Ada

for the Embedded C Developer

Quentin Ochem
Robert Tice
Gustavo A. Hoffmann
Patrick Rogers

LEARN.
ADACORE.COM

LEARN.
ADACORE.COM

Ada for the Embedded C Developer

Release 2021-01

Quentin Ochem
Gustavo A. Hoffmann
Robert Tice
Patrick Rogers.

Jan 16, 2021

CONTENTS

This course introduces you to the Ada language by comparing it to C. It assumes that you have good knowledge of the C language. It also assumes that the choice of learning Ada is guided by considerations linked to reliability, safety or security. In that sense, it teaches you Ada paradigms that should be applied in replacement of those usually applied in C.

This course also introduces you to the SPARK subset of the Ada programming language, which removes a few features of the language with undefined behavior, so that the code is fit for sound static analysis techniques.

This course was written by Quentin Ochem, Robert Tice, Gustavo A. Hoffmann, and Patrick Rogers and reviewed by Patrick Rogers, Filip Gajowniczek, and Tucker Taft.

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

CHAPTER

ONE

INTRODUCTION

1.1 So, what is this Ada thing anyway?

To answer this question let's introduce Ada as it compares to C for an embedded application. C developers are used to a certain coding semantic and style of programming. Especially in the embedded domain, developers are used to working at a very low level near the hardware to directly manipulate memory and registers. Normal operations involve mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations as it is a low level language that was designed to replace assembly language for faster, more efficient programming. Because of this minimal abstraction, the programmer has to model the data that represents the problem they are trying to solve using the language of the physical hardware.

Let's look at an example of this problem in action by comparing the same program in Ada and C:

[C]

```c
#include <stdio.h>
#include <stdlib.h>

#define DEGREES_MAX           (360)
typedef unsigned int degrees;

#define MOD_DEGREES(x)   (x % DEGREES_MAX)

degrees add_angles(degrees* list, int length)
{
  degrees sum = 0;
  for(int i = 0; i < length; ++i) {
    sum += list[i];
  }

  return sum;
}

int main(int argc, char** argv)
{
  degrees list[argc - 1];

  for(int i = 1; i < argc; ++i) {
    list[i - 1] = MOD_DEGREES(atoi(argv[i]));
  }

  printf("Sum: %d\n", add_angles(list, argc - 1));

  return 0;
}
```

[Ada]

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

procedure Sum_Angles is

   DEGREES_MAX : constant := 360;
   type Degrees is mod DEGREES_MAX;

   type Degrees_List is array (Natural range <>) of Degrees;

   function Add_Angles (List : Degrees_List) return Degrees
   is
      Sum : Degrees := 0;
   begin
      for I in List'Range loop
         Sum := Sum + List (I);
      end loop;

      return Sum;
   end Add_Angles;

   List : Degrees_List (1 .. Argument_Count);
begin
   for I in List'Range loop
      List (I) := Degrees (Integer'Value (Argument (I)));
   end loop;

   Put_Line ("Sum:" & Add_Angles (List)'Img);
end Sum_Angles;
```

Here we have a piece of code in C and in Ada that takes some numbers from the command line and stores them in an array. We then sum all of the values in the array and print the result. The tricky part here is that we are working with values that model an angle in degrees. We know that angles are modular types, meaning that angles greater than 360° can also be represented as `Angle mod 360`. So if we have an angle of 400°, this is equivalent to 40°. In order to model this behavior in C we had to create the `MOD_DEGREES` macro, which performs the modulus operation. As we read values from the command line, we convert them to integers and perform the modulus before storing them into the array. We then call add_angles which returns the sum of the values in the array. Can you spot the problem with the C code?

Try running the Ada and C examples using the input sequence `340 2 50 70`. What does the C program output? What does the Ada program output? Why are they different?

The problem with the C code is that we forgot to call `MOD_DEGREES` in the for loop of add_angles. This means that it is possible for add_angles to return values greater than `DEGREES_MAX`. Let's look at the equivalent Ada code now to see how Ada handles the situation. The first thing we do in the Ada code is to create the type `Degrees` which is a modular type. This means that the compiler is going to handle performing the modulus operation for us. If we use the same for loop in the `Add_Angles` function, we can see that we aren't doing anything special to make sure that our resulting value is within the 360° range we need it to be in.

The takeaway from this example is that Ada tries to abstract some concepts from the developer so that the developer can focus on solving the problem at hand using a data model that models the real world rather than using data types prescribed by the hardware. The main benefit of this is that the compiler takes some responsibility from the developer for generating correct code. In this example we forgot to put in a check in the C code. The compiler inserted the check for us in the Ada code because we told the compiler what we were trying to accomplish by defining strong types.

Ideally, we want all the power that the C programming language can give us to manipulate the hardware we are working on while also allowing us the ability to more accurately model data in a safe way. So, we have a dilemma; what can give us the power of operations like the C language,

but also provide us with features that can minimize the potential for developer error? Since this course is about Ada, it's a good bet we're about to introduce the Ada language as the answer to this question...

Unlike C, the Ada language was designed as a higher level language from its conception; giving more responsibility to the compiler to generate correct code. As mentioned above, with C, developers are constantly shifting, masking, and accessing bits directly on memory pointers. In Ada, all of these operations are possible, but in most cases, there is a better way to perform these operations using higher level constructs that are less prone to mistakes, like off-by-one or unintentional buffer overflows. If we were to compare the same application written using C and with Ada using high level constructs, we would see similar performance in terms of speed and memory efficiency. If we compare the object code generated by both compilers, it's possible that they even look identical!

1.2 Ada — The Technical Details

Like C, Ada is a compiled language. This means that the compiler will parse the source code and emit machine code native to the target hardware. The Ada compiler we will be discussing in this course is the GNAT compiler. This compiler is based on the GCC technology like many C and C++ compilers available. When the GNAT compiler is invoked on Ada code, the GNAT front-end expands and translates the Ada code into an intermediate language which is passed to GCC where the code is optimized and translated to machine code. A C compiler based on GCC performs the same steps and uses the same intermediate GCC representation. This means that the optimizations we are used to seeing with a GCC based C compiler can also be applied to Ada code. The main difference between the two compilers is that the Ada compiler is expanding high level constructs into intermediate code. After expansion, the Ada code will be very similar to the equivalent C code.

It is possible to do a line-by-line translation of C code to Ada. This feels like a natural step for a developer used to C paradigms. However, there may be very little benefit to doing so. For the purpose of this course, we're going to assume that the choice of Ada over C is guided by considerations linked to reliability, safety or security. In order to improve upon the reliability, safety and security of our application, Ada paradigms should be applied in replacement of those usually applied in C. Constructs such as pointers, preprocessor macros, bitwise operations and defensive code typically get expressed in Ada in very different ways, improving the overall reliability and readability of the applications. Learning these new ways of coding, often, requires effort by the developer at first, but proves more efficient once the paradigms are understood.

In this course we will also introduce the SPARK subset of the Ada programming language. The SPARK subset removes a few features of the language, i.e., those that make proof difficult, such as pointer aliasing. By removing these features we can write code that is fit for sound static analysis techniques. This means that we can run mathematical provers on the SPARK code to prove certain safety or security properties about the code.

CHAPTER

TWO

THE C DEVELOPER'S PERSPECTIVE ON ADA

2.1 What we mean by Embedded Software

The Ada programming language is a general programming language, which means it can be used for many different types of applications. One type of application where it particularly shines is reliable and safety-critical embedded software; meaning, a platform with a microprocessor such as ARM, PowerPC, x86, or RISC-V. The application may be running on top of an embedded operating system, such as an embedded Linux, or directly on bare metal. And the application domain can range from small entities such as firmware or device controllers to flight management systems, communication based train control systems, or advanced driver assistance systems.

2.2 The GNAT Toolchain

The toolchain used throughout this course is called GNAT, which is a suite of tools with a compiler based on the GCC environment. It can be obtained from AdaCore, either as part of a commercial contract with GNAT Pro[2] or at no charge with the GNAT Community edition[3]. The information in this course will be relevant no matter which edition you're using. Most examples will be runnable on the native Linux or Windows version for convenience. Some will only be relevant in the context of a cross toolchain, in which case we'll be using the embedded ARM bare metal toolchain.

As for any Ada compiler, GNAT takes advantage of implementation permissions and offers a project management system. Because we're talking about embedded platforms, there are a lot of topics that we'll go over which will be specific to GNAT, and sometimes to specific platforms supported by GNAT. We'll try to make the distinction between what is GNAT-specific and Ada generic as much as possible throughout this course.

For an introduction to the GNAT Toolchain for the GNAT Community edition, you may refer to the Introduction to GNAT Toolchain[4] course.

2.3 The GNAT Toolchain for Embedded Targets

When we're discussing embedded programming, our target device is often different from the host, which is the device we're using to actually write and build an application. In this case, we're talking about cross compilation platforms (concisely referred to as cross platforms).

The GNAT toolchain supports cross platform compilation for various target devices. This section provides a short introduction to the topic. For more details, please refer to the GNAT User's Guide Supplement for Cross Platforms[5]

[2] https://www.adacore.com/gnatpro
[3] https://www.adacore.com/community
[4] https://learn.adacore.com/courses/GNAT_Toolchain_Intro/index.html
[5] https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx.html

GNAT supports two types of cross platforms:

- **cross targets**, where the target device has an embedded operating system.
  - ARM-Linux, which is commonly found in a Raspberry-Pi, is a prominent example.
- **bareboard targets**, where the run-times do not depend on an operating system.
  - In this case, the application has direct access to the system hardware.

For each platform, a set of run-time libraries is available. Run-time libraries implement a subset of the Ada language for different use cases, and they're different for each target platform. They may be selected via an attribute in the project's GPR project file or as a command-line switch to **`GPRbuild`**. Although the run-time libraries may vary from target to target, the user interface stays the same, providing portability for the application.

Run-time libraries consists of:

1. Files that are dependent on the target board.
   - These files are responsible for configuring and interacting with the hardware.
   - They are known as a Board Support Package — commonly referred to by their abbreviation *BSP*.
2. Code that is target-independent.
   - This code implements language-defined functionality.

The bareboard run-time libraries are provided as customized run-times that are configured to target a very specific micro-controller or processor. Therefore, for different micro-controllers and processors, the run-time libraries need to be ported to the specific target. These are some examples of what needs to be ported:

- startup code / scripts;
- clock frequency initializations;
- memory mapping / allocation;
- interrupts and interrupt priorities;
- register descriptions.

For more details on the topic, please refer to the following chapters of the GNAT User's Guide Supplement for Cross Platforms[6]:

- Bareboard Topics[7]
- Customized Run-Time Libraries[8]

2.4 Hello World in Ada

The first piece of code to translate from C to Ada is the usual Hello World program:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    printf("Hello World\n");
    return 0;
}
```

