors would also occur for non-tagged types.

15.7 Classwide access types

In this section, we'll discuss an useful pattern for object-oriented programming in Ada: classwide access type. Let's start with an example where we declare a tagged type `T` and a derived type `T_New`:

```
package P is
   type T is tagged null record;

   procedure Show (Dummy : T);

   type T_New is new T with null record;

   procedure Show (Dummy : T_New);
end P;
```

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

package body P is

   procedure Show (Dummy : T) is
   begin
      Put_Line ("Using type " & T'External_Tag);
   end Show;

   procedure Show (Dummy : T_New) is
   begin
      Put_Line ("Using type " & T_New'External_Tag);
   end Show;
```

(continues on next page)

(continued from previous page)

```
end P;
```

Note that we're using null records for both types T and T_New. Although these types don't actually have any component, we can still use them to demonstrate dispatching. Also note that the example above makes use of the 'External_Tag attribute in the implementation of the Show procedure to get a string for the corresponding tagged type.

As we've seen before, we must use a classwide type to create objects that can make dispatching calls. In other words, objects of type T'Class will dispatch. For example:

```
with P; use P;

procedure Dispatching_Example is
  T2         :          T_New;
  T_Dispatch : constant T'Class := T2;
begin
  T_Dispatch.Show;
end Dispatching_Example;
```

A more useful application is to declare an array of objects that can dispatch. For example, we'd like to declare an array T_Arr, loop over this array and dispatch according to the actual type of each individual element:

```
for I in T_Arr'Range loop
   T_Arr (I).Show;
   --  Call Show procedure according to actual type of T_Arr (I)
end loop;
```

However, it's not possible to declare an array of type T'Class directly:

```
with P; use P;

procedure Classwide_Compilation_Error is
  T_Arr  : array (1 .. 2) of T'Class;
  --                         ^ Compilation Error!
begin
  for I in T_Arr'Range loop
     T_Arr (I).Show;
  end loop;
end Classwide_Compilation_Error;
```

In fact, it's impossible for the compiler to know which type would actually be used for each element of the array. However, if we use dynamic allocation via access types, we can allocate objects of different types for the individual elements of an array T_Arr. We do this by using classwide access types, which have the following format:

```
type T_Class is access T'Class;
```

We can rewrite the previous example using the T_Class type. In this case, dynamically allocated objects of this type will dispatch according to the actual type used during the allocation. Also, let's introduce an Init procedure that won't be overridden for the derived T_New type. This is the adapted code:

```
package P is
   type T is tagged record
       E : Integer;
   end record;

   type T_Class is access T'Class;
```

(continues on next page)

(continued from previous page)

```
   procedure Init (A : in out T);

   procedure Show (Dummy : T);

   type T_New is new T with null record;

   procedure Show (Dummy : T_New);

end P;
```

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

package body P is

   procedure Init (A : in out T) is
   begin
      Put_Line ("Initializing type T...");
      A.E := 0;
   end Init;

   procedure Show (Dummy : T) is
   begin
      Put_Line ("Using type " & T'External_Tag);
   end Show;

   procedure Show (Dummy : T_New) is
   begin
      Put_Line ("Using type " & T_New'External_Tag);
   end Show;

end P;
```

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

procedure Main is
  T_Arr  : array (1 .. 2) of T_Class;
begin
  T_Arr (1) := new T;
  T_Arr (2) := new T_New;

  for I in T_Arr'Range loop
     Put_Line ("Element # " & Integer'Image (I));

     T_Arr (I).Init;
     T_Arr (I).Show;

     Put_Line ("-----------");
  end loop;
end Main;
```

In this example, the first element (`T_Arr (1)`) is of type `T`, while the second element is of type `T_New`. When running the example, the `Init` procedure of type `T` is called for both elements of the `T_Arr` array, while the call to the `Show` procedure selects the corresponding procedure according to the type of each element of `T_Arr`.

CHAPTER

SIXTEEN

STANDARD LIBRARY: CONTAINERS

In previous chapters, we've used arrays as the standard way to group multiple objects of a specific data type. In many cases, arrays are good enough for manipulating those objects. However, there are situations that require more flexibility and more advanced operations. For those cases, Ada provides support for containers — such as vectors and sets — in its standard library.

We present an introduction to containers here. For a list of all containers available in Ada, see *Appendix B* (page 189).

16.1 Vectors

In the following sections, we present a general overview of vectors, including instantiation, initialization, and operations on vector elements and vectors.

16.1.1 Instantiation

Here's an example showing the instantiation and declaration of a vector `V`:

```
with Ada.Containers.Vectors;

procedure Show_Vector_Inst is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   V : Integer_Vectors.Vector;
begin
   null;
end Show_Vector_Inst;
```

Containers are based on generic packages, so we can't simply declare a vector as we would declare an array of a specific type:

```
A : array (1 .. 10) of Integer;
```

Instead, we first need to instantiate one of those packages. We `with` the container package (`Ada.Containers.Vectors` in this case) and instantiate it to create an instance of the generic package for the desired type. Only then can we declare the vector using the type from the instantiated package. This instantiation needs to be done for any container type from the standard library.

In the instantiation of `Integer_Vectors`, we indicate that the vector contains elements of `Integer` type by specifying it as the `Element_Type`. By setting `Index_Type` to `Natural`, we specify that the allowed range includes all natural numbers. We could have used a more restrictive range if desired.

16.1.2 Initialization

One way to initialize a vector is from a concatenation of elements. We use the & operator, as shown in the following example:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Init is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector has "
             & Count_Type'Image (V.Length) & " elements");
end Show_Vector_Init;
```

We specify `use Integer_Vectors`, so we have direct access to the types and operations from the instantiated package. Also, the example introduces another operation on the vector: `Length`, which retrieves the number of elements in the vector. We can use the dot notation because `Vector` is a tagged type, allowing us to write either `V.Length` or `Length (V)`.

16.1.3 Appending and prepending elements

You add elements to a vector using the `Prepend` and `Append` operations. As the names suggest, these operations add elements to the beginning or end of a vector, respectively. For example:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Append is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   V : Vector;
begin
   Put_Line ("Appending some elements to the vector...");
   V.Append (20);
   V.Append (10);
   V.Append (0);
   V.Append (13);
   Put_Line ("Finished appending.");

   Put_Line ("Prepending some elements to the vector...");
   V.Prepend (30);
   V.Prepend (40);
   V.Prepend (100);
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("Finished prepending.");

   Put_Line ("Vector has "
             & Count_Type'Image (V.Length) & " elements");
end Show_Vector_Append;
```

This example puts elements into the vector in the following sequence: (100, 40, 30, 20, 10, 0, 13).

The Reference Manual specifies that the worst-case complexity must be:

- O($logN$) for the `Append` operation, and
- O($NlogN$) for the `Prepend` operation.

16.1.4 Accessing first and last elements

We access the first and last elements of a vector using the `First_Element` and `Last_Element` functions. For example:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_First_Last_Element is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   function Img (I : Integer)    return String renames Integer'Image;
   function Img (I : Count_Type) return String renames Count_Type'Image;

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector has " & Img (V.Length) & " elements");

   --  Using V.First_Element to retrieve first element
   Put_Line ("First element is " & Img (V.First_Element));

   --  Using V.Last_Element to retrieve last element
   Put_Line ("Last element is " & Img (V.Last_Element));
end Show_Vector_First_Last_Element;
```

You can swap elements by calling the procedure `Swap` and retrieving a reference (a *cursor*) to the first and last elements of the vector by calling `First` and `Last`. A cursor allows us to iterate over a container and process individual elements from it.

With these operations, we're able to write code to swap the first and last elements of a vector:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_First_Last_Element is

   package Integer_Vectors is new Ada.Containers.Vectors
```

(continues on next page)

(continued from previous page)

```
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

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

   V : Vector := 20 & 10 & 0 & 13;
begin
   --  We use V.First and V.Last to retrieve cursor for first and
   --  last elements.
   --  We use V.Swap to swap elements.
   V.Swap (V.First, V.Last);

   Put_Line ("First element is now " & Img (V.First_Element));
   Put_Line ("Last element is now " & Img (V.Last_Element));
end Show_Vector_First_Last_Element;
```

16.1.5 Iterating

The easiest way to iterate over a container is to use a `for E of Our_Container` loop. This gives us a reference (E) to the element at the current position. We can then use E directly. For example:

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Iteration is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

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

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector elements are: ");

   --
   --  Using for ... of loop to iterate:
   --
   for E of V loop
      Put_Line ("- " & Img (E));
   end loop;

end Show_Vector_Iteration;
```

This code displays each element from the vector V.

Because we're given a reference, we can display not only the value of an element but also modify it. For example, we could easily write a loop to add one to each element of vector V:

```
for E of V loop
   E := E + 1;
end loop;
```

We can also use indices to access vector elements. The format is similar to a loop over array elements: we use a `for I in <range>` loop. The range is provided by `V.First_Index` and `V.Last_Index`. We can access the current element by using it as an array index: `V (I)`. For example:

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Index_Iteration is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector elements are: ");

   --
   --  Using indices in a "for I in ..." loop to iterate:
   --
   for I in V.First_Index .. V.Last_Index loop
      --  Displaying current index I
      Put ("- ["
           & Extended_Index'Image (I)
           & "] ");

      Put (Integer'Image (V (I)));

      --  We could also use the V.Element (I) function to retrieve the
      --  element at the current index I

      New_Line;
   end loop;

end Show_Vector_Index_Iteration;
```

Here, in addition to displaying the vector elements, we're also displaying each index, `I`, just like what we can do for array indices. Also, we can access the element by using either the short form `V (I)` or the longer form `V.Element (I)` but not `V.I`.

As mentioned in the previous section, you can use cursors to iterate over containers. For this, use the function `Iterate`, which retrieves a cursor for each position in the vector. The corresponding loop has the format `for C in V.Iterate loop`. Like the previous example using indices, you can again access the current element by using the cursor as an array index: `V (C)`. For example:

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Cursor_Iteration is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13;
```

(continues on next page)

(continued from previous page)

```
begin
   Put_Line ("Vector elements are: ");

   --
   --  Use a cursor to iterate in a loop:
   --
   for C in V.Iterate loop
      --  Using To_Index function to retrieve index
      --  for the cursor position
      Put ("- ["
           & Extended_Index'Image (To_Index (C))
           & "] ");

      Put (Integer'Image (V (C)));

      --  We could use Element (C) to retrieve the vector
      --  element for the cursor position

      New_Line;
   end loop;

   --  Alternatively, we could iterate with a while-loop:
   --
   --  declare
   --     C : Cursor := V.First;
   --  begin
   --     while C /= No_Element loop
   --        some processing here...
   --
   --        C := Next (C);
   --     end loop;
   --  end;

end Show_Vector_Cursor_Iteration;
```

Instead of accessing an element in the loop using `V (C)`, we could also have used the longer form `Element (C)`. In this example, we're using the function `To_Index` to retrieve the index corresponding to the current cursor.