[6] https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx.html
[7] http://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx/bareboard_topics.html
[8] http://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx/customized_run-time_libraries.html

[Ada]

```
with Ada.Text_IO;

procedure Hello_World
is
begin
   Ada.Text_IO.Put_Line ("Hello World");
end Hello_World;
```

The resulting program will print `Hello World` on the screen. Let's now dissect the Ada version to describe what is going on:

The first line of the Ada code is giving us access to the `Ada.Text_IO` library which contains the `Put_Line` function we will use to print the text to the console. This is similar to C's `#include <stdio.h>`. We then create a procedure which executes `Put_Line` which prints to the console. This is similar to C's `printf` function. For now, we can assume these Ada and C features have similar functionality. In reality, they are very different. We will explore that more as we delve further into the Ada language.

You may have noticed that the Ada syntax is more verbose than C. Instead of using braces `{}` to declare scope, Ada uses keywords. `is` opens a declarative scope — which is empty here as there's no variable to declare. `begin` opens a sequence of statements. Within this sequence, we're calling the function `Put_Line`, prefixing explicitly with the name of the library unit where it's declared, `Ada.Text_IO`. The absence of the end of line `\n` can also be noted, as `Put_Line` always terminates by an end of line.

2.5 The Ada Syntax

Ada syntax might seem peculiar at first glance. Unlike many other languages, it's not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a more explicit language — its syntax was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner. For example:

- full words like `begin` and `end` are used in place of curly braces.
- Conditions are written using `if`, `then`, `elsif`, `else`, and `end if`.
- Ada's assignment operator does not double as an expression, eliminating potential mistakes that could be caused by = being used where == should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens `--` mark the start of a comment that continues to the end of the line. This is exactly the same as using `//` for comments in C. Multi line comments like C's `/* */` do not exist in Ada.

Ada compilers are stricter with type and range checking than most C programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding, but this helps detect problems and vulnerabilities at compile time — early on in the development cycle. In addition, checks (such as array bounds checks) provide verification that could not be done at compile time but can be performed either at run-time, or through formal proof (with the SPARK tooling).

Ada identifiers and reserved words are case insensitive. The identifiers `VAR`, `var` and `VaR` are treated as the same identifier; likewise `begin`, `BEGIN`, `Begin`, etc. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:

abort	else	null	select
abs	elsif	of	separate
abstract	end	or	some
accept	entry	others	subtype
access	exception	out	synchronized
aliased	exit	overriding	tagged
all	for	package	task
and	function	pragma	terminate
array	generic	private	then
at	goto	procedure	type
begin	if	protected	until
body	in	raise	use
case	interface	range	when
constant	is	record	while
declare	limited	rem	with
delay	loop	renames	xor
delta	mod	requeue	
digits	new	return	
do	not	reverse	

2.6 Compilation Unit Structure

Both C and Ada were designed with the idea that the code specification and code implementation could be separated into two files. In C, the specification typically lives in the .h, or header file, and the implementation lives in the .c file. Ada is superficially similar to C. With the GNAT toolchain, compilation units are stored in files with an .ads extension for specifications and with an .adb extension for implementations.

One main difference between the C and Ada compilation structure is that Ada compilation units are structured into something called packages.

2.7 Packages

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C. A specification defines a package and the implementation implements the package. We saw this in an earlier example when we included the `Ada.Text_IO` package into our application. The package specification has the structure:

[Ada]

```
--  my_package.ads
package My_Package is

   --  public declarations

private

   --  private declarations

end My_Package;
```

The package implementation, or body, has the structure:

```
--  my_package.adb
package body My_Package is

   --  implementation

end My_Package;
```

2.7.1 Declaration Protection

An Ada package contains three parts that, for GNAT, are separated into two files: `.ads` files contain public and private Ada specifications, and `.adb` files contain the implementation, or Ada bodies.

[Ada]

```
package Package_Name is
   -- public specifications
private
   -- private specifications
end Package_Name;

package body Package_Name is
   -- implementation
end Package_Name;
```

Private types are useful for preventing the users of a package's types from depending on the types' implementation details. Another use-case is the prevention of package users from accessing package state/data arbitrarily. The private reserved word splits the package spec into *public* and *private* parts. For example:

[Ada]

```
package Types is
   type Type_1 is private;
   type Type_2 is private;
   type Type_3 is private;
   procedure P (X : Type_1);
   --  ...
private
   procedure Q (Y : Type_1);
   type Type_1 is new Integer range 1 .. 1000;
   type Type_2 is array (Integer range 1 .. 1000) of Integer;
   type Type_3 is record
      A, B : Integer;
   end record;
end Types;
```

Subprograms declared above the `private` separator (such as P) will be visible to the package user, and the ones below (such as Q) will not. The body of the package, the implementation, has access to both parts. A package specification does not require a private section.

2.7.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

[Ada]

```
-- root-child.ads
```

(continues on next page)

(continued from previous page)

```
package Root.Child is
   --  package spec goes here
end Root.Child;

-- root-child.adb

package body Root.Child is
   --  package body goes here
end Root.Child;
```

Here, `Root.Child` is a child package of `Root`. The public part of `Root.Child` has access to the public part of `Root`. The private part of `Child` has access to the private part of `Root`, which is one of the main advantages of child packages. However, there is no visibility relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

2.7.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a `with` clause that references the package, which is similar to the C `#include` directive. After a `with` clause makes a package available, references to the package contents require the name of the package as a prefix, with a dot after the package name. This prefix can be omitted if a `use` clause is employed.

[Ada]

```
--  pck.ads

package Pck is
   My_Glob : Integer;
end Pck;
```

```
--  main.adb

with Pck;

procedure Main is
begin
   Pck.My_Glob := 0;
end Main;
```

In contrast to C, the Ada `with` clause is a *semantic inclusion* mechanism rather than a *text inclusion* mechanism; for more information on this difference please refer to Packages[9] .

2.8 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
```

(continues on next page)

[9] https://learn.adacore.com/courses/intro-to-ada/chapters/modular_programming.html

(continued from previous page)

```
{
    // variable declarations
    int a = 0, b = 0, c = 100, d;

    // c shorthand for increment
    a++;

    // regular addition
    d = a + b + c;

    // printing the result
    printf("d = %d\n", d);

    return 0;
}
```

[Ada]

```
with Ada.Text_IO;

procedure Main
is
   --  variable declaration
   A, B : Integer := 0;
   C    : Integer := 100;
   D    : Integer;
begin
   --  Ada does not have a shortcut format for increment like in C
   A := A + 1;

   --  regular addition
   D := A + B + C;

   --  printing the result
   Ada.Text_IO.Put_Line ("D =" & D'Img);
end Main;
```

You'll notice that, in both languages, statements are terminated with a semicolon. This means that you can have multi-line statements.

The shortcuts of incrementing and decrementing

You may have noticed that Ada does not have something similar to the `a++` or `a--` operators. Instead you must use the full assignment `A := A + 1` or `A := A - 1`.

In the Ada example above, there are two distinct sections to the `procedure Main`. This first section is delimited by the `is` keyword and the `begin` keyword. This section is called the declarative block of the subprogram. The declarative block is where you will define all the local variables which will be used in the subprogram. C89 had something similar, where developers were required to declare their variables at the top of the scope block. Most C developers may have run into this before when trying to write a for loop:

[C]

```
/* The C89 version */

#include <stdio.h>

int average(int* list, int length)
{
```

(continues on next page)

(continued from previous page)

```
    int i;
    int sum = 0;

    for(i = 0; i < length; ++i) {
        sum += list[i];
    }
    return (sum / length);
}

int main(int argc, const char * argv[])
{
    int vals[] = { 2, 2, 4, 4 };

    printf("Average: %d\n", average(vals, 4));
}
```

[C]

```
// The modern C way

#include <stdio.h>

int average(int* list, int length)
{
    int sum = 0;

    for(int i = 0; i < length; ++i) {
        sum += list[i];
    }

    return (sum / length);
}

int main(int argc, const char * argv[])
{
    int vals[] = { 2, 2, 4, 4 };

    printf("Average: %d\n", average(vals, 4));
}
```

For the fun of it, let's also see the Ada way to do this:

[Ada]

```
with Ada.Text_IO;

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

   function Average (List : Int_Array) return Integer
   is
      Sum : Integer := 0;
   begin
      for I in List'Range loop
         Sum := Sum + List (I);
      end loop;

      return (Sum / List'Length);
   end Average;