As shown in the comments after the loop, we could also use a `while ... loop` to iterate over the vector. In this case, we would start with a cursor for the first element (retrieved by calling `V.First`) and then call `Next (C)` to retrieve a cursor for subsequent elements. `Next (C)` returns `No_Element` when the cursor reaches the end of the vector.

You can directly modify the elements using a reference. This is what it looks like when using both indices and cursors:

```
--  Modify vector elements using index
for I in V.First_Index .. V.Last_Index loop
   V (I) := V (I) + 1;
end loop;

--  Modify vector elements using cursor
for C in V.Iterate loop
   V (C) := V (C) + 1;
end loop;
```

The Reference Manual requires that the worst-case complexity for accessing an element be O($logN$).

Another way of modifing elements of a vector is using a *process procedure*, which takes an individual element and does some processing on it. You can call `Update_Element` and pass both a cursor

and an access to the process procedure. For example:

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Update is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   procedure Add_One (I : in out Integer) is
   begin
      I := I + 1;
   end Add_One;

   V : Vector := 20 & 10 & 12;
begin
   --
   --  Use V.Update_Element to process elements
   --
   for C in V.Iterate loop
      V.Update_Element (C, Add_One'Access);
   end loop;

end Show_Vector_Update;
```

16.1.6 Finding and changing elements

You can locate a specific element in a vector by retrieving its index. `Find_Index` retrieves the index of the first element matching the value you're looking for. Alternatively, you can use `Find` to retrieve a cursor referencing that element. For example:

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Find_Vector_Element is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13;
   Idx : Extended_Index;
   C   : Cursor;
begin
   --  Using Find_Index to retrieve index of element with value 10
   Idx := V.Find_Index (10);
   Put_Line ("Index of element with value 10 is "
             & Extended_Index'Image (Idx));

   --  Using Find to retrieve cursor for element with value 13
   C   := V.Find (13);
   Idx := To_Index (C);
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("Index of element with value 13 is "
             & Extended_Index'Image (Idx));
end Show_Find_Vector_Element;
```

As we saw in the previous section, we can directly access vector elements by using either an index or cursor. However, an exception is raised if we try to access an element with an invalid index or cursor, so we must check whether the index or cursor is valid before using it to access an element. In our example, `Find_Index` or `Find` might not have found the element in the vector. We check for this possibility by comparing the index to `No_Index` or the cursor to `No_Element`. For example:

```
--  Modify vector element using index
if Idx /= No_Index then
   V (Idx) := 11;
end if;

--  Modify vector element using cursor
if C /= No_Element then
   V (C) := 14;
end if;
```

Instead of writing `V (C) := 14`, we could use the longer form `V.Replace_Element (C, 14)`.

16.1.7 Inserting elements

In the previous sections, we've seen examples of how to add elements to a vector:

- using the concatenation operator (&) at the vector declaration, or
- calling the `Prepend` and `Append` procedures.

You may want to insert an element at a specific position, e.g. before a certain element in the vector. You do this by calling `Insert`. For example:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Insert is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   procedure Show_Elements (V : Vector) is
   begin
      New_Line;
      Put_Line ("Vector has "
                & Count_Type'Image (V.Length) & " elements");

      if not V.Is_Empty then
         Put_Line ("Vector elements are: ");
         for E of V loop
            Put_Line ("- " & Integer'Image (E));
         end loop;
      end if;
   end Show_Elements;
```

(continues on next page)

(continued from previous page)

```
   V : Vector := 20 & 10 & 12;
   C : Cursor;
begin
   Show_Elements (V);

   New_Line;
   Put_Line ("Adding element with value 9 (before 10)...");

   --
   --  Using V.Insert to insert element into vector
   --
   C := V.Find (10);
   if C /= No_Element then
      V.Insert (C, 9);
   end if;

   Show_Elements (V);

end Show_Vector_Insert;
```

In this example, we're looking for an element with the value of 10. If we find it, we insert an element with the value of 9 before it.

16.1.8 Removing elements

You can remove elements from a vector by passing either a valid index or cursor to the `Delete` procedure. If we combine this with the functions `Find_Index` and `Find` from the previous section, we can write a program that searches for a specific element and deletes it, if found:

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Remove_Vector_Element is
   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13 & 10 & 13;
   Idx : Extended_Index;
   C   : Cursor;
begin
   --  Use Find_Index to retrieve index of element with value 10
   Idx := V.Find_Index (10);

   --  Checking whether index is valid
   if Idx /= No_Index then
      --  Removing element using V.Delete
      V.Delete (Idx);
   end if;

   --  Use Find to retrieve cursor for element with value 13
   C := V.Find (13);

   --  Check whether index is valid
   if C /= No_Element then
      --  Remove element using V.Delete
```

(continues on next page)

(continued from previous page)

```
      V.Delete (C);
   end if;

end Show_Remove_Vector_Element;
```

We can extend this approach to delete all elements matching a certain value. We just need to keep searching for the element in a loop until we get an invalid index or cursor. For example:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Remove_Vector_Elements is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   use Integer_Vectors;

   procedure Show_Elements (V : Vector) is
   begin
      New_Line;
      Put_Line ("Vector has " & Count_Type'Image (V.Length) & " elements");

      if not V.Is_Empty then
         Put_Line ("Vector elements are: ");
         for E of V loop
            Put_Line ("- " & Integer'Image (E));
         end loop;
      end if;
   end Show_Elements;

   V : Vector := 20 & 10 & 0 & 13 & 10 & 14 & 13;
begin
   Show_Elements (V);

   --
   --  Remove elements using an index
   --
   declare
      E : constant Integer := 10;
      I : Extended_Index;
   begin
      New_Line;
      Put_Line ("Removing all elements with value of "
                & Integer'Image (E) & "...");
      loop
         I := V.Find_Index (E);
         exit when I = No_Index;
         V.Delete (I);
      end loop;
   end;

   --
   --  Remove elements using a cursor
   --
   declare
      E : constant Integer := 13;
      C : Cursor;
```

(continues on next page)

(continued from previous page)

```
   begin
      New_Line;
      Put_Line ("Removing all elements with value of "
                & Integer'Image (E) & "...");
      loop
         C := V.Find (E);
         exit when C = No_Element;
         V.Delete (C);
      end loop;
   end;

   Show_Elements (V);
end Show_Remove_Vector_Elements;
```

In this example, we remove all elements with the value 10 from the vector by retrieving their index. Likewise, we remove all elements with the value 13 by retrieving their cursor.

16.1.9 Other Operations

We've seen some operations on vector elements. Here, we'll see operations on the vector as a whole. The most prominent is the concatenation of multiple vectors, but we'll also see operations on vectors, such as sorting and sorted merging operations, that view the vector as a sequence of elements and operate on the vector considering the element's relations to each other.

We do vector concatenation using the `&` operator on vectors. Let's consider two vectors `V1` and `V2`. We can concatenate them by doing `V := V1 & V2`. `V` contains the resulting vector.

The generic package `Generic_Sorting` is a child package of `Ada.Containers.Vectors`. It contains sorting and merging operations. Because it's a generic package, you can't use it directly, but have to instantiate it. In order to use these operations on a vector of integer values (`Integer_Vectors`, in our example), you need to instantiate it directly as a child of `Integer_Vectors`. The next example makes it clear how to do this.

After instantiating `Generic_Sorting`, we make all the operations available to us with the `use` statement. We can then call `Sort` to sort the vector and `Merge` to merge one vector into another.

The following example presents code that manipulates three vectors (`V1`, `V2`, `V3`) using the concatenation, sorting and merging operations:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Ops is

   package Integer_Vectors is new Ada.Containers.Vectors
     (Index_Type   => Natural,
      Element_Type => Integer);

   package Integer_Vectors_Sorting is new Integer_Vectors.Generic_Sorting;

   use Integer_Vectors;
   use Integer_Vectors_Sorting;

   procedure Show_Elements (V : Vector) is
   begin
      New_Line;
      Put_Line ("Vector has " & Count_Type'Image (V.Length) & " elements");
```

(continues on next page)

(continued from previous page)

```
      if not V.Is_Empty then
         Put_Line ("Vector elements are: ");
         for E of V loop
            Put_Line ("- " & Integer'Image (E));
         end loop;
      end if;
   end Show_Elements;

   V, V1, V2, V3 : Vector;
begin
   V1 := 10 & 12 & 18;
   V2 := 11 & 13 & 19;
   V3 := 15 & 19;

   New_Line;
   Put_Line ("---- V1 ----");
   Show_Elements (V1);

   New_Line;
   Put_Line ("---- V2 ----");
   Show_Elements (V2);

   New_Line;
   Put_Line ("---- V3 ----");
   Show_Elements (V3);

   New_Line;
   Put_Line ("Concatenating V1, V2 and V3 into V:");

   V := V1 & V2 & V3;

   Show_Elements (V);

   New_Line;
   Put_Line ("Sorting V:");

   Sort (V);

   Show_Elements (V);

   New_Line;
   Put_Line ("Merging V2 into V1:");

   Merge (V1, V2);

   Show_Elements (V1);

end Show_Vector_Ops;
```

The Reference Manual requires that the worst-case complexity of a call to `Sort` be O(N^2) and the average complexity be better than O(N^2).

16.2 Sets

Sets are another class of containers. While vectors allow duplicated elements to be inserted, sets ensure that no duplicated elements exist.

In the following sections, we'll see operations you can perform on sets. However, since many of the operations on vectors are similar to the ones used for sets, we'll cover them more quickly here.

Please refer back to the section on vectors for a more detailed discussion.

16.2.1 Initialization and iteration

To initialize a set, you can call the `Insert` procedure. However, if you do, you need to ensure no duplicate elements are being inserted: if you try to insert a duplicate, you'll get an exception. If you have less control over the elements to be inserted so that there may be duplicates, you can use another option instead:

- a version of `Insert` that returns a Boolean value indicating whether the insertion was successful;
- the `Include` procedure, which silently ignores any attempt to insert a duplicated element.

To iterate over a set, you can use a `for E of S` loop, as you saw for vectors. This gives you a reference to each element in the set.