   Vals : constant Int_Array (1 .. 4) := (2, 2, 4, 4);
```

(continues on next page)

(continued from previous page)

```
begin
   Ada.Text_IO.Put_Line ("Average: " & Integer'Image (Average (Vals)));
end Main;
```

We will explore more about the syntax of loops in Ada in a future section of this course; but for now, notice that the `I` variable used as the loop index is not declared in the declarative section!

Declaration Flippy Floppy

Something peculiar that you may have noticed about declarations in Ada is that they are backwards from the way C does declarations. The C language expects the type followed by the variable name. Ada expects the variable name followed by a semicolon and then the type.

The next block in the Ada example is between the `begin` and `end` keywords. This is where your statements will live. You can create new scopes by using the `declare` keyword:

[Ada]

```
with Ada.Text_IO;

procedure Main
is
   --  variable declaration
   A, B : Integer := 0;
   C    : Integer := 100;
   D    : Integer;
begin
   --  Ada does not have a shortcut format for increment like in C
   A := A + 1;

   --  regular addition
   D := A + B + C;

   --  printing the result
   Ada.Text_IO.Put_Line ("D =" & D'Img);

   declare
      E : constant Integer := D * 100;
   begin
      --  printing the result
      Ada.Text_IO.Put_Line ("E =" & E'Img);
   end;

end Main;
```

Notice that we declared a new variable E whose scope only exists in our newly defined block. The equivalent C code is:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // variable declarations
    int a = 0, b = 0, c = 100, d;

    // c shorthand for increment
    a++;
```

(continues on next page)

(continued from previous page)

```
    // regular addition
    d = a + b + c;

    // printing the result
    printf("d = %d\n", d);

    {
        const int e = d * 100;
        printf("e = %d\n", e);
    }

    return 0;
}
```

Fun Fact about the C language assignment operator =: Did you know that an assignment in C can be used in an expression? Let's look at an example:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 0;

    if (a = 10)
        printf("True\n");
    else
        printf("False\n");

    return 0;
}
```

Run the above code example. What does it output? Is that what you were expecting?

The author of the above code example probably meant to test if `a == 10` in the if statement but accidentally typed = instead of ==. Because C treats assignment as an expression, it was able to evaluate `a = 10`.

Let's look at the equivalent Ada code:

[Ada]

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

procedure Main
is
   A : Integer := 0;
begin

   if A := 10 then
      Put_Line ("True");
   else
      Put_Line ("False");
   end if;
end Main;
```

The above code will not compile. This is because Ada does no allow assignment as an expression.

The "use" clause

You'll notice in the above code example, after `with Ada.Text_IO;` there is a new statement we

haven't seen before — `use Ada.Text_IO;`. You may also notice that we are not using the `Ada.Text_IO` prefix before the `Put_Line` statements. When we add the use clause it tells the compiler that we won't be using the prefix in the call to subprograms of that package. The use clause is something to use with caution. For example: if we use the `Ada.Text_IO` package and we also have a `Put_Line` subprogram in our current compilation unit with the same signature, we have a (potential) collision!

2.9 Conditions

The syntax of an if statement:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // try changing the initial value to change the
    //    output of the program
    int v = 0;

    if (v > 0) {
        printf("Positive\n");
    }
    else if (v < 0) {
        printf("Negative\n");
    }
    else {
        printf("Zero\n");
    }

    return 0;
}
```

[Ada]

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

procedure Main
is
   --  try changing the initial value to change the
   --    output of the program
   V : constant Integer := 0;
begin
   if V > 0 then
      Put_Line ("Positive");
   elsif V < 0 then
      Put_Line ("Negative");
   else
      Put_Line ("Zero");
   end if;
end Main;
```

In Ada, everything that appears between the `if` and `then` keywords is the conditional expression, no parentheses are required. Comparison operators are the same except for:

Operator	C	Ada
Equality	==	=
Inequality	!=	/=
Not	!	not
And	&&	and
Or	||	or

The syntax of a switch/case statement:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    // try changing the initial value to change the
    //    output of the program
    int v = 0;

    switch(v) {
        case 0:
            printf("Zero\n");
            break;
        case 1: case 2: case 3: case 4: case 5:
        case 6: case 7: case 8: case 9:
            printf("Positive\n");
            break;
        case 10: case 12: case 14: case 16: case 18:
            printf("Even number between 10 and 18\n");
            break;
        default:
            printf("Something else\n");
            break;
    }

    return 0;
}
```

[Ada]

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

procedure Main
is
   --  try changing the initial value to change the
   --    output of the program
   V : constant Integer := 0;
begin
   case V is
      when 0 =>
         Put_Line ("Zero");
      when 1 .. 9 =>
         Put_Line ("Positive");
      when 10 | 12 | 14 | 16 | 18 =>
         Put_Line ("Even number between 10 and 18");
      when others =>
         Put_Line ("Something else");
   end case;
end Main;
```

Switch or Case?

A switch statement in C is the same as a case statement in Ada. This may be a little strange because C uses both keywords in the statement syntax. Let's make an analogy between C and Ada: C's `switch` is to Ada's `case` as C's `case` is to Ada's `when`.

Notice that in Ada, the case statement does not use the `break` keyword. In C, we use `break` to stop the execution of a case branch from falling through to the next branch. Here is an example:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int v = 0;

    switch(v) {
        case 0:
            printf("Zero\n");
        case 1:
            printf("One\n");
        default:
            printf("Other\n");
    }

    return 0;
}
```

Run the above code with `v = 0`. What prints? What prints when we change the assignment to `v = 1`?

When `v = 0` the program outputs the strings `Zero` then `One` then `Other`. This is called fall through. If you add the `break` statements back into the `switch` you can stop this fall through behavior from happening. The reason why fall through is allowed in C is to allow the behavior from the previous example where we want a specific branch to execute for multiple inputs. Ada solves this a different way because it is possible, or even probable, that the developer might forget a `break` statement accidentally. So Ada does not allow fall through. Instead, you can use Ada's syntax to identify when a specific branch can be executed by more than one input. If you want a range of values for a specific branch you can use the `First .. Last` notation. If you want a few non-consecutive values you can use the `Value1 | Value2 | Value3` notation.

Instead of using the word `default` to denote the catch-all case, Ada uses the `others` keyword.

2.10 Loops

Let's start with some syntax:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int v;

    // this is a while loop
    v = 1;
    while(v < 100) {
        v *= 2;
    }
```

(continues on next page)

(continued from previous page)

```
    printf("v = %d\n", v);

    // this is a do while loop
    v = 1;
    do {
        v *= 2;
    } while(v < 200);
    printf("v = %d\n", v);

    // this is a for loop
    v = 0;
    for(int i = 0; i < 5; ++i) {
        v += (i * i);
    }
    printf("v = %d\n", v);

    // this is a forever loop with a conditional exit
    v = 0;
    while(1) {
        // do stuff here
        v += 1;
        if(v == 10)
            break;
    }
    printf("v = %d\n", v);

    // this is a loop over an array
    {
        #define ARR_SIZE (10)
        const int arr[ARR_SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        for(int i = 0; i < ARR_SIZE; ++i) {
            sum += arr[i];
        }
        printf("sum = %d\n", sum);
    }
}
```

[Ada]

```
with Ada.Text_IO;

procedure Main is
   V : Integer;
begin
   --  this is a while loop
   V := 1;
   while V < 100 loop
      V := V * 2;
   end loop;
   Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));

   --  Ada doesn't have an explicit do while loop
   --  instead you can use the loop and exit keywords
   V := 1;
   loop
      V := V * 2;
      exit when V >= 200;
   end loop;
   Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));
```

(continues on next page)

(continued from previous page)

```
   --  this is a for loop
   V := 0;
   for I in 0 .. 4 loop
      V := V + (I * I);
   end loop;
   Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));

   --  this is a forever loop with a conditional exit
   V := 0;
   loop
      --  do stuff here
      V := V + 1;
      exit when V = 10;
   end loop;
   Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));

   --  this is a loop over an array
   declare
      type Int_Array is array (Natural range 1 .. 10) of Integer;

      Arr : constant Int_Array := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
      Sum : Integer := 0;
   begin
      for I in Arr'Range loop
         Sum := Sum + Arr (I);
      end loop;
      Ada.Text_IO.Put_Line ("Sum = " & Integer'Image (Sum));
   end;
end Main;
```

The loop syntax in Ada is pretty straightforward. The `loop` and `end loop` keywords are used to open and close the loop scope. Instead of using the `break` keyword to exit the loop, Ada has the `exit` statement. The `exit` statement can be combined with a logic expression using the `exit when` syntax.

The major deviation in loop syntax is regarding for loops. You'll notice, in C, that you sometimes declare, and at least initialize a loop counter variable, specify a loop predicate, or an expression that indicates when the loop should continue executing or complete, and last you specify an expression to update the loop counter.

[C]

```
for (initialization expression; loop predicate; update expression) {
    // some statements
}
```

In Ada, you don't declare or initialize a loop counter or specify an update expression. You only name the loop counter and give it a range to loop over. The loop counter is **read-only**! You cannot modify the loop counter inside the loop like you can in C. And the loop counter will increment consecutively along the specified range. But what if you want to loop over the range in reverse order?