Let's see an example:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Init is

   package Integer_Sets is new Ada.Containers.Ordered_Sets
     (Element_Type => Integer);

   use Integer_Sets;

   S : Set;
   --  Same as:  S : Integer_Sets.Set;
   C : Cursor;
   Ins : Boolean;
begin
   S.Insert (20);
   S.Insert (10);
   S.Insert (0);
   S.Insert (13);

   --  Calling S.Insert(0) now would raise Constraint_Error
   --  because this element is already in the set.
   --  We instead call a version of Insert that doesn't raise an
   --  exception but instead returns a Boolean indicating the status

   S.Insert (0, C, Ins);
   if not Ins then
      Put_Line ("Inserting 0 into set was not successful");
   end if;

   --  We can also call S.Include instead
   --  If the element is already present, the set remains unchanged
   S.Include (0);
   S.Include (13);
   S.Include (14);

   Put_Line ("Set has " & Count_Type'Image (S.Length) & " elements");

   --
   --  Iterate over set using for .. of loop
   --
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("Elements:");
   for E of S loop
       Put_Line ("- " & Integer'Image (E));
   end loop;
end Show_Set_Init;
```

16.2.2 Operations on elements

In this section, we briefly explore the following operations on sets:

- `Delete` and `Exclude` to remove elements;
- `Contains` and `Find` to verify the existence of elements.

To delete elements, you call the procedure `Delete`. However, analogously to the `Insert` procedure above, `Delete` raises an exception if the element to be deleted isn't present in the set. If you want to permit the case where an element might not exist, you can call `Exclude`, which silently ignores any attempt to delete a non-existent element.

`Contains` returns a Boolean value indicating whether a value is contained in the set. `Find` also looks for an element in a set, but returns a cursor to the element or `No_Element` if the element doesn't exist. You can use either function to search for elements in a set.

Let's look at an example that makes use of these operations:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Element_Ops is

   package Integer_Sets is new Ada.Containers.Ordered_Sets
     (Element_Type => Integer);

   use Integer_Sets;

   procedure Show_Elements (S : Set) is
   begin
      New_Line;
      Put_Line ("Set has " & Count_Type'Image (S.Length) & " elements");
      Put_Line ("Elements:");
      for E of S loop
         Put_Line ("- " & Integer'Image (E));
      end loop;
   end Show_Elements;

   S : Set;
begin
   S.Insert (20);
   S.Insert (10);
   S.Insert (0);
   S.Insert (13);

   S.Delete (13);

   --  Calling S.Delete (13) again raises Constraint_Error
   --  because the element is no longer present
   --  in the set, so it can't be deleted.
   --  We can call V.Exclude instead:
```

(continues on next page)

(continued from previous page)

```
   S.Exclude (13);

   if S.Contains (20) then
      Put_Line ("Found element 20 in set");
   end if;

   --  Alternatively, we could use S.Find instead of S.Contains
   if S.Find (0) /= No_Element then
      Put_Line ("Found element 0 in set");
   end if;

   Show_Elements (S);
end Show_Set_Element_Ops;
```

In addition to ordered sets used in the examples above, the standard library also offers hashed sets. The Reference Manual requires the following average complexity of each operation:

Operations	`Ordered_Sets`	`Hashed_Sets`
• Insert • Include • Replace • Delete • Exclude • Find	O($(logN)^2$) or better	$O(logN)$
Subprogram using cursor	O(1)	O(1)

16.2.3 Other Operations

The previous sections mostly dealt with operations on individual elements of a set. But Ada also provides typical set operations: union, intersection, difference and symmetric difference. In contrast to some vector operations we've seen before (e.g. `Merge`), here you can use built-in operators, such as `-`. The following table lists the operations and its associated operator:

Set Operation	Operator
Union	`or`
Intersection	`and`
Difference	`-`
Symmetric difference	`xor`

The following example makes use of these operators:

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Ops is

   package Integer_Sets is new Ada.Containers.Ordered_Sets
     (Element_Type => Integer);

   use Integer_Sets;

   procedure Show_Elements (S : Set) is
```

(continues on next page)

(continued from previous page)

```
   begin
      Put_Line ("Elements:");
      for E of S loop
         Put_Line ("- " & Integer'Image (E));
      end loop;
   end Show_Elements;

   procedure Show_Op (S       : Set;
                      Op_Name : String) is
   begin
      New_Line;
      Put_Line (Op_Name & "(set #1, set #2) has "
                & Count_Type'Image (S.Length) & " elements");
   end Show_Op;

   S1, S2, S3 : Set;
begin
   S1.Insert (0);
   S1.Insert (10);
   S1.Insert (13);

   S2.Insert (0);
   S2.Insert (10);
   S2.Insert (14);

   S3.Insert (0);
   S3.Insert (10);

   New_Line;
   Put_Line ("---- Set #1 ----");
   Show_Elements (S1);

   New_Line;
   Put_Line ("---- Set #2 ----");
   Show_Elements (S2);

   New_Line;
   Put_Line ("---- Set #3 ----");
   Show_Elements (S3);

   New_Line;
   if S3.Is_Subset (S1) then
      Put_Line ("S3 is a subset of S1");
   else
      Put_Line ("S3 is not a subset of S1");
   end if;

   S3 := S1 and S2;
   Show_Op (S3, "Intersection");
   Show_Elements (S3);

   S3 := S1 or S2;
   Show_Op (S3, "Union");
   Show_Elements (S3);

   S3 := S1 - S2;
   Show_Op (S3, "Difference");
   Show_Elements (S3);

   S3 := S1 xor S2;
   Show_Op (S3, "Symmetric difference");
```

(continues on next page)

(continued from previous page)

```
   Show_Elements (S3);

end Show_Set_Ops;
```

16.3 Indefinite maps

The previous sections presented containers for elements of definite types. Although most examples in those sections presented `Integer` types as element type of the containers, containers can also be used with indefinite types, an example of which is the `String` type. However, indefinite types require a different kind of containers designed specially for them.

We'll also be exploring a different class of containers: maps. They associate a key with a specific value. An example of a map is the one-to-one association between a person and their age. If we consider a person's name to be the key, the value is the person's age.

16.3.1 Hashed maps

Hashed maps are maps that make use of a hash as a key. The hash itself is calculated by a function you provide.

In other languages

Hashed maps are similar to dictionaries in Python and hashes in Perl. One of the main differences is that these scripting languages allow using different types for the values contained in a single map, while in Ada, both the type of key and value are specified in the package instantiation and remains constant for that specific map. You can't have a map where two elements are of different types or two keys are of different types. If you want to use multiple types, you must create a different map for each and use only one type in each map.

When instantiating a hashed map from `Ada.Containers.Indefinite_Hashed_Maps`, we specify following elements:

- `Key_Type`: type of the key
- `Element_Type`: type of the element
- `Hash`: hash function for the `Key_Type`
- `Equivalent_Keys`: an equality operator (e.g. `=`) that indicates whether two keys are to be considered equal.
  - If the type specified in `Key_Type` has a standard operator, you can use it, which you do by specifing using that operator as the value of `Equivalent_Keys`.

In the next example, we'll use a string as a key type. We'll use the `Hash` function provided by the standard library for strings (in the `Ada.Strings` package) and the standard equality operator.

You add elements to a hashed map by calling `Insert`. If an element is already contained in a map `M`, you can access it directly by using its key. For example, you can change the value of an element by calling `M ("My_Key") := 10`. If the key is not found, an exception is raised. To verify if a key is available, use the function `Contains` (as we've seen above in the section on sets).

Let's see an example:

```
with Ada.Containers.Indefinite_Hashed_Maps;
with Ada.Strings.Hash;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Hashed_Map is

   package Integer_Hashed_Maps is new
     Ada.Containers.Indefinite_Hashed_Maps
       (Key_Type        => String,
        Element_Type    => Integer,
        Hash            => Ada.Strings.Hash,
        Equivalent_Keys => "=");

   use Integer_Hashed_Maps;

   M : Map;
   --  Same as:  M : Integer_Hashed_Maps.Map;
begin
   M.Include ("Alice", 24);
   M.Include ("John",  40);
   M.Include ("Bob",   28);

   if M.Contains ("Alice") then
      Put_Line ("Alice's age is "
                & Integer'Image (M ("Alice")));
   end if;

   --  Update Alice's age
   --  Key must already exist in M.
   --  Otherwise an exception is raised.
   M ("Alice") := 25;

   New_Line; Put_Line ("Name & Age:");
   for C in M.Iterate loop
      Put_Line (Key (C) & ": " & Integer'Image (M (C)));
   end loop;

end Show_Hashed_Map;
```

16.3.2 Ordered maps

Ordered maps share many features with hashed maps. The main differences are:

- A hash function isn't needed. Instead, you must provide an ordering function (< operator), which the ordered map will use to order elements and allow fast access, $O(logn)$, using a binary search.
  - If the type specified in `Key_Type` has a standard < operator, you can use it in a similar way as we did for `Equivalent_Keys` above for hashed maps.

Let's see an example:

```
with Ada.Containers.Indefinite_Ordered_Maps;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Ordered_Map is

   package Integer_Ordered_Maps is new
```

(continues on next page)

(continued from previous page)

```
     Ada.Containers.Indefinite_Ordered_Maps
       (Key_Type        => String,
        Element_Type    => Integer);

   use Integer_Ordered_Maps;

   M : Map;
begin
   M.Include ("Alice", 24);
   M.Include ("John",  40);
   M.Include ("Bob",   28);

   if M.Contains ("Alice") then
      Put_Line ("Alice's age is "
                & Integer'Image (M ("Alice")));
   end if;

   --  Update Alice's age
   --  Key must already exist in M
   M ("Alice") := 25;

   New_Line; Put_Line ("Name & Age:");
   for C in M.Iterate loop
      Put_Line (Key (C) & ": " & Integer'Image (M (C)));
   end loop;

end Show_Ordered_Map;
```

You can see a great similarity between the examples above and from the previous section. In fact, since both kinds of maps share many operations, we didn't need to make extensive modifications when we changed our example to use ordered maps instead of hashed maps. The main difference is seen when we run the examples: the output of a hashed map is usually unordered, but the output of a ordered map is always ordered, as implied by its name.

16.3.3 Complexity

Hashed maps are generally the fastest data structure available to you in Ada if you need to associate heterogeneous keys to values and search for them quickly. In most cases, they are slightly faster than ordered maps. So if you don't need ordering, use hashed maps.

The Reference Manual requires the following average complexity of operations:

Operations	`Ordered_Maps`	`Hashed_Maps`
• Insert • Include • Replace • Delete • Exclude • Find	O($(logN)^2$) or better	$O(logN)$
Subprogram using cursor	O(1)	O(1)

CHAPTER

SEVENTEEN

STANDARD LIBRARY: DATES & TIMES

The standard library supports processing of dates and times using two approaches:

- *Calendar* approach, which is suitable for handling dates and times in general;
- *Real-time* approach, which is better suited for real-time applications that require enhanced precision — for example, by having access to an absolute clock and handling time spans. Note that this approach only supports times, not dates.

The following sections present these two approaches.

17.1 Date and time handling

The `Ada.Calendar` package supports handling of dates and times. Let's look at a simple example:

```
with Ada.Calendar;            use Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Ada.Text_IO;             use Ada.Text_IO;

procedure Display_Current_Time is
   Now : Time := Clock;
begin
   Put_Line ("Current time: " & Image (Now));
end Display_Current_Time;
```

This example displays the current date and time, which is retrieved by a call to the `Clock` function. We call the function `Image` from the `Ada.Calendar.Formatting` package to get a `String` for the current date and time. We could instead retrieve each component using the `Split` function. For example:

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

procedure Display_Current_Year is
   Now         : Time := Clock;

   Now_Year    : Year_Number;
   Now_Month   : Month_Number;
   Now_Day     : Day_Number;
   Now_Seconds : Day_Duration;
begin
   Split (Now,
          Now_Year,
          Now_Month,
          Now_Day,
          Now_Seconds);
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("Current year  is: " & Year_Number'Image (Now_Year));
   Put_Line ("Current month is: " & Month_Number'Image (Now_Month));
   Put_Line ("Current day   is: " & Day_Number'Image (Now_Day));
end Display_Current_Year;
```

Here, we're retrieving each element and displaying it separately.

17.1.1 Delaying using date

You can delay an application so that it restarts at a specific date and time. We saw something similar in the chapter on tasking. You do this using a `delay until` statement. For example:

```
with Ada.Calendar;            use Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Ada.Calendar.Time_Zones; use Ada.Calendar.Time_Zones;
with Ada.Text_IO;             use Ada.Text_IO;

procedure Display_Delay_Next_Specific_Time is
   TZ   : Time_Offset := UTC_Time_Offset;
   Next : Time        := Ada.Calendar.Formatting.Time_Of
     (Year        => 2018,
      Month       => 5,
      Day         => 1,
      Hour        => 15,
      Minute      => 0,
      Second      => 0,
      Sub_Second  => 0.0,
      Leap_Second => False,
      Time_Zone   => TZ);

   --  Next = 2018-05-01 15:00:00.00 (local time-zone)
begin
   Put_Line ("Let's wait until...");
   Put_Line (Image (Next, True, TZ));

   delay until Next;

   Put_Line ("Enough waiting!");
end Display_Delay_Next_Specific_Time;
```

In this example, we specify the date and time by initializing `Next` using a call to `Time_Of`, a function taking the various components of a date (year, month, etc) and returning an element of the `Time` type. Because the date specified is in the past, the `delay until` statement won't produce any noticeable effect. However, if we passed a date in the future, the program would wait until that specific date and time arrived.

Here we're converting the time to the local timezone. If we don't specify a timezone, *Coordinated Universal Time* (abbreviated to UTC) is used by default. By retrieving the time offset to UTC with a call to `UTC_Time_Offset` from the `Ada.Calendar.Time_Zones` package, we can initialize `TZ` and use it in the call to `Time_Of`. This is all we need do to make the information provided to `Time_Of` relative to the local time zone.

We could achieve a similar result by initializing `Next` with a `String`. We can do this with a call to `Value` from the `Ada.Calendar.Formatting` package. This is the modified code:

```
with Ada.Calendar;            use Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Ada.Calendar.Time_Zones; use Ada.Calendar.Time_Zones;
with Ada.Text_IO;             use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```
procedure Display_Delay_Next_Specific_Time is
   TZ   : Time_Offset := UTC_Time_Offset;
   Next : Time        := Ada.Calendar.Formatting.Value
     ("2018-05-01 15:00:00.00", TZ);

   --  Next = 2018-05-01 15:00:00.00 (local time-zone)
begin
   Put_Line ("Let's wait until...");
   Put_Line (Image (Next, True, TZ));

   delay until Next;

   Put_Line ("Enough waiting!");
end Display_Delay_Next_Specific_Time;
```

In this example, we're again using `TZ` in the call to `Value` to adjust the input time to the current time zone.

In the examples above, we were delaying to a specific date and time. Just like we saw in the tasking chapter, we could instead specify the delay relative to the current time. For example, we could delay by 5 seconds, using the current time:

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

procedure Display_Delay_Next is
   D    : Duration := 5.0;       --  seconds
   Now  : Time     := Clock;
   Next : Time     := Now + D;   --  use duration to
                                 --  specify next point in time
begin
   Put_Line ("Let's wait "
             & Duration'Image (D) & " seconds...");
   delay until Next;
   Put_Line ("Enough waiting!");
end Display_Delay_Next;
```

Here, we're specifying a duration of 5 seconds in `D`, adding it to the current time from `Now`, and storing the sum in `Next`. We then use it in the `delay until` statement.

17.2 Real-time

In addition to `Ada.Calendar`, the standard library also supports time operations for real-time applications. These are included in the `Ada.Real_Time` package. This package also include a `Time` type. However, in the `Ada.Real_Time` package, the `Time` type is used to represent an absolute clock and handle a time span. This contrasts with the `Ada.Calendar`, which uses the `Time` type to represent dates and times.

In the previous section, we used the `Time` type from the `Ada.Calendar` and the `delay until` statement to delay an application by 5 seconds. We could have used the `Ada.Real_Time` package instead. Let's modify that example:

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

procedure Display_Delay_Next_Real_Time is
   D     : Time_Span := Seconds (5);
```

(continues on next page)

(continued from previous page)

```
   Next  : Time      := Clock + D;
begin
   Put_Line ("Let's wait "
             & Duration'Image (To_Duration (D)) & " seconds...");
   delay until Next;
   Put_Line ("Enough waiting!");
end Display_Delay_Next_Real_Time;
```

The main difference is that `D` is now a variable of type `Time_Span`, defined in the `Ada.Real_Time` package. We call the function `Seconds` to initialize `D`, but could have gotten a finer granularity by calling `Nanoseconds` instead. Also, we need to first convert `D` to the `Duration` type using `To_Duration` before we can display it.

17.2.1 Benchmarking

One interesting application using the `Ada.Real_Time` package is benchmarking. We've used that package before in a previous section when discussing tasking. Let's look at an example of benchmarking:

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

procedure Display_Benchmarking is

   procedure Computational_Intensive_App is
   begin
      delay 5.0;
   end Computational_Intensive_App;

   Start_Time, Stop_Time : Time;
   Elapsed_Time          : Time_Span;

begin
   Start_Time := Clock;

   Computational_Intensive_App;

   Stop_Time    := Clock;
   Elapsed_Time := Stop_Time - Start_Time;

   Put_Line ("Elapsed time: "
             & Duration'Image (To_Duration (Elapsed_Time))
             & " seconds");
end Display_Benchmarking;
```

This example defines a dummy `Computational_Intensive_App` implemented using a simple `delay` statement. We initialize `Start_Time` and `Stop_Time` from the then-current clock and calculate the elapsed time. By running this program, we see that the time is roughly 5 seconds, which is expected due to the `delay` statement.

A similar application is benchmarking of CPU time. We can implement this using the `Execution_Time` package. Let's modify the previous example to measure CPU time:

```
with Ada.Text_IO;        use Ada.Text_IO;
with Ada.Real_Time;      use Ada.Real_Time;
with Ada.Execution_Time; use Ada.Execution_Time;

procedure Display_Benchmarking_CPU_Time is
```

(continues on next page)

(continued from previous page)

```
   procedure Computational_Intensive_App is
   begin
      delay 5.0;
   end Computational_Intensive_App;

   Start_Time, Stop_Time : CPU_Time;
   Elapsed_Time          : Time_Span;

begin
   Start_Time := Clock;

   Computational_Intensive_App;

   Stop_Time    := Clock;
   Elapsed_Time := Stop_Time - Start_Time;

   Put_Line ("CPU time: "
             & Duration'Image (To_Duration (Elapsed_Time))
             & " seconds");
end Display_Benchmarking_CPU_Time;
```

In this example, `Start_Time` and `Stop_Time` are of type `CPU_Time` instead of `Time`. However, we still call the `Clock` function to initialize both variables and calculate the elapsed time in the same way as before. By running this program, we see that the CPU time is significantly lower than the 5 seconds we've seen before. This is because the `delay` statement doesn't require much CPU time. The results will be different if we change the implementation of `Computational_Intensive_App` to use a mathematical functions in a long loop. For example:

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Real_Time;       use Ada.Real_Time;
with Ada.Execution_Time; use Ada.Execution_Time;

with Ada.Numerics.Generic_Elementary_Functions;

procedure Display_Benchmarking_Math is

   procedure Computational_Intensive_App is
      package Funcs is new Ada.Numerics.Generic_Elementary_Functions
        (Float_Type => Long_Long_Float);
      use Funcs;

      X : Long_Long_Float;
   begin
      for I in 0 .. 1_000_000 loop
         X := Tan (Arctan
                   (Tan (Arctan
                      (Tan (Arctan
                         (Tan (Arctan
                            (Tan (Arctan
                               (Tan (Arctan
                                  (0.577))))))))))));
      end loop;
   end Computational_Intensive_App;

   procedure Benchm_Elapsed_Time is
      Start_Time, Stop_Time : Time;
      Elapsed_Time          : Time_Span;

   begin
      Start_Time := Clock;
```

(continues on next page)

(continued from previous page)

```
      Computational_Intensive_App;

      Stop_Time    := Clock;
      Elapsed_Time := Stop_Time - Start_Time;

      Put_Line ("Elapsed time: "
                & Duration'Image (To_Duration (Elapsed_Time))
                & " seconds");
   end Benchm_Elapsed_Time;

   procedure Benchm_CPU_Time is
      Start_Time, Stop_Time : CPU_Time;
      Elapsed_Time          : Time_Span;

   begin
      Start_Time := Clock;

      Computational_Intensive_App;

      Stop_Time    := Clock;
      Elapsed_Time := Stop_Time - Start_Time;

      Put_Line ("CPU time: "
                & Duration'Image (To_Duration (Elapsed_Time))
                & " seconds");
   end Benchm_CPU_Time;
begin
   Benchm_Elapsed_Time;
   Benchm_CPU_Time;
end Display_Benchmarking_Math;
```

Now that our dummy `Computational_Intensive_App` involves mathematical operations requiring significant CPU time, the measured elapsed and CPU time are much closer to each other than before.