[C]

```
#include <stdio.h>

#define MY_RANGE (10)

int main(int argc, const char * argv[])
{
```

(continues on next page)

(continued from previous page)

```
    for (int i = MY_RANGE; i >= 0; --i) {
        printf("%d\n", i);
    }

    return 0;
}
```

[Ada]

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

procedure Main
is
   My_Range : constant := 10;
begin
   for I in reverse 0 .. My_Range loop
      Put_Line (I'Img);
   end loop;
end Main;
```

Tick Image

Strangely enough, Ada people call the single apostrophe symbol, `'`, "tick". This "tick" says the we are accessing an attribute of the variable. When we do `'Img` on a variable of a numerical type, we are going to return the string version of that numerical type. So in the for loop above, `I'Img`, or "I tick image" will return the string representation of the numerical value stored in I. We have to do this because Put_Line is expecting a string as an input parameter.

We'll discuss attributes in more details *later in this chapter* (page 33).

In the above example, we are traversing over the range in reverse order. In Ada, we use the `reverse` keyword to accomplish this.

In many cases, when we are writing a for loop, it has something to do with traversing an array. In C, this is a classic location for off-by-one errors. Let's see an example in action:

[C]

```
#include <stdio.h>

#define LIST_LENGTH (100)

int main(int argc, const char * argv[])
{
    int list[LIST_LENGTH];

    for(int i = LIST_LENGTH; i > 0; --i) {
        list[i] = LIST_LENGTH - i;
    }

    for (int i = 0; i < LIST_LENGTH; ++i)
    {
        printf("%d ", list[i]);
    }
    printf("\n");

    return 0;
}
```

[Ada]

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

procedure Main
is
   type Int_Array is array (Natural range 1 .. 100) of Integer;

   List : Int_Array;
begin

   for I in reverse List'Range loop
      List (I) := List'Last - I;
   end loop;

   for I in List'Range loop
      Put (List (I)'Img & " ");
   end loop;

   New_Line;
end Main;
```

The above Ada and C code should initialize an array using a for loop. The initial values in the array should be contiguously decreasing from 99 to 0 as we index from the first index to the last index. In other words, the first index has a value of 99, the next has 98, the next 97 ... the last has a value of 0.

If you run both the C and Ada code above you'll notice that the outputs of the two programs are different. Can you spot why?

In the C code there are two problems:

1. There's a buffer overflow in the first iteration of the loop. We would need to modify the loop initialization to `int i = LIST_LENGTH - 1;`. The loop predicate should be modified to `i >= 0;`
2. The C code also has another off-by-one problem in the math to compute the value stored in `list[i]`. The expression should be changed to be `list[i] = LIST_LENGTH - i - 1;`.

These are typical off-by-one problems that plagues C programs. You'll notice that we didn't have this problem with the Ada code because we aren't defining the loop with arbitrary numeric literals. Instead we are accessing attributes of the array we want to manipulate and are using a keyword to determine the indexing direction.

We can actually simplify the Ada for loop a little further using iterators:

[Ada]

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

procedure Main
is
   type Int_Array is array (Natural range 1 .. 100) of Integer;

   List : Int_Array;
begin

   for I in reverse List'Range loop
      List (I) := List'Last - I;
   end loop;

   for I of List loop
      Put (I'Img & " ");
   end loop;
```

(continues on next page)

(continued from previous page)

```
   New_Line;
end Main;
```

In the second for loop, we changed the syntax to `for I of List`. Instead of I being the index counter, it is now an iterator that references the underlying element. This example of Ada code is identical to the last bit of Ada code. We just used a different method to index over the second for loop. There is no C equivalent to this Ada feature, but it is similar to C++'s range based for loop.

2.11 Type System

2.11.1 Strong Typing

Ada is considered a "strongly typed" language. This means that the language does not define any implicit type conversions. C does define implicit type conversions, sometimes referred to as *integer promotion*. The rules for promotion are fairly straightforward in simple expressions but can get confusing very quickly. Let's look at a typical place of confusion with implicit type conversion:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    unsigned char a = 0xFF;
    char b = 0xFF;

    printf("Does a == b?\n");
    if(a == b)
        printf("Yes.\n");
    else
        printf("No.\n");

    printf("a: 0x%08X, b: 0x%08X\n", a, b);

    return 0;
}
```

Run the above code. You will notice that `a != b`! If we look at the output of the last `printf` statement we will see the problem. `a` is an unsigned number where `b` is a signed number. We stored a value of `0xFF` in both variables, but `a` treated this as the decimal number `255` while `b` treated this as the decimal number `-1`. When we compare the two variables, of course they aren't equal; but that's not very intuitive. Let's look at the equivalent Ada example:

[Ada]

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

 procedure Main
 is
    type Char is range 0 .. 255;
    type Unsigned_Char is mod 256;

    A : Char := 16#FF#;
    B : Unsigned_Char := 16#FF#;
 begin

    Put_Line ("Does A = B?");
```

(continues on next page)

(continued from previous page)

```
    if A = B then
       Put_Line ("Yes");
    else
       Put_Line ("No");
    end if;

 end Main;
```

If you try to run this Ada example you will get a compilation error. This is because the compiler is telling you that you cannot compare variables of two different types. We would need to explicitly cast one side to make the comparison against two variables of the same type. By enforcing the explicit cast we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect.

Another example: you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

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

procedure Strong_Typing is
   Alpha  : constant Integer := 1;
   Beta   : constant Integer := 10;
   Result : Float;
begin
   Result := Float (Alpha) / Float (Beta);

   Put_Line (Float'Image (Result));
end Strong_Typing;
```

[C]

```
#include <stdio.h>

void weakTyping (void) {
    const int   alpha = 1;
    const int   beta = 10;
    float result;

    result = alpha / beta;

    printf("%f\n", result);
}

int main(int argc, const char * argv[])
{
    weakTyping();
}
```

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from `Integer` to `Float` explicit. In fact, it significantly changes the behavior of the computation. While the Ada code performs a floating point operation $1.0/10.0$ and stores 0.1 in `Result`, the C version instead store 0.0 in `result`. This is because the C version perform an integer operation between two integer variables: $1/10$ is 0. The result of the integer division is then converted to a `float` and stored. Errors of this sort can be very hard to locate in complex pieces of code, and systematic specification of how the operation should be interpreted

helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to `Float`:

[Ada]

```
-- Perform an Integer division then convert to Float
Result := Float (Alpha / Beta);
```

The complete example would then be:

[Ada]

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

procedure Strong_Typing is
   Alpha  : constant Integer := 1;
   Beta   : constant Integer := 10;
   Result : Float;
begin
   Result := Float (Alpha / Beta);

   Put_Line (Float'Image (Result));
end Strong_Typing;
```

Floating Point Literals

In Ada, a floating point literal must be written with both an integral and decimal part. `10` is not a valid literal for a floating point value, while `10.0` is.

2.11.2 Language-Defined Types

The principal scalar types predefined by Ada are `Integer`, `Float`, `Boolean`, and `Character`. These correspond to `int`, `float`, `int` (when used for Booleans), and `char`, respectively. The names for these types are not reserved words; they are regular identifiers. There are other language-defined integer and floating-point types as well. All have implementation-defined ranges and precision.

2.11.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators +, -, *, and / for integer and floating-point types take operands of the same type and return a value of that type.

[Ada]

```
procedure Main is
    type Distance is new Float;
    type Area is new Float;
```

(continues on next page)

(continued from previous page)

```
   D1 : Distance := 2.0;
   D2 : Distance := 3.0;
   A  : Area;
begin
   D1 := D1 + D2; -- OK
   D1 := D1 + A;  -- NOT OK: incompatible types for "+"
   A  := D1 * D2; -- NOT OK: incompatible types for ":="
   A  := Area (D1 * D2); -- OK
end Main;
```

Even though the `Distance` and `Area` types above are just `Float`, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two `Distance` yields a `Distance`) and prohibit some useful cases (for example multiplying two `Distances` should deliver an `Area`). These situations can be handled through other mechanisms. A predefined operation can be identified as abstract to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

Ada enumerations work similarly to C `enum`:

[Ada]

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

   D : Day := Monday;
begin
   null;
end Main;
```

[C]

```
enum Day {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
};

int main(int argc, const char * argv[])
{
    enum Day d = Monday;
}
```

But even though such enumerations may be implemented by the compiler as numeric values, at the language level Ada will not confuse the fact that `Monday` is a `Day` and is not an `Integer`. You can compare a `Day` with another `Day`, though. To specify implementation details like the numeric

values that correspond with enumeration values in C you include them in the original `enum` declaration:

[C]

```c
#include <stdio.h>

enum Day {
    Monday    = 10,
    Tuesday   = 11,
    Wednesday = 12,
    Thursday  = 13,
    Friday    = 14,
    Saturday  = 15,
    Sunday    = 16
};

int main(int argc, const char * argv[])
{
    enum Day d = Monday;

    printf("d = %d\n", d);
}
```

But in Ada you must use both a type definition for `Day` as well as a separate representation clause for it like:

[Ada]

```ada
with Ada.Text_IO;

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

   --  Representation clause for Day type:
   for Day use
     (Monday    => 10,
      Tuesday   => 11,
      Wednesday => 12,
      Thursday  => 13,
      Friday    => 14,
      Saturday  => 15,
      Sunday    => 16);

   D : Day := Monday;
   V : Integer;
begin
   V := Day'Enum_Rep (D);
   Ada.Text_IO.Put_Line (Integer'Image (V));
end Main;
```

Note that however, unlike C, values for enumerations in Ada have to be unique.