CHAPTER

EIGHTEEN

STANDARD LIBRARY: STRINGS

In previous chapters, we've seen source-code examples using the `String` type, which is a fixed-length string type — essentialy, it's an array of characters. In many cases, this data type is good enough to deal with textual information. However, there are situations that require more advanced text processing. Ada offers alternative approaches for these cases:

- *Bounded strings*: similar to fixed-length strings, bounded strings have a maximum length, which is set at its instantiation. However, bounded strings are not arrays of characters. At any time, they can contain a string of varied length — provided this length is below or equal to the maximum length.
- *Unbounded strings*: similar to bounded strings, unbounded strings can contain strings of varied length. However, in addition to that, they don't have a maximum length. In this sense, they are very flexible.

The following sections present an overview of the different string types and common operations for string types.

18.1 String operations

Operations on standard (fixed-length) strings are available in the `Ada.Strings.Fixed` package. As mentioned previously, standard strings are arrays of elements of `Character` type with *a fixed-length*. That's why this child package is called `Fixed`.

One of the simplest operations provided is counting the number of substrings available in a string (`Count`) and finding their corresponding indices (`Index`). Let's look at an example:

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure Show_Find_Substring is

   S   : String := "Hello" & 3 * " World";
   P   : constant String := "World";
   Idx : Natural;
   Cnt : Natural;
begin
   Cnt := Ada.Strings.Fixed.Count
     (Source  => S,
      Pattern => P);

   Put_Line ("String: " & S);
   Put_Line ("Count for '" & P & "': " & Natural'Image (Cnt));

   Idx := 0;
   for I in 1 .. Cnt loop
      Idx := Index
```

(continues on next page)

(continued from previous page)

```
        (Source  => S,
         Pattern => P,
         From    => Idx + 1);

      Put_Line ("Found instance of '" & P & "' at position: "
                & Natural'Image (Idx));
   end loop;

end Show_Find_Substring;
```

We initialize the string `S` using a multiplication. Writing `"Hello" & 3 * " World"` creates the string `Hello World World World`. We then call the function *Count*' to get the number of instances of the word `World` in `S`. Next we call the function `Index` in a loop to find the index of each instance of `World` in `S`.

That example looked for instances of a specific substring. In the next example, we retrieve all the words in the string. We do this using `Find_Token` and specifying whitespaces as separators. For example:

```
with Ada.Strings;       use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Strings.Maps;  use Ada.Strings.Maps;
with Ada.Text_IO;       use Ada.Text_IO;

procedure Show_Find_Words is

   S    : String := "Hello" & 3 * " World";
   F    : Positive;
   L    : Natural;
   I    : Natural := 1;

   Whitespace : constant Character_Set :=
     To_Set (' ');
begin
   Put_Line ("String: " & S);
   Put_Line ("String length: " & Integer'Image (S'Length));

   while I in S'Range loop
      Find_Token
        (Source  => S,
         Set     => Whitespace,
         From    => I,
         Test    => Outside,
         First   => F,
         Last    => L);

      exit when L = 0;

      Put_Line ("Found word instance at position "
                & Natural'Image (F)
                & ": '" & S (F .. L) & "'");
      --   & "-" & F'Img & "-" & L'Img

      I := L + 1;
   end loop;
end Show_Find_Words;
```

We pass a set of characters to be used as delimitators to the procedure `Find_Token`. This set is a member of the `Character_Set` type from the `Ada.Strings.Maps` package. We call the `To_Set` function (from the same package) to initialize the set to `Whitespace` and then call `Find_Token` to loop over each valid index and find the starting index of each word. We pass `Outside` to the

`Test` parameter of the `Find_Token` procedure to indicate that we're looking for indices that are outside the `Whitespace` set, i.e. actual words. The `First` and `Last` parameters of `Find_Token` are output parameters that indicate the valid range of the substring. We use this information to display the string (`S (F .. L)`).

The operations we've looked at so far read strings, but don't modify them. We next discuss operations that change the content of strings:

Operation	Description
Insert	Insert substring in a string
Overwrite	Overwrite a string with a substring
Delete	Delete a substring
Trim	Remove whitespaces from a string

All these operations are available both as functions or procedures. Functions create a new string but procedures perform the operations in place. The procedure will raise an exception if the constraints of the string are not satisfied. For example, if we have a string S containing 10 characters, inserting a string with two characters (e.g. `"!!"`) into it produces a string containing 12 characters. Since it has a fixed length, we can't increase its size. One possible solution in this case is to specify that truncation should be applied while inserting the substring. This keeps the length of S fixed. Let's see an example that makes use of both function and procedure versions of `Insert`, `Overwrite`, and `Delete`:

```
with Ada.Strings;       use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure Show_Adapted_Strings is

   S   : String := "Hello World";
   P   : constant String := "World";
   N   : constant String := "Beautiful";

   procedure Display_Adapted_String
     (Source   : String;
      Before   : Positive;
      New_Item : String;
      Pattern  : String)
   is
      S_Ins_In : String := Source;
      S_Ovr_In : String := Source;
      S_Del_In : String := Source;

      S_Ins : String := Insert (Source, Before, New_Item & " ");
      S_Ovr : String := Overwrite (Source, Before, New_Item);
      S_Del : String := Trim (Delete (Source,
                                      Before,
                                      Before + Pattern'Length - 1),
                              Ada.Strings.Right);
   begin
      Insert (S_Ins_In,    Before, New_Item, Right);
      Overwrite (S_Ovr_In, Before, New_Item, Right);
      Delete (S_Del_In,    Before, Before + Pattern'Length - 1);

      Put_Line ("Original:  '" & Source & "'");

      Put_Line ("Insert:    '" & S_Ins  & "'");
      Put_Line ("Overwrite: '" & S_Ovr  & "'");
      Put_Line ("Delete:    '" & S_Del  & "'");

      Put_Line ("Insert    (in-place): '" & S_Ins_In & "'");
```

(continues on next page)

(continued from previous page)

```
      Put_Line ("Overwrite (in-place): '" & S_Ovr_In & "'");
      Put_Line ("Delete    (in-place): '" & S_Del_In & "'");
   end Display_Adapted_String;

   Idx : Natural;
begin
   Idx := Index
     (Source  => S,
      Pattern => P);

   if Idx > 0 then
      Display_Adapted_String (S, Idx, N, P);
   end if;
end Show_Adapted_Strings;
```

In this example, we look for the index of the substring `World` and perform operations on this substring within the outer string. The procedure `Display_Adapted_String` uses both versions of the operations. For the procedural version of `Insert` and `Overwrite`, we apply truncation to the right side of the string (`Right`). For the `Delete` procedure, we specify the range of the substring, which is replaced by whitespaces. For the function version of `Delete`, we also call `Trim` which trims the trailing whitespace.

18.2 Limitation of fixed-length strings

Using fixed-length strings is usually good enough for strings that are initialized when they are declared. However, as seen in the previous section, procedural operations on strings cause difficulties when done on fixed-length strings because fixed-length strings are arrays of characters. The following example shows how cumbersome the initialization of fixed-length strings can be when it's not performed in the declaration:

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

procedure Show_Char_Array is
   S : String (1 .. 15);
   --  Strings are arrays of Character
begin
   S := "Hello          ";
   --  Alternatively:
   --
   --  #1:
   --      S (1 .. 5)      := "Hello";
   --      S (6 .. S'Last) := (others => ' ');
   --
   --  #2:
   --      S := ('H', 'e', 'l', 'l', 'o', others => ' ');

   Put_Line ("String: " & S);
   Put_Line ("String Length: " & Integer'Image (S'Length));
end Show_Char_Array;
```

In this case, we can't simply write `S := "Hello"` because the resulting array of characters for the `Hello` constant has a different length than the `S` string. Therefore, we need to include trailing whitespaces to match the length of `S`. As shown in the example, we could use an exact range for the initialization (`S (1 .. 5)`) or use an explicit array of individual characters.

When strings are initialized or manipulated at run-time, it's usually better to use bounded or unbounded strings. An important feature of these types is that they aren't arrays, so the difficulties presented above don't apply. Let's start with bounded strings.

18.3 Bounded strings

Bounded strings are defined in the `Ada.Strings.Bounded.Generic_Bounded_Length` package. Because this is a generic package, you need to instantiate it and set the maximum length of the bounded string. You can then declare bounded strings of the `Bounded_String` type.

Both bounded and fixed-length strings have a maximum length that they can hold. However, bounded strings are not arrays, so initializing them at run-time is much easier. For example:

```
with Ada.Strings;          use Ada.Strings;
with Ada.Strings.Bounded;
with Ada.Text_IO;          use Ada.Text_IO;

procedure Show_Bounded_String is
   package B_Str is new
     Ada.Strings.Bounded.Generic_Bounded_Length (Max => 15);
   use B_Str;

   S1, S2 : Bounded_String;

   procedure Display_String_Info (S : Bounded_String) is
   begin
      Put_Line ("String: " & To_String (S));
      Put_Line ("String Length: " & Integer'Image (Length (S)));
      --  String:         S'Length => ok
      --  Bounded_String: S'Length => compilation error
      --                              bounded strings are not arrays!

      Put_Line ("Max.   Length: " & Integer'Image (Max_Length));
   end Display_String_Info;
begin
   S1 := To_Bounded_String ("Hello");
   Display_String_Info (S1);

   S2 := To_Bounded_String ("Hello World");
   Display_String_Info (S2);

   S1 := To_Bounded_String ("Something longer to say here...",
                            Right);
   Display_String_Info (S1);
end Show_Bounded_String;
```

By using bounded strings, we can easily assign to `S1` and `S2` multiple times during execution. We use the `To_Bounded_String` and `To_String` functions to convert, in the respective direction, between fixed-length and bounded strings. A call to `To_Bounded_String` raises an exception if the length of the input string is greater than the maximum capacity of the bounded string. To avoid this, we can use the truncation parameter (`Right` in our example).

Bounded strings are not arrays, so we can't use the `'Length` attribute as we did for fixed-length strings. Instead, we call the `Length` function, which returns the length of the bounded string. The `Max_Length` constant represents the maximum length of the bounded string that we set when we instantiated the package.

After initializing a bounded string, we can manipulate it. For example, we can append a string to a bounded string using `Append` or concatenate bounded strings using the `&` operator. Like so:

```
with Ada.Strings;          use Ada.Strings;
with Ada.Strings.Bounded;
with Ada.Text_IO;          use Ada.Text_IO;

procedure Show_Bounded_String_Op is
```

(continues on next page)

(continued from previous page)

```
   package B_Str is new
     Ada.Strings.Bounded.Generic_Bounded_Length (Max => 30);
   use B_Str;

   S1, S2 : Bounded_String;
begin
   S1 := To_Bounded_String ("Hello");
   --  Alternatively: A := Null_Bounded_String & "Hello";
   Append (S1, " World");
   --  Alternatively: Append (A, " World", Right);
   Put_Line ("String: " & To_String (S1));

   S2 := To_Bounded_String ("Hello!");
   S1 := S1 & " " & S2;
   Put_Line ("String: " & To_String (S1));
end Show_Bounded_String_Op;
```

We can initialize a bounded string with an empty string using the `Null_Bounded_String` constant. Also, we can use the `Append` procedure and specify the truncation mode like we do with the `To_Bounded_String` function.

18.4 Unbounded strings

Unbounded strings are defined in the `Ada.Strings.Unbounded` package. This is *not* a generic package, so we don't need to instantiate it before using the `Unbounded_String` type. As you may recall from the previous section, bounded strings require a package instantiation.

Unbounded strings are similar to bounded strings. The main difference is that they can hold strings of any size and adjust according to the input string: if we assign, e.g., a 10-character string to an unbounded string and later assign a 50-character string, internal operations in the container ensure that memory is allocated to store the new string. In most cases, developers don't need to worry about these operations. Also, no truncation is necessary.

Initialization of unbounded strings is very similar to bounded strings. Let's look at an example:

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

procedure Show_Unbounded_String is
   S1, S2 : Unbounded_String;

   procedure Display_String_Info (S : Unbounded_String) is
   begin
      Put_Line ("String: " & To_String (S));
      Put_Line ("String Length: " & Integer'Image (Length (S)));
   end Display_String_Info;
begin
   S1 := To_Unbounded_String ("Hello");
   --  Alternatively: A := Null_Unbounded_String & "Hello";
   Display_String_Info (S1);

   S2 := To_Unbounded_String ("Hello World");
   Display_String_Info (S2);

   S1 := To_Unbounded_String ("Something longer to say here...");
   Display_String_Info (S1);
end Show_Unbounded_String;
```

Like bounded strings, we can assign to S1 and S2 multiple times during execution and use the To_Unbounded_String and To_String functions to convert back-and-forth between fixed-length strings and unbounded strings. However, in this case, truncation is not needed.

And, just like for bounded strings, you can use the Append function and the & operator for unbounded strings. For example:

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

procedure Show_Unbounded_String_Op is
   S1, S2 : Unbounded_String := Null_Unbounded_String;
begin
   S1 := S1 & "Hello";
   S2 := S2 & "Hello!";

   Append (S1, " World");
   Put_Line ("String: " & To_String (S1));

   S1 := S1 & " " & S2;
   Put_Line ("String: " & To_String (S1));
end Show_Unbounded_String_Op;
```

CHAPTER

NINETEEN

STANDARD LIBRARY: FILES AND STREAMS

Ada provides different approaches for file input/output (I/O):

- *Text I/O*, which supports file I/O in text format, including the display of information on the console.
- *Sequential I/O*, which supports file I/O in binary format written in a sequential fashion for a specific data type.
- *Direct I/O*, which supports file I/O in binary format for a specific data type, but also supporting access to any position of a file.
- *Stream I/O*, which supports I/O of information for multiple data types, including objects of unbounded types, using files in binary format.

This table presents a summary of the features we've just seen:

File I/O option	Format	Random access	Data types
Text I/O	text		string type
Sequential I/O	binary		single type
Direct I/O	binary	X	single type
Stream I/O	binary	X	multiple types

In the following sections, we discuss details about these I/O approaches.

19.1 Text I/O

In most parts of this course, we used the `Put_Line` procedure to display information on the console. However, this procedure also accepts a `File_Type` parameter. For example, you can select between standard output and standard error by setting this parameter explicitly:

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

procedure Show_Std_Text_Out is
begin
   Put_Line (Standard_Output, "Hello World #1");
   Put_Line (Standard_Error,  "Hello World #2");
end Show_Std_Text_Out;
```

You can also use this parameter to write information to any text file. To create a new file for writing, use the `Create` procedure, which initializes a `File_Type` element that you can later pass to `Put_Line` (instead of, e.g., `Standard_Output`). After you finish writing information, you can close the file by calling the `Close` procedure.

You use a similar method to read information from a text file. However, when opening the file, you must specify that it's an input file (`In_File`) instead of an output file. Also, instead of calling the `Put_Line` procedure, you call the `Get_Line` function to read information from the file.

Let's see an example that writes information into a new text file and then reads it back from the same file:

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

procedure Show_Simple_Text_File_IO is
   F         : File_Type;
   File_Name : constant String := "simple.txt";
begin
   Create (F, Out_File, File_Name);
   Put_Line (F, "Hello World #1");
   Put_Line (F, "Hello World #2");
   Put_Line (F, "Hello World #3");
   Close (F);

   Open (F, In_File, File_Name);
   while not End_Of_File (F) loop
      Put_Line (Get_Line (F));
   end loop;
   Close (F);
end Show_Simple_Text_File_IO;
```

In addition to the `Create` and `Close` procedures, the standard library also includes a `Reset` procedure, which, as the name implies, resets (erases) all the information from the file. For example:

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

procedure Show_Text_File_Reset is
   F         : File_Type;
   File_Name : constant String := "simple.txt";
begin
   Create (F, Out_File, File_Name);
   Put_Line (F, "Hello World #1");
   Reset (F);
   Put_Line (F, "Hello World #2");
   Close (F);

   Open (F, In_File, File_Name);
   while not End_Of_File (F) loop
      Put_Line (Get_Line (F));
   end loop;
   Close (F);
end Show_Text_File_Reset;
```

By running this program, we notice that, although we've written the first string (`Hello World #1`) to the file, it has been erased because of the call to `Reset`.

In addition to opening a file for reading or writing, you can also open an existing file and append to it. Do this by calling the `Open` procedure with the `Append_File` option.

When calling the `Open` procedure, an exception is raised if the specified file isn't found. Therefore, you should handle exceptions in that context. The following example deletes a file and then tries to open the same file for reading:

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

procedure Show_Text_File_Input_Except is
   F         : File_Type;
   File_Name : constant String := "simple.txt";
begin
   --  Open output file and delete it
   Create (F, Out_File, File_Name);
   Delete (F);
```

(continues on next page)

(continued from previous page)

```
   --  Try to open deleted file
   Open (F, In_File, File_Name);
   Close (F);
exception
   when Name_Error =>
      Put_Line ("File does not exist");
   when others =>
      Put_Line ("Error while processing input file");
end Show_Text_File_Input_Except;
```

In this example, we create the file by calling `Create` and then delete it by calling `Delete`. After the call to `Delete`, we can no longer use the `File_Type` element'. After deleting the file, we try to open the non-existent file, which raises a `Name_Error` exception.

19.2 Sequential I/O

The previous section presented details about text file I/O. Here, we discuss doing file I/O in binary format. The first package we'll explore is the `Ada.Sequential_IO` package. Because this package is a generic package, you need to instantiate it for the data type you want to use for file I/O. Once you've done that, you can use the same procedures we've seen in the previous section: `Create`, `Open`, `Close`, `Reset` and `Delete`. However, instead of calling the `Get_Line` and `Put_Line` procedures, you'd call the `Read` and `Write` procedures.

In the following example, we instantiate the `Ada.Sequential_IO` package for floating-point types:

```
with Ada.Text_IO;
with Ada.Sequential_IO;

procedure Show_Seq_Float_IO is
   package Float_IO is new Ada.Sequential_IO (Float);
   use Float_IO;

   F         : Float_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   Write (F,  1.5);
   Write (F,  2.4);
   Write (F,  6.7);
   Close (F);

   declare
      Value : Float;
   begin
      Open (F, In_File, File_Name);
      while not End_Of_File (F) loop
         Read (F, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
      Close (F);
   end;
end Show_Seq_Float_IO;
```

We use the same approach to read and write complex information. The following example uses a record that includes a Boolean and a floating-point value:

```
with Ada.Text_IO;
with Ada.Sequential_IO;

procedure Show_Seq_Rec_IO is
   type Num_Info is record
      Valid : Boolean := False;
      Value : Float;
   end record;

   procedure Put_Line (N : Num_Info) is
   begin
      if N.Valid then
         Ada.Text_IO.Put_Line ("(ok,     " & Float'Image (N.Value) & ")");
      else
         Ada.Text_IO.Put_Line ("(not ok,  ----------)");
      end if;
   end Put_Line;

   package Num_Info_IO is new Ada.Sequential_IO (Num_Info);
   use Num_Info_IO;

   F         : Num_Info_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   Write (F,  (True,  1.5));
   Write (F,  (False, 2.4));
   Write (F,  (True,  6.7));
   Close (F);

   declare
      Value : Num_Info;
   begin
      Open (F, In_File, File_Name);
      while not End_Of_File (F) loop
         Read (F, Value);
         Put_Line (Value);
      end loop;
      Close (F);
   end;
end Show_Seq_Rec_IO;
```

As the example shows, we can use the same approach we used for floating-point types to perform file I/O for this record. Once we instantiate the `Ada.Sequential_IO` package for the record type, file I/O operations are performed the same way.

19.3 Direct I/O

Direct I/O is available in the `Ada.Direct_IO` package. This mechanism is similar to the sequential I/O approach just presented, but allows us to access any position in the file. The package instantiation and most operations are very similar to sequential I/O. To rewrite the `Show_Seq_Float_IO` application presented in the previous section to use the `Ada.Direct_IO` package, we just need to replace the instances of the `Ada.Sequential_IO` package by the `Ada.Direct_IO` package. This is the new source code:

```
with Ada.Text_IO;
with Ada.Direct_IO;
```

(continues on next page)

(continued from previous page)

```
procedure Show_Dir_Float_IO is
   package Float_IO is new Ada.Direct_IO (Float);
   use Float_IO;

   F         : Float_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   Write (F,  1.5);
   Write (F,  2.4);
   Write (F,  6.7);
   Close (F);

   declare
      Value : Float;
   begin
      Open (F, In_File, File_Name);
      while not End_Of_File (F) loop
         Read (F, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
      Close (F);
   end;
end Show_Dir_Float_IO;
```

Unlike sequential I/O, direct I/O allows you to access any position in the file. However, it doesn't offer an option to append information to a file. Instead, it provides an `Inout_File` mode allowing reading and writing to a file via the same `File_Type` element.