2.11.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a *range constraint* introduced with the `range` reserved word, for example:

[Ada]

```
procedure Main is
   type Grade is range 0 .. 100;

   G1, G2  : Grade;
   N       : Integer;
begin
   --  ...              --  Initialization of N
   G1 := 80;            --  OK
   G1 := N;             --  Illegal (type mismatch)
   G1 := Grade (N);     --  Legal, run-time range check
   G2 := G1 + 10;       --  Legal, run-time range check
   G1 := (G1 + G2) / 2; --  Legal, run-time range check
end Main;
```

In the above example, `Grade` is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to `G1` is correct and will not raise a run-time exception. Assigning `N` to `G1` is illegal since `Grade` is a different type than `Integer`. Converting `N` to `Grade` makes the assignment legal, and a range check on the conversion confirms that the value is within `0 .. 100`. Assigning `G1 + 10` to `G2` is legal since `+` for `Grade` returns a `Grade` (note that the literal `10` is interpreted as a `Grade` value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression `G1 + G2` may be outside the range of `Grade`, but the final result will be in range. Nevertheless, depending on the representation chosen for `Grade`, the addition may overflow. If the compiler represents `Grade` values as signed 8-bit integers (i.e., machine numbers in the range `-128 .. 127`) then the sum `G1 + G2` may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

[Ada]

```
with Ada.Text_IO;

procedure Main is
   type Grade is range 0 .. 100;

   G1, G2 : Grade := 99;
begin
   G1 := Grade ((Integer (G1) + Integer (G2)) / 2);
   Ada.Text_IO.Put_Line (Grade'Image (G1));
end Main;
```

Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the `Day` example, one can write:

[Ada]

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

   type Business_Day is new Day range Monday .. Friday;
   type Weekend_Day is new Day range Saturday .. Sunday;
begin
   null;
end Main;
```

Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind *subtypes* in Ada. A subtype is a type with optional additional constraints. For example:

[Ada]

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

   subtype Business_Day is Day range Monday .. Friday;
   subtype Weekend_Day is Day range Saturday .. Sunday;
   subtype Dice_Throw is Integer range 1 .. 6;
begin
   null;
end Main;
```

These declarations don't create new types, just new names for constrained ranges of their base types.

The purpose of numeric ranges is to express some application-specific constraint that we want the compiler to help us enforce. More importantly, we want the compiler to tell us when that constraint cannot be met — when the underlying hardware cannot support the range given. There are two things to consider:

- just a range constraint, such as `A : Integer range 0 .. 10;`, or
- a type declaration, such as `type Result is range 0 .. 1_000_000_000;`.

Both represent some sort of application-specific constraint, but in addition, the type declaration promotes portability because it won't compile on targets that do not have a sufficiently large hardware numeric type. That's a definition of portability that is preferable to having something compile anywhere but not run correctly, as in C.

2.11.5 Unsigned And Modular Types

Unsigned integer numbers are quite common in embedded applications. In C, you can use them by declaring `unsigned int` variables. In Ada, you have two options:

- declare custom *unsigned* range types;
  - In addition, you can declare custom range *subtypes* or use existing subtypes such as `Natural`.
- declare custom modular types.

The following table presents the main features of each type. We discuss these types right after.

Feature	[C] `unsigned int`	[Ada] Unsigned range	[Ada] Modular
Excludes negative value	✓	✓	✓
Wraparound	✓		✓

When declaring custom range types in Ada, you may use the full range in the same way as in C. For example, this is the declaration of a 32-bit unsigned integer type and the X variable in Ada:

[Ada]

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

procedure Main is
   type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;

   X : Unsigned_Int_32 := 42;
begin
   Put_Line ("X = " & Unsigned_Int_32'Image (X));
end Main;
```

In C, when `unsigned int` has a size of 32 bits, this corresponds to the following declaration:

[C]

```
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
{
    unsigned int x = 42;
    printf("x = %u\n", x);
}
```

Another strategy is to declare subtypes for existing signed types and specify just the range that excludes negative numbers. For example, let's declare a custom 32-bit signed type and its unsigned subtype:

[Ada]

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

procedure Main is
   type Signed_Int_32 is range -2 ** 31 .. 2 ** 31 - 1;

   subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. Signed_Int_32'Last;
   --  Equivalent to:
   --  subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. 2 ** 31 - 1;

   X : Unsigned_Int_31 := 42;
begin
   Put_Line ("X = " & Unsigned_Int_31'Image (X));
end Main;
```

In this case, we're just skipping the sign bit of the `Signed_Int_32` type. In other words, while `Signed_Int_32` has a size of 32 bits, `Unsigned_Int_31` has a range of 31 bits, even if the base type has 32 bits.

Note that the declaration above is actually similar to the existing `Natural` subtype. Ada provides the following standard subtypes:

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

Since they're standard subtypes, you can declare variables of those subtypes directly in your implementation, in the same way as you can declare `Integer` variables.

As indicated in the table above, however, there is a difference in behavior for the variables we just declared, which occurs in case of overflow. Let's consider this C example:

[C]

```
#include <stdio.h>
#include <limits.h>

int main(int argc, const char * argv[])
{
    unsigned int x = UINT_MAX + 1;
    /* Now: x == 0 */

    printf("x = %u\n", x);
}
```

The corresponding code in Ada raises an exception:

[Ada]

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

procedure Main is
   type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;

   X : Unsigned_Int_32 := Unsigned_Int_32'Last + 1;
   --  Overflow: exception is raised!
begin
   Put_Line ("X = " & Unsigned_Int_32'Image (X));
end Main;
```

While the C uses modulo arithmetic for unsigned integer, Ada doesn't use it for the `Unsigned_Int_32` type. Ada does, however, support modular types via type definitions using the `mod` keyword. In this example, we declare a 32-bit modular type:

[Ada]

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

procedure Main is
   type Unsigned_32 is mod 2**32;

   X : Unsigned_32 := Unsigned_32'Last + 1;
   --  Now: X = 0
begin
   Put_Line ("X = " & Unsigned_32'Image (X));
end Main;
```

In this case, the behavior is the same as in the C declaration above.

Modular types, unlike Ada's signed integers, also provide bit-wise operations, a typical application for unsigned integers in C. In Ada, you can use operators such as `and`, `or`, `xor` and `not`. You can also use typical bit-shifting operations, such as `Shift_Left`, `Shift_Right`, `Shift_Right_Arithmetic`, `Rotate_Left` and `Rotate_Right`.

2.11.6 Attributes

Attributes start with a single apostrophe ("tick"), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The `'Image` and `'Value` attributes allow you to transform a scalar value into a `String` and vice-versa. For example:

[Ada]

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

procedure Main is
   A : Integer := 10;
begin
   Put_Line (Integer'Image (A));
   A := Integer'Value ("99");
   Put_Line (Integer'Image (A));
end Main;
```

Important

Semantically, attributes are equivalent to subprograms. For example, `Integer'Image` is defined as follows:

```
function Integer'Image(Arg : Integer'Base) return String;
```

Certain attributes are provided only for certain kinds of types. For example, the `'Val` and `'Pos` attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]

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

procedure Main is
   C : Character := 'a';
begin
   Put (C);
   C := Character'Val (Character'Pos (C) + 1);
   Put (C);
end Main;
```

A more concise way to get the next value in Ada is to use the `'Succ` attribute:

[Ada]

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

procedure Main is
   C : Character := 'a';
begin
   Put (C);
   C := Character'Succ (C);
   Put (C);
end Main;
```

You can get the previous value using the `'Pred` attribute. Here is the equivalent in C:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    char c = 'a';
    printf("%c", c);
    c++;
    printf("%c", c);
}
```

Other interesting examples are the `'First` and `'Last` attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, `Integer'First` returns -2^{31} and `Integer'Last` returns $2^{31} - 1$.

2.11.7 Arrays and Strings

C arrays are pointers with offsets, but the same is not the case for Ada. Arrays in Ada are not interchangeable with operations on pointers, and array types are considered first-class citizens. They have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds — the lower bound doesn't necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here's an example of declaring an array of 26 characters, initializing the values from `'a'` to `'z'`:

[Ada]

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

procedure Main is
   type Arr_Type is array (Integer range <>) of Character;
   Arr : Arr_Type (1 .. 26);
   C : Character := 'a';
begin
   for I in Arr'Range loop
      Arr (I) := C;
      C := Character'Succ (C);

      Put (Arr (I) & " ");
   end loop;
end Main;
```

[C]

```
#include <stdio.h>

void main(void)
{
    char Arr [26];
    char C = 'a';

    for (int I = 0; I < 26; ++I) {
        Arr [I] = C++;
        printf ("%c ", Arr [I]);
    }
}
```

In C, only the size of the array is given during declaration. In Ada, array index ranges are specified using two values of a discrete type. In this example, the array type declaration specifies the use of `Integer` as the index type, but does not provide any constraints (use <>, pronounced *box*, to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called `'Range`. In our example, `Arr'Range` can also be expressed as `Arr'First .. Arr'Last`; both expressions will resolve to `1 .. 26`. So the `'Range` attribute supplies the bounds for our `for` loop. There is no risk of stating either of the bounds incorrectly, as one might do in C where `I <= 26` may be specified as the end-of-loop condition.

As in C, Ada `String` is an array of `Character`. Ada strings, importantly, are not delimited with the special character `'\0'` like they are in C. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined `String` type is very straightforward to use:

[Ada]

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

procedure Main is
   My_String : String (1 .. 19) := "This is an example!";
begin
   Put_Line (My_String);
end Main;
```

Unlike C, Ada does not offer escape sequences such as `'\n'`. Instead, explicit values from the ASCII package must be concatenated (via the concatenation operator, &). Here for example, is how to initialize a line of text ending with a new line:

[Ada]

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

procedure Main is
   My_String : String := "This is a line" & ASCII.LF;
begin
   Put (My_String);
end Main;
```

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of `My_String`'s bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada, the actual array contents are duplicated. To get the above behavior, actual pointer types would have to be defined and used.

[Ada]

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

procedure Main is
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type (1 .. 2);
   A2 : Arr_Type (1 .. 2);
begin
   A1 (1) := 0;
   A1 (2) := 1;

   A2 := A1;

   for I in A2'Range loop
      Put_Line (Integer'Image (A2 (I)));
```