To access any position in the file, call the `Set_Index` procedure to set the new position / index. You can use the `Index` function to retrieve the current index. Let's see an example:

```
with Ada.Text_IO;
with Ada.Direct_IO;

procedure Show_Dir_Float_In_Out_File is
   package Float_IO is new Ada.Direct_IO (Float);
   use Float_IO;

   F         : Float_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   --  Open file for input / output
   Create (F, Inout_File, File_Name);
   Write (F,  1.5);
   Write (F,  2.4);
   Write (F,  6.7);

   --  Set index to previous position and overwrite value
   Set_Index (F, Index (F) - 1);
   Write (F,  7.7);

   declare
      Value : Float;
   begin
      --  Set index to start of file
      Set_Index (F, 1);

      while not End_Of_File (F) loop
         Read (F, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
```

(continues on next page)

(continued from previous page)

```
      end loop;
      Close (F);
   end;
end Show_Dir_Float_In_Out_File;
```

By running this example, we see that the file contains `7.7`, rather than the previous `6.7` that we wrote. We overwrote the value by changing the index to the previous position before doing another write.

In this example we used the `Inout_File` mode. Using that mode, we just changed the index back to the initial position before reading from the file (`Set_Index (F, 1)`) instead of closing the file and reopening it for reading.

19.4 Stream I/O

All the previous approaches for file I/O in binary format (sequential and direct I/O) are specific for a single data type (the one we instantiate them with). You can use these approaches to write objects of a single data type that may be an array or record (potentially with many fields), but if you need to create and process files that include different data types, or any objects of an unbounded type, these approaches are not sufficient. Instead, you should use stream I/O.

Stream I/O shares some similarities with the previous approaches. We still use the `Create`, `Open` and `Close` procedures. However, instead of accessing the file directly via a `File_Type` element, you use a `Stream_Access` element. To read and write information, you use the `'Read` or `'Write` attributes of the data types you're reading or writing.

Let's look at a version of the `Show_Dir_Float_IO` procedure from the previous section that makes use of stream I/O instead of direct I/O:

```
with Ada.Text_IO;
with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;

procedure Show_Float_Stream is
   F         : File_Type;
   S         : Stream_Access;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   S := Stream (F);

   Float'Write (S, 1.5);
   Float'Write (S, 2.4);
   Float'Write (S, 6.7);

   Close (F);

   declare
      Value : Float;
   begin
      Open (F, In_File, File_Name);
      S := Stream (F);

      while not End_Of_File (F) loop
         Float'Read (S, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
      Close (F);
   end;
end Show_Float_Stream;
```

After the call to `Create`, we retrieve the corresponding `Stream_Access` element by calling the `Stream` function. We then use this stream to write information to the file via the `'Write` attribute of the `Float` type. After closing the file and reopening it for reading, we again retrieve the corresponding `Stream_Access` element and processed to read information from the file via the `'Read` attribute of the `Float` type.

You can use streams to create and process files containing different data types within the same file. You can also read and write unbounded data types such as strings. However, when using unbounded data types you must call the `'Input` and `'Output` attributes of the unbounded data type: these attributes write information about bounds or discriminants in addition to the object's actual data.

The following example shows file I/O that mixes both strings of different lengths and floating-point values:

```
with Ada.Text_IO;
with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;

procedure Show_String_Stream is
   F         : File_Type;
   S         : Stream_Access;
   File_Name : constant String := "float_file.bin";

   procedure Output (S  : Stream_Access;
                     FV : Float;
                     SV : String) is
   begin
      String'Output (S, SV);
      Float'Output (S,  FV);
   end Output;

   procedure Input_Display (S : Stream_Access) is
      SV : String := String'Input (S);
      FV : Float  := Float'Input (S);
   begin
      Ada.Text_IO.Put_Line (Float'Image (FV) & " --- " & SV);
   end Input_Display;

begin
   Create (F, Out_File, File_Name);
   S := Stream (F);

   Output (S, 1.5, "Hi!!");
   Output (S, 2.4, "Hello world!");
   Output (S, 6.7, "Something longer here...");

   Close (F);

   Open (F, In_File, File_Name);
   S := Stream (F);

   while not End_Of_File (F) loop
      Input_Display (S);
   end loop;
   Close (F);

end Show_String_Stream;
```

When you use Stream I/O, no information is written into the file indicating the type of the data that you wrote. If a file contains data from different types, you must reference types in the same order when reading a file as when you wrote it. If not, the information you get will be corrupted. Unfortunately, strong data typing doesn't help you in this case. Writing simple procedures for file I/O (as in the example above) may help ensuring that the file format is consistent.

Like direct I/O, stream I/O supports also allows you to access any location in the file. However, when doing so, you need to be extremely careful that the position of the new index is consistent with the data types you're expecting.

CHAPTER

TWENTY

STANDARD LIBRARY: NUMERICS

The standard library provides support for common numeric operations on floating-point types as well as on complex types and matrices. In the sections below, we present a brief introduction to these numeric operations.

20.1 Elementary Functions

The `Ada.Numerics.Elementary_Functions` package provides common operations for floating-point types, such as square root, logarithm, and the trigonometric functions (e.g., sin, cos). For example:

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

with Ada.Numerics.Elementary_Functions;
use  Ada.Numerics.Elementary_Functions;

procedure Show_Elem_Math is
   X : Float;
begin
   X := 2.0;
   Put_Line ("Square root of " & Float'Image (X)
             & " is " & Float'Image (Sqrt (X)));

   X := e;
   Put_Line ("Natural log of " & Float'Image (X)
             & " is " & Float'Image (Log (X)));

   X := 10.0 ** 6.0;
   Put_Line ("Log_10      of " & Float'Image (X)
             & " is " & Float'Image (Log (X, 10.0)));

   X := 2.0 ** 8.0;
   Put_Line ("Log_2       of " & Float'Image (X)
             & " is " & Float'Image (Log (X, 2.0)));

   X := Pi;
   Put_Line ("Cos         of " & Float'Image (X)
             & " is " & Float'Image (Cos (X)));

   X := -1.0;
   Put_Line ("Arccos      of " & Float'Image (X)
             & " is " & Float'Image (Arccos (X)));
end Show_Elem_Math;
```

Here we use the standard `e` and `Pi` constants from the `Ada.Numerics` package.

The `Ada.Numerics.Elementary_Functions` package provides operations for the `Float` type. Similar packages are available for `Long_Float` and `Long_Long_Float` types. For example, the `Ada.Numerics.Long_Elementary_Functions` package offers the same set of operations for the `Long_Float` type. In addition, the `Ada.Numerics.Generic_Elementary_Functions` package is a generic version of the package that you can instantiate for custom floating-point types. In fact, the `Elementary_Functions` package can be defined as follows:

```
package Elementary_Functions is new
  Ada.Numerics.Generic_Elementary_Functions (Float);
```

20.2 Random Number Generation

The `Ada.Numerics.Float_Random` package provides a simple random number generator for the range between 0.0 and 1.0. To use it, declare a generator `G`, which you pass to `Random`. For example:

```
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Numerics.Float_Random; use Ada.Numerics.Float_Random;

procedure Show_Float_Random_Num is
   G : Generator;
   X : Uniformly_Distributed;
begin
   Reset (G);

   Put_Line ("Some random numbers between "
             & Float'Image (Uniformly_Distributed'First) & " and "
             & Float'Image (Uniformly_Distributed'Last)  & ":");
   for I in 1 .. 15 loop
      X := Random (G);
      Put_Line (Float'Image (X));
   end loop;
end Show_Float_Random_Num;
```

The standard library also includes a random number generator for discrete numbers, which is part of the `Ada.Numerics.Discrete_Random` package. Since it's a generic package, you have to instantiate it for the desired discrete type. This allows you to specify a range for the generator. In the following example, we create an application that displays random integers between 1 and 10:

```
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Numerics.Discrete_Random;

procedure Show_Discrete_Random_Num is

   subtype Random_Range is Integer range 1 .. 10;

   package R is new Ada.Numerics.Discrete_Random (Random_Range);
   use R;

   G : Generator;
   X : Random_Range;
begin
   Reset (G);

   Put_Line ("Some random numbers between "
             & Integer'Image (Random_Range'First) & " and "
             & Integer'Image (Random_Range'Last)  & ":");
   for I in 1 .. 15 loop
```

(continues on next page)

(continued from previous page)

```
      X := Random (G);
      Put_Line (Integer'Image (X));
   end loop;
end Show_Discrete_Random_Num;
```

Here, package R is instantiated with the `Random_Range` type, which has a constrained range between 1 and 10. This allows us to control the range used for the random numbers. We could easily modify the application to display random integers between 0 and 20 by changing the specification of the `Random_Range` type. We can also use floating-point or fixed-point types.

20.3 Complex Types

The `Ada.Numerics.Complex_Types` package provides support for complex number types and the `Ada.Numerics.Complex_Elementary_Functions` package provides support for common operations on complex number types, similar to the `Ada.Numerics.Elementary_Functions` package. Finally, you can use the `Ada.Text_IO.Complex_IO` package to perform I/O operations on complex numbers. In the following example, we declare variables of the `Complex` type and initialize them using an aggregate:

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

with Ada.Numerics.Complex_Types;
use  Ada.Numerics.Complex_Types;

with Ada.Numerics.Complex_Elementary_Functions;
use  Ada.Numerics.Complex_Elementary_Functions;

with Ada.Text_IO.Complex_IO;

procedure Show_Elem_Math is

   package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
   use C_IO;

   X, Y  : Complex;
   R, Th : Float;
begin
   X := (2.0, -1.0);
   Y := (3.0,  4.0);

   Put (X);
   Put (" * ");
   Put (Y);
   Put (" is ");
   Put (X * Y);
   New_Line;
   New_Line;

   R  := 3.0;
   Th := Pi / 2.0;
   X  := Compose_From_Polar (R, Th);
   --  Alternatively:
   --  X := R * Exp ((0.0, Th));
   --  X := R * e ** Complex'(0.0, Th);

   Put ("Polar form:    "
```

(continues on next page)

(continued from previous page)

```
         & Float'Image (R)  & " * e**(i * "
         & Float'Image (Th) & ")");
   New_Line;

   Put ("Modulus     of ");
   Put (X);
   Put (" is ");
   Put (Float'Image (abs (X)));
   New_Line;

   Put ("Argument    of ");
   Put (X);
   Put (" is ");
   Put (Float'Image (Argument (X)));
   New_Line;
   New_Line;

   Put ("Sqrt        of ");
   Put (X);
   Put (" is ");
   Put (Sqrt (X));
   New_Line;
end Show_Elem_Math;
```

As we can see from this example, all the common operators, such as * and +, are available for complex types. You also have typical operations on complex numbers, such as `Argument` and `Exp`. In addition to initializing complex numbers in the cartesian form using aggregates, you can do so from the polar form by calling the `Compose_From_Polar` function.