(continues on next page)

(continued from previous page)

```
   end loop;
end Main;
```

[C]

```
#include <stdio.h>
#include <string.h>

int main(int argc, const char * argv[])
{
    int A1 [2];
    int A2 [2];

    A1 [0] = 0;
    A1 [1] = 1;

    memcpy (A2, A1, sizeof (int) * 2);

    for (int i = 0; i < 2; i++) {
        printf("%d\n", A2[i]);
    }
}
```

In all of the examples above, the source and destination arrays must have precisely the same number of elements. Ada allows you to easily specify a portion, or slice, of an array. So you can write the following:

[Ada]

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

procedure Main is
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type (1 .. 10) := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
   A2 : Arr_Type (1 .. 5)  := (1, 2, 3, 4, 5);
begin
   A2 (1 .. 3) := A1 (4 .. 6);

   for I in A2'Range loop
      Put_Line (Integer'Image (A2 (I)));
   end loop;
end Main;
```

This assigns the 4th, 5th, and 6th elements of A1 into the 1st, 2nd, and 3rd elements of A2. Note that only the length matters here: the values of the indexes don't have to be equal; they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as opposed to their addresses:

[Ada]

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

procedure Main is
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type (1 .. 2) := (10, 20);
   A2 : Arr_Type (1 .. 2) := (10, 20);
begin
   if A1 = A2 then
     Put_Line ("A1 = A2");
   else
```

(continues on next page)

(continued from previous page)

```
     Put_Line ("A1 /= A2");
   end if;
end Main;
```

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int A1 [2] = { 10, 20 };
    int A2 [2] = { 10, 20 };

    int eq = 1;

    for (int i = 0; i < 2; ++i) {
        if (A1 [i] != A2 [i]) {
            eq = 0;
            break;
        }
    }

    if (eq) {
        printf("A1 == A2\n");
    }
    else {
        printf("A1 != A2\n");
    }
}
```

You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the others expression to define a default value for all the unspecified array elements. Therefore, you can write:

[Ada]

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

procedure Main is
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type (-2 .. 42) := (others => 0);
begin
   --  use a slice to assign A1 elements 11 .. 19 to 1
   A1 (11 .. 19) := (others => 1);

   Put_Line ("---- A1 ----");
   for I in A1'Range loop
      Put_Line (Integer'Image (I) & " => " &
                Integer'Image (A1 (I)));
   end loop;
end Main;
```

In this example, we're specifying that `A1` has a range between -2 and 42. We use `(others => 0)` to initialize all array elements with zero. In the next example, the number of elements is determined by looking at the right-hand side:

[Ada]

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

procedure Main is
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
begin
   A1 := (1, 2, 3, others => 10);

   Put_Line ("---- A1 ----");
   for I in A1'Range loop
      Put_Line (Integer'Image (I) & " => " &
                Integer'Image (A1 (I)));
   end loop;
end Main;
```

Since `A1` is initialized with an aggregate of 9 elements, `A1` automatically has 9 elements. Also, we're not specifying any range in the declaration of `A1`. Therefore, the compiler uses the default range of the underlying array type `Arr_Type`, which has an unconstrained range based on the `Integer` type. The compiler selects the first element of that type (`Integer'First`) as the start index of `A1`. If you replaced `Integer range <>` in the declaration of the `Arr_Type` by `Positive range <>`, then `A1`'s start index would be `Positive'First` — which corresponds to one.

2.11.8 Heterogeneous Data Structures

The structure corresponding to a C `struct` is an Ada `record`. Here are some simple records:

[Ada]

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

procedure Main is
   type R is record
      A, B : Integer;
      C    : Float;
   end record;

   V : R;
begin
   V.A := 0;
   Put_Line ("V.A = " & Integer'Image (V.A));
end Main;
```

[C]

```
#include <stdio.h>

struct R {
    int A, B;
    float C;
};

int main(int argc, const char * argv[])
{
    struct R V;
    V.A = 0;
    printf("V.A = %d\n", V.A);
}
```

Ada allows specification of default values for fields just like C. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by others => <> to specify that fields not listed will take their default values. For example:

[Ada]

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

procedure Main is

   type R is record
      A, B : Integer := 0;
      C    : Float   := 0.0;
   end record;

   procedure Put_R (V : R; Name : String) is
   begin
      Put_Line (Name & " = ("
                & Integer'Image (V.A) & ", "
                & Integer'Image (V.B) & ", "
                & Float'Image (V.C) & ")");
   end Put_R;

   V1 : constant R := (1, 2, 1.0);
   V2 : constant R := (A => 1, B => 2, C => 1.0);
   V3 : constant R := (C => 1.0, A => 1, B => 2);
   V4 : constant R := (C => 1.0, others => <>);

begin
   Put_R (V1, "V1");
   Put_R (V2, "V2");
   Put_R (V3, "V3");
   Put_R (V4, "V4");
end Main;
```

2.11.9 Pointers

As a foreword to the topic of pointers, it's important to keep in mind the fact that most situations that would require a pointer in C do not in Ada. In the vast majority of cases, indirect memory management can be hidden from the developer and thus saves from many potential errors. However, there are situation that do require the use of pointers, or said differently that require to make memory indirection explicit. This section will present Ada access types, the equivalent of C pointers. A further section will provide more details as to how situations that require pointers in C can be done without access types in Ada.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

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

procedure Main is
   type R is record
      A, B : Integer;
   end record;

   procedure Put_R (V : R; Name : String) is
   begin
      Put_Line (Name & " = ("
                & Integer'Image (V.A) & ", "
                & Integer'Image (V.B) & ")");
   end Put_R;
```

(continues on next page)

(continued from previous page)

```
   V1, V2 : R;

begin
   V1.A := 0;
   V2 := V1;
   V2.A := 1;

   Put_R (V1, "V1");
   Put_R (V2, "V2");
end Main;
```

[C]

```
#include <stdio.h>

struct R {
    int A, B;
};

void print_r(const struct R *v,
             const char     *name)
{
    printf("%s = (%d, %d)\n", name, v->A, v->B);
}

int main(int argc, const char * argv[])
{
    struct R V1, V2;
    V1.A = 0;
    V2 = V1;
    V2.A = 1;

    print_r(&V1, "V1");
    print_r(&V2, "V2");
}
```

There are many commonalities between the Ada and C semantics above. In Ada and C, objects are allocated on the stack and are directly accessed. V1 and V2 are two different objects and the assignment statement copies the value of V1 into V2. V1 and V2 are two distinct objects.

Here's now a similar example, but using heap allocation instead:

[Ada]

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

procedure Main is
   type R is record
      A, B : Integer;
   end record;

   type R_Access is access R;

   procedure Put_R (V : R; Name : String) is
   begin
      Put_Line (Name & " = ("
                & Integer'Image (V.A) & ", "
                & Integer'Image (V.B) & ")");
   end Put_R;

   V1 : R_Access;
   V2 : R_Access;
```

(continues on next page)

(continued from previous page)

```
begin
   V1 := new R;
   V1.A := 0;
   V2 := V1;
   V2.A := 1;

   Put_R (V1.all, "V1");
   Put_R (V2.all, "V2");
end Main;
```

[C]

```
#include <stdio.h>
#include <stdlib.h>

struct R {
    int A, B;
};

void print_r(const struct R *v,
             const char     *name)
{
    printf("%s = (%d, %d)\n", name, v->A, v->B);
}

int main(int argc, const char * argv[])
{
    struct R * V1, * V2;
    V1 = malloc(sizeof(struct R));
    V1->A = 0;
    V2 = V1;
    V2->A = 1;

    print_r(V1, "V1");
    print_r(V2, "V2");
}
```

In this example, an object of type R is allocated on the heap. The same object is then referred to through V1 and V2. As in C, there's no garbage collector in Ada, so objects allocated by the new operator need to be expressly freed (which is not the case here).

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append `.all`. The equivalent of V1->A in C can be written either as `V1.A` or `V1.all.A`.

Pointers to scalar objects in Ada and C look like:

[Ada]

```
procedure Main is
   type A_Int is access Integer;
   Var : A_Int := new Integer;
begin
   Var.all := 0;
end Main;
```

[C]

```
#include <stdlib.h>

int main(int argc, const char * argv[])
```

(continues on next page)

(continued from previous page)

```
{
    int * Var = malloc (sizeof(int));
    *Var = 0;
    return 0;
}
```

In Ada, an initializer can be specified with the allocation by appending `'(value)`:

[Ada]

```
procedure Main is
   type A_Int is access Integer;

   Var : A_Int := new Integer'(0);
begin
   null;
end Main;
```

When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being aliased. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either `access all` (if the referenced object needs to be assigned to) or `access constant` (if the referenced object is a constant). The `'Access` attribute works like the C & operator to get a pointer to the object, but with a *scope accessibility* check to prevent references to objects that have gone out of scope. For example:

[Ada]

```
procedure Main is
   type A_Int is access all Integer;
   Var : aliased Integer;
   Ptr : A_Int := Var'Access;
begin
   null;
end Main;
```

[C]

```
int main(int argc, const char * argv[])
{
    int Var;
    int * Ptr = &Var;
}
```

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs, it's called `Ada.Unchecked_Deallocation`. To create your customized deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

[Ada]

```
with Ada.Unchecked_Deallocation;

procedure Main is
   type Integer_Access is access all Integer;
   procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);
   My_Pointer : Integer_Access := new Integer;
begin
   Free (My_Pointer);
end Main;
```

[C]

```c
#include <stdlib.h>

int main(int argc, const char * argv[])
{
    int * my_pointer = malloc (sizeof(int));
    free (my_pointer);
}
```

We'll discuss generics later *in this section* (page 121).

2.12 Functions and Procedures

2.12.1 General Form

Subroutines in C are always expressed as functions which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term *subprogram* to refer to both functions and procedures.

Parameters can be passed in three distinct modes:

- `in`, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram.
- `out` is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller.
- `in out` is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant pointer in C).

Ada also provides `access` and `aliased` parameters, which are in effect explicit pass-by-reference indicators.

In Ada, the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). C has the programmer specify how to pass the parameter.

Important

There are some exceptions to the "general" rule in Ada. For example, parameters of scalar types are always passed by copy, for all three modes.

Here's a first example:

[Ada]

```
procedure Proc
 (Var1 : Integer;
  Var2 : out Integer;
  Var3 : in out Integer);
```

```
function Func (Var : Integer) return Integer;
```

```
with Func;

procedure Proc
 (Var1 : Integer;
```

(continues on next page)

(continued from previous page)

```
  Var2 : out Integer;
  Var3 : in out Integer)
is
begin
   Var2 := Func (Var1);
   Var3 := Var3 + 1;
end Proc;
```

```
function Func (Var : Integer) return Integer
is
begin
   return Var + 1;
end Func;
```

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

procedure Main is
   V1, V2 : Integer;
begin
   V2 := 2;
   Proc (5, V1, V2);

   Put_Line ("V1: " & Integer'Image (V1));
   Put_Line ("V2: " & Integer'Image (V2));
end Main;
```

[C]

```
void Proc
    (int   Var1,
     int * Var2,
     int * Var3);
```

```
int Func (int Var);
```

```
#include "func.h"

void Proc
    (int   Var1,
     int * Var2,
     int * Var3)
{
    *Var2 = Func (Var1);
    *Var3 += 1;
}
```

```
int Func (int Var)
{
    return Var + 1;
}
```

```
#include <stdio.h>
#include "proc.h"

void main (void)
{
    int v1, v2;
```

(continues on next page)

(continued from previous page)

```
    v2 = 2;
    Proc (5, &v1, &v2);

    printf("v1: %d\n", v1);
    printf("v2: %d\n", v2);
}
```

The first two declarations for `Proc` and `Func` are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C, a function that has not yet been seen cannot be used. Here, `Proc` can call `Func` because its specification has been declared.

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax (except for the modes), including default values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.

In Ada 202X

Ada 202X allows for using static expression functions, which are evaluated at compile time. To achieve this, we can use an aspect — we'll discuss aspects *later in this chapter* (page 47).

An expression function is static when the `Static` aspect is specified. For example:

```
procedure Main is

   X1 : constant := (if True then 37 else 42);

   function If_Then_Else (Flag : Boolean; X, Y : Integer)
     return Integer is
      (if Flag then X else Y) with Static;

   X2 : constant := If_Then_Else (True, 37, 42);

begin
   null;
end Main;
```

In this example, we declare X1 using an expression. In the declaration of X2, we call the static expression function `If_Then_Else`. Both X1 and X2 have the same constant value.

2.12.2 Overloading

In C, function names must be unique. Ada allows overloading, in which multiple subprograms can share the same name as long as the subprogram signatures (the parameter types, and function return types) are different. The compiler will be able to resolve the calls to the proper routines or it will reject the calls. For example:

[Ada]

```
package Machine is
   type Status is (Off, On);
   type Code is new Integer range 0 .. 3;
   type Threshold is new Float range 0.0 .. 10.0;

   function Get (S : Status) return Code;
```

(continues on next page)

(continued from previous page)

```
   function Get (S : Status) return Threshold;

end Machine;
```

```
package body Machine is

   function Get (S : Status) return Code is
   begin
      case S is
         when Off => return 1;
         when On  => return 3;
      end case;
   end Get;

   function Get (S : Status) return Threshold is
   begin
      case S is
         when Off => return  2.0;
         when On  => return 10.0;
      end case;
   end Get;

end Machine;
```

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

procedure Main is
   S : Status;
   C : Code;
   T : Threshold;
begin
   S := On;
   C := Get (S);
   T := Get (S);

   Put_Line ("S: " & Status'Image (S));
   Put_Line ("C: " & Code'Image (C));
   Put_Line ("T: " & Threshold'Image (T));
end Main;
```

The Ada compiler knows that an assignment to `C` requires a `Code` value. So, it chooses the `Get` function that returns a `Code` to satisfy this requirement.

Operators in Ada are functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To declare an operator as a function, enclose its "name" in quotes:

[Ada]

```
package Machine_2 is
   type Status is (Off, Waiting, On);
   type Input is new Float range 0.0 .. 10.0;

   function Get (I : Input) return Status;

   function "=" (Left : Input; Right : Status) return Boolean;

end Machine_2;
```

```
package body Machine_2 is

   function Get (I : Input) return Status is
   begin
      if I >= 0.0 and I < 3.0 then
         return Off;
      elsif I >= 3.0 and I < 6.5 then
         return Waiting;
      else
         return On;
      end if;
   end Get;

   function "=" (Left : Input; Right : Status) return Boolean is
   begin
      return Get (Left) = Right;
   end "=";

end Machine_2;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Machine_2;   use Machine_2;

procedure Main is
   I : Input;
begin
   I := 3.0;
   if I = Off then
      Put_Line ("Machine is off.");
   else
      Put_Line ("Machine is not off.");
   end if;
end Main;
```

2.12.3 Aspects

Aspect specifications allow you to define certain characteristics of a declaration using the `with` keyword after the declaration:

```
procedure Some_Procedure is <procedure_definition>
  with Some_Aspect => <aspect_specification>;

function Some_Function is <function_definition>
  with Some_Aspect => <aspect_specification>;

type Some_Type is <type_definition>
  with Some_Aspect => <aspect_specification>;

Obj : Some_Type with Some_Aspect => <aspect_specification>;
```

For example, you can inline a subprogram by specifying the `Inline` aspect:

[Ada]

```
package Float_Arrays is

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

   function Average (Data : Float_Array) return Float
     with Inline;
```

(continues on next page)

(continued from previous page)

```
end Float_Arrays;
```

We'll discuss inlining *later in this course* (page 160).

Aspect specifications were introduced in Ada 2012. In previous versions of Ada, you had to use a `pragma` instead. The previous example would be written as follows:

[Ada]

```
package Float_Arrays is

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

   function Average (Data : Float_Array) return Float;

   pragma Inline (Average);

end Float_Arrays;
```

Aspects and attributes might refer to the same kind of information. For example, we can use the `Size` aspect to define the expected minimum size of objects of a certain type:

[Ada]

```
package My_Device_Types is

   type UInt10 is mod 2 ** 10
     with Size => 10;

end My_Device_Types;
```

In the same way, we can use the size attribute to retrieve the size of a type or of an object:

[Ada]

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

with My_Device_Types; use My_Device_Types;

procedure Show_Device_Types is
   UInt10_Obj : constant UInt10 := 0;
begin
   Put_Line ("Size of UInt10 type:   " & Positive'Image (UInt10'Size));
   Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_Obj'Size));
end Show_Device_Types;
```

We'll explain both `Size` aspect and `Size` attribute *later in this course* (page 72).

CHAPTER

THREE

CONCURRENCY AND REAL-TIME

3.1 Understanding the various options

Concurrent and real-time programming are standard parts of the Ada language. As such, they have the same semantics, whether executing on a native target with an OS such as Linux, on a real-time operating system (RTOS) such as VxWorks, or on a bare metal target with no OS or RTOS at all.

For resource-constrained systems, two subsets of the Ada concurrency facilities are defined, known as the Ravenscar and Jorvik profiles. Though restricted, these subsets have highly desirable properties, including: efficiency, predictability, analyzability, absence of deadlock, bounded blocking, absence of priority inversion, a real-time scheduler, and a small memory footprint. On bare metal systems, this means in effect that Ada comes with its own real-time kernel.

For further information

We'll discuss the Ravenscar profile *later in this chapter* (page 57). Details about the Jorvik profile can be found elsewhere [Jorvik].

Enhanced portability and expressive power are the primary advantages of using the standard concurrency facilities, potentially resulting in considerable cost savings. For example, with little effort, it is possible to migrate from Windows to Linux to a bare machine without requiring any changes to the code. Thread management and synchronization is all done by the implementation, transparently. However, in some situations, it's critical to be able to access directly the services provided by the platform. In this case, it's always possible to make direct system calls from Ada code. Several targets of the GNAT compiler provide this sort of API by default, for example win32ada for Windows and Florist for POSIX systems.

On native and RTOS-based platforms GNAT typically provides the full concurrency facilities. In contrast, on bare metal platforms GNAT typically provides the two standard subsets: Ravenscar and Jorvik.

3.2 Tasks

Ada offers a high level construct called a *task* which is an independent thread of execution. In GNAT, tasks are either mapped to the underlying OS threads, or use a dedicated kernel when not available.

The following example will display the 26 letters of the alphabet twice, using two concurrent tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]

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

procedure Main is -- implicitly called by the environment task
   subtype A_To_Z is Character range 'A' .. 'Z';

   task My_Task;

   task body My_Task is
   begin
      for I in A_To_Z'Range loop
         Put (I);
      end loop;
      New_Line;
   end My_Task;
begin
   for I in A_To_Z'Range loop
      Put (I);
   end loop;
   New_Line;
end Main;
```

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the begin. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

Task types can be parameterized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the `'Z'` Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the underlying implementation of the task scheduling algorithm.

[Ada]

```
package My_Tasks is

   task type My_Task (First : Character);

end My_Tasks;
```

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

package body My_Tasks is

   task body My_Task is
   begin
      for I in First .. 'Z' loop
         Put (I);
      end loop;
      New_Line;
   end My_Task;

end My_Tasks;
```

```
with My_Tasks; use My_Tasks;

procedure Main is
```

(continues on next page)

(continued from previous page)

```
   Dummy_Tab : array (0 .. 9) of My_Task ('G');
begin
   null;
end Main;
```

In Ada, a task may be dynamically allocated rather than declared statically. The task will then start as soon as it has been allocated, and terminates when its work is completed.