The `Ada.Numerics.Complex_Types` and `Ada.Numerics.Complex_Elementary_Functions` packages provide operations for the `Float` type. Similar packages are available for `Long_Float` and `Long_Long_Float` types. In addition, the `Ada.Numerics.Generic_Complex_Types` and `Ada.Numerics.Generic_Complex_Elementary_Functions` packages are generic versions that you can instantiate for custom or pre-defined floating-point types. For example:

```
with Ada.Numerics.Generic_Complex_Types;
with Ada.Numerics.Generic_Complex_Elementary_Functions;
with Ada.Text_IO.Complex_IO;

procedure Show_Elem_Math is

   package Complex_Types is new
     Ada.Numerics.Generic_Complex_Types (Float);
   use Complex_Types;

   package Elementary_Functions is new
     Ada.Numerics.Generic_Complex_Elementary_Functions (Complex_Types);
   use Elementary_Functions;

   package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
   use C_IO;

   X, Y  : Complex;
   R, Th : Float;
```

20.4 Vector and Matrix Manipulation

The `Ada.Numerics.Real_Arrays` package provides support for vectors and matrices. It includes common matrix operations such as inverse, determinant, eigenvalues in addition to simpler operators such as matrix addition and multiplication. You can declare vectors and matrices using the `Real_Vector` and `Real_Matrix` types, respectively.

The following example uses some of the operations from the `Ada.Numerics.Real_Arrays` package:

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

with Ada.Numerics.Real_Arrays;
use  Ada.Numerics.Real_Arrays;

procedure Show_Matrix is

   procedure Put_Vector (V : Real_Vector) is
   begin
      Put ("    (");
      for I in V'Range loop
         Put (Float'Image (V (I)) & " ");
      end loop;
      Put_Line (")");
   end Put_Vector;

   procedure Put_Matrix (M : Real_Matrix) is
   begin
      for I in M'Range (1) loop
         Put ("    (");
         for J in M'Range (2) loop
            Put (Float'Image (M (I, J)) & " ");
         end loop;
         Put_Line (")");
      end loop;
   end Put_Matrix;

   V1       : Real_Vector := (1.0, 3.0);
   V2       : Real_Vector := (75.0, 11.0);

   M1       : Real_Matrix :=
                 ((1.0, 5.0, 1.0),
                  (2.0, 2.0, 1.0));
   M2       : Real_Matrix :=
                 ((31.0, 11.0, 10.0),
                  (34.0, 16.0, 11.0),
                  (32.0, 12.0, 10.0),
                  (31.0, 13.0, 10.0));
   M3       : Real_Matrix := ((1.0, 2.0),
                              (2.0, 3.0));
begin
   Put_Line ("V1");
   Put_Vector (V1);
   Put_Line ("V2");
   Put_Vector (V2);
   Put_Line ("V1 * V2 =");
   Put_Line ("    "
             & Float'Image (V1 * V2));
   Put_Line ("V1 * V2 =");
   Put_Matrix (V1 * V2);
   New_Line;
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("M1");
   Put_Matrix (M1);
   Put_Line ("M2");
   Put_Matrix (M2);
   Put_Line ("M2 * Transpose(M1) =");
   Put_Matrix (M2 * Transpose (M1));
   New_Line;

   Put_Line ("M3");
   Put_Matrix (M3);
   Put_Line ("Inverse (M3) =");
   Put_Matrix (Inverse (M3));
   Put_Line ("abs Inverse (M3) =");
   Put_Matrix (abs Inverse (M3));
   Put_Line ("Determinant (M3) =");
   Put_Line ("    "
             & Float'Image (Determinant (M3)));
   Put_Line ("Solve (M3, V1) =");
   Put_Vector (Solve (M3, V1));
   Put_Line ("Eigenvalues (M3) =");
   Put_Vector (Eigenvalues (M3));
   New_Line;
end Show_Matrix;
```

Matrix dimensions are automatically determined from the aggregate used for initialization when you don't specify them. You can, however, also use explicit ranges. For example:

```
M1       : Real_Matrix (1 .. 2, 1 .. 3) :=
             ((1.0, 5.0, 1.0),
              (2.0, 2.0, 1.0));
```

The `Ada.Numerics.Real_Arrays` package implements operations for the `Float` type. Similar packages are available for `Long_Float` and `Long_Long_Float` types. In addition, the `Ada.Numerics.Generic_Real_Arrays` package is a generic version that you can instantiate with custom floating-point types. For example, the `Real_Arrays` package can be defined as follows:

```
package Real_Arrays is new
  Ada.Numerics.Generic_Real_Arrays (Float);
```

CHAPTER

TWENTYONE

APPENDICES

21.1 Appendix A: Generic Formal Types

The following tables contain examples of available formal types for generics:

Formal type	Actual type
Incomplete type **Format**: `type T;`	Any type
Discrete type **Format**: `type T is (<>);`	Any integer, modular or enumeration type
Range type **Format**: `type T is range <>;`	Any signed integer type
Modular type **Format**: `type T is mod <>;`	Any modular type
Floating-point type **Format**: `type T is digits <>;`	Any floating-point type
Binary fixed-point type **Format**: `type T is delta <>;`	Any binary fixed-point type
Decimal fixed-point type **Format**: `type T is delta <> digits <>;`	Any decimal fixed-point type
Definite nonlimited private type **Format**: `type T is private;`	Any nonlimited, definite type
Nonlimited Private type with discriminant **Format**: `type T (D : DT) is private;`	Any nonlimited type with discriminant
Access type **Format**: `type A is access T;`	Any access type for type T
Definite derived type **Format**: `type T is new B;`	Any concrete type derived from base type B
Limited private type **Format**: `type T is limited private;`	Any definite type, limited or not
Incomplete tagged type **Format**: `type T is tagged;`	Any concrete, definite, tagged type
Definite tagged private type **Format**: `type T is tagged private;`	Any concrete, definite, tagged type
Definite tagged limited private type **Format**: `type T is tagged limited private;`	Any concrete definite tagged type, limited or not
Definite abstract tagged private type **Format**: `type T is abstract tagged private;`	Any nonlimited, definite tagged type, abstract or concrete

Continued on next page

Table 1 – continued from previous page

Formal type	Actual type
Definite abstract tagged limited private type **Format**: `type T is abstract tagged limited private;`	Any definite tagged type, limited or not, abstract or concrete
Definite derived tagged type **Format**: `type T is new B with private;`	Any concrete tagged type derived from base type B
Definite abstract derived tagged type **Format**: `type T is abstract new B with private;`	Any tagged type derived from base type B abstract or concrete
Array type **Format**: `type A is array (R) of T;`	Any array type with range R containing elements of type T
Interface type **Format**: `type T is interface;`	Any interface type T
Limited interface type **Format**: `type T is limited interface;`	Any limited interface type T
Task interface type **Format**: `type T is task interface;`	Any task interface type T
Synchronized interface type **Format**: `type T is synchronized interface;`	Any synchronized interface type T
Protected interface type **Format**: `type T is protected interface;`	Any protected interface type T
Derived interface type **Format**: `type T is new B and I with private;`	Any type T derived from base type B and interface I
Derived type with multiple interfaces **Format**: `type T is new B and I1 and I2 with private;`	Any type T derived from base type B and interfaces I1 and I2
Abstract derived interface type **Format**: `type T is abstract new B and I with private;`	Any type T derived from abstract base type B and interface I
Limited derived interface type **Format**: `type T is limited new B and I with private;`	Any type T derived from limited base type B and limited interface I
Abstract limited derived interface type **Format**: `type T is abstract limited new B and I with private;`	Any type T derived from abstract limited base type B and limited interface I
Synchronized interface type **Format**: `type T is synchronized new SI with private;`	Any type T derived from synchronized interface SI
Abstract synchronized interface type **Format**: `type T is abstract synchronized new SI with private;`	Any type T derived from synchronized interface SI

21.1.1 Indefinite version

Many of the examples above can be used for formal indefinite types:

Formal type	Actual type
Indefinite incomplete type **Format**: `type T (<>);`	Any type
Indefinite nonlimited private type **Format**: `type T (<>) is private;`	Any nonlimited type indefinite or definite
Indefinite limited private type **Format**: `type T (<>) is limited private;`	Any type, limited or not, indefinite or definite
Incomplete indefinite tagged private type **Format**: `type T (<>) is tagged;`	Any concrete tagged type, indefinite or definite
Indefinite tagged private type **Format**: `type T (<>) is tagged private;`	Any concrete, limited tagged type, indefinite or definite
Indefinite tagged limited private type **Format**: `type T (<>) is tagged limited private;`	Any concrete tagged type, limited or not, indefinite or definite
Indefinite abstract tagged private type **Format**: `type T (<>) is abstract tagged private;`	Any nonlimited tagged type, indefinite or definite, abstract or concrete
Indefinite abstract tagged limited private type **Format**: `type T (<>) is abstract tagged limited private;`	Any tagged type, limited or not, indefinite or definite abstract or concrete
Indefinite derived tagged type **Format**: `type T (<>) is new B with private;`	Any tagged type derived from base type B, indefinite or definite
Indefinite abstract derived tagged type **Format**: `type T (<>) is abstract new B with private;`	Any tagged type derived from base type B, indefinite or definite abstract or concrete

The same examples could also contain discriminants. In this case, `(<>)` is replaced by a list of discriminants, e.g.: `(D: DT)`.

21.2 Appendix B: Containers

The following table shows all containers available in Ada, including their versions (standard, bounded, unbounded, indefinite):

Category	Container	Std	Bounded	Unbounded	Indefinite
Vector	`Vectors`	Y	Y		Y
List	`Doubly_Linked_Lists`	Y	Y		Y
Map	`Hashed_Maps`	Y	Y		Y
Map	`Ordered_Maps`	Y	Y		Y
Set	`Hashed_Sets`	Y	Y		Y
Set	`Ordered_Sets`	Y	Y		Y
Tree	`Multiway_Trees`	Y	Y		Y
Generic	`Holders`				Y
Queue	`Synchronized_Queue_Interfaces`	Y			
Queue	`Synchronized_Queues`		Y	Y	
Queue	`Priority_Queues`		Y	Y	

The following table presents the prefixing applied to the container name that depends on its version. As indicated in the table, the standard version does not have a prefix associated with it.

Version	Naming prefix
Std	
Bounded	`Bounded_`
Unbounded	`Unbounded_`
Indefinite	`Indefinite_`