[Ada]

```
with My_Tasks; use My_Tasks;

procedure Main is
   type Ptr_Task is access My_Task;

   T : Ptr_Task;
begin
   T := new My_Task ('G');
end Main;
```

3.3 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Let's consider the following example:

[Ada]

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

procedure Main is

   task After is
      entry Go;
   end After;

   task body After is
   begin
      accept Go;
      Put_Line ("After");
   end After;

begin
   Put_Line ("Before");
   After.Go;
end Main;
```

The `Go` entry declared in `After` is the client interface to the task. In the task body, the `accept` statement causes the task to wait for a call on the entry. This particular `entry` and `accept` pair simply causes the task to wait until `Main` calls `After.Go`. So, even though the two tasks start simultaneously and execute independently, they can coordinate via `Go`. Then, they both continue execution independently after the rendezvous.

The `entry`/`accept` pair can take/pass parameters, and the `accept` statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

[Ada]

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

procedure Main is

   task After is
      entry Go (Text : String);
   end After;

   task body After is
   begin
      accept Go (Text : String) do
         Put_Line ("After: " & Text);
      end Go;
   end After;

begin
   Put_Line ("Before");
   After.Go ("Main");
end Main;
```

In the above example, the `Put_Line` is placed in the accept statement. Here's a possible execution trace, assuming a uniprocessor:

1. At the begin of `Main`, task `After` is started and the main procedure is suspended.
2. `After` reaches the `accept` statement and is suspended, since there is no pending call on the `Go` entry.
3. The main procedure is awakened and executes the `Put_Line` invocation, displaying the string `"Before"`.
4. The main procedure calls the `Go` entry. Since `After` is suspended on its `accept` statement for this entry, the call succeeds.
5. The main procedure is suspended, and the task `After` is awakened to execute the body of the `accept` statement. The actual parameter `"Main"` is passed to the `accept` statement, and the `Put_Line` invocation is executed. As a result, the string `"After: Main"` is displayed.
6. When the `accept` statement is completed, both the `After` task and the main procedure are ready to run. Suppose that the `Main` procedure is given the processor. It reaches its end, but the local task `After` has not yet terminated. The main procedure is suspended.
7. The `After` task continues, and terminates since it is at its end. The main procedure is resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform various optimizations to avoid unnecessary context switches.

3.4 Selective Rendezvous

The `accept` statement by itself can only wait for a single event (call) at a time. The `select` statement allows a task to listen for multiple events simultaneously, and then to deal with the first event to occur. This feature is illustrated by the task below, which maintains an integer value that is modified by other tasks that call `Increment`, `Decrement`, and `Get`:

[Ada]

```
package Counters is

   task Counter is
      entry Get (Result : out Integer);
```

(continues on next page)

(continued from previous page)

```
      entry Increment;
      entry Decrement;
   end Counter;

end Counters;
```

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

package body Counters is

   task body Counter is
      Value : Integer := 0;
   begin
      loop
         select
            accept Increment do
               Value := Value + 1;
            end Increment;
         or
            accept Decrement do
               Value := Value - 1;
            end Decrement;
         or
            accept Get (Result : out Integer) do
               Result := Value;
            end Get;
         or
            delay 5.0;
            Put_Line ("Exiting Counter task...");
            exit;
         end select;
      end loop;
   end Counter;

end Counters;
```

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

procedure Main is
   V : Integer;
begin
   Put_Line ("Main started.");

   Counter.Get (V);
   Put_Line ("Got value. Value = " & Integer'Image (V));

   Counter.Increment;
   Put_Line ("Incremented value.");

   Counter.Increment;
   Put_Line ("Incremented value.");

   Counter.Get (V);
   Put_Line ("Got value. Value = " & Integer'Image (V));

   Counter.Decrement;
   Put_Line ("Decremented value.");

   Counter.Get (V);
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("Got value. Value = " & Integer'Image (V));

   Put_Line ("Main finished.");
end Main;
```

When the task's statement flow reaches the select, it will wait for all four events — three entries and a delay — in parallel. If the delay of five seconds is exceeded, the task will execute the statements following the `delay` statement (and in this case will exit the loop, in effect terminating the task). The `accept` bodies for the `Increment`, `Decrement`, or `Get` entries will be otherwise executed as they're called. These four sections of the select statement are mutually exclusive: at each iteration of the loop, only one will be invoked. This is a critical point; if the task had been written as a package, with procedures for the various operations, then a *race condition* could occur where multiple tasks simultaneously calling, say, `Increment`, cause the value to only get incremented once. In the tasking version, if multiple tasks simultaneously call `Increment` then only one at a time will be accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one of the entries and Counter is not ready to accept the call (i.e., if `Counter` is not suspended at the `select` statement) then the calling task is suspended, and placed in the queue of the entry that it is calling. From the perspective of the `Counter` task, at any iteration of the loop there are several possibilities:

- There is no call pending on any of the entries. In this case `Counter` is suspended. It will be awakened by the first of two events: a call on one of its entries (which will then be immediately accepted), or the expiration of the five second delay (whose effect was noted above).
- There is a call pending on exactly one of the entries. In this case control passes to the `select` branch with an `accept` statement for that entry.
- There are calls pending on more than one entry. In this case one of the entries with pending callers is chosen, and then one of the callers is chosen to be de-queued. The choice of which caller to accept depends on the queuing policy, which can be specified via a `pragma` defined in the Real-Time Systems Annex of the Ada standard; the default is *First-In First-Out*.

3.5 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared data object, an alternative (and generally preferable) style is through a protected object, an efficiently implementable mechanism that makes the effect more explicit. A protected object has a public interface (its protected operations) for accessing and manipulating the object's components (its private part). Mutual exclusion is enforced through a conceptual lock on the object, and encapsulation ensures that the only external access to the components are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by procedures or entries, and read-only operations by functions. The lock mechanism is implemented so that it's possible to perform concurrent read operations but not concurrent write or read/write operations.

Let's reimplement our earlier tasking example with a protected object called `Counter`:

[Ada]

```
package Counters is

   protected Counter is
      function Get return Integer;
      procedure Increment;
      procedure Decrement;
   private
      Value : Integer := 0;
```

(continues on next page)

(continued from previous page)

```
   end Counter;

end Counters;
```

```
package body Counters is

   protected body Counter is
      function Get return Integer is
      begin
         return Value;
      end Get;

      procedure Increment is
      begin
         Value := Value + 1;
      end Increment;

      procedure Decrement is
      begin
         Value := Value - 1;
      end Decrement;
   end Counter;

end Counters;
```

Having two completely different ways to implement the same paradigm might seem complicated. However, in practice the actual problem to solve usually drives the choice between an active structure (a task) or a passive structure (a protected object).

A protected object can be accessed through prefix notation:

[Ada]

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

procedure Main is
begin
   Counter.Increment;
   Counter.Decrement;
   Put_Line (Integer'Image (Counter.Get));
end Main;
```

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```
declare
   protected type Counter is
      -- as above
   end Counter;

   protected body Counter is
      -- as above
   end Counter;

   C1 : Counter;
```

(continues on next page)

(continued from previous page)

```
   C2 : Counter;
begin
   C1.Increment;
   C2.Decrement;
   -- .
end;
```

Protected objects and types can declare a procedure-like operation known as an *entry*. An entry is somewhat similar to a procedure but includes a so-called barrier condition that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

[Ada]

```
package Binary_Semaphores is

   protected type Binary_Semaphore is
      entry Wait;
      procedure Signal;
   private
      Signaled : Boolean := False;
   end Binary_Semaphore;

end Binary_Semaphores;
```

```
package body Binary_Semaphores is

   protected body Binary_Semaphore is
      entry Wait when Signaled is
      begin
         Signaled := False;
      end Wait;

      procedure Signal is
      begin
         Signaled := True;
      end Signal;
   end Binary_Semaphore;

end Binary_Semaphores;
```

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

procedure Main is
   B : Binary_Semaphore;

   task T1;
   task T2;

   task body T1 is
   begin
      Put_Line ("Task T1 waiting...");
      B.Wait;
```

(continues on next page)

(continued from previous page)

```
      Put_Line ("Task T1.");
      delay 1.0;

      Put_Line ("Task T1 will signal...");
      B.Signal;

      Put_Line ("Task T1 finished.");
   end T1;

   task body T2 is
   begin
      Put_Line ("Task T2 waiting...");
      B.Wait;

      Put_Line ("Task T2");
      delay 1.0;

      Put_Line ("Task T2 will signal...");
      B.Signal;

      Put_Line ("Task T2 finished.");
   end T2;

begin
   Put_Line ("Main started.");
   B.Signal;
   Put_Line ("Main finished.");
end Main;
```

Ada concurrency features provide much further generality than what's been presented here. For additional information please consult one of the works cited in the *References* section.

3.6 Ravenscar

The Ravenscar profile is a subset of the Ada concurrency facilities that supports determinism, schedulability analysis, constrained memory utilization, and certification to the highest integrity levels. Four distinct application domains are intended:

- hard real-time applications requiring predictability,
- safety-critical systems requiring formal, stringent certification,
- high-integrity applications requiring formal static analysis and verification,
- embedded applications requiring both a small memory footprint and low execution overhead.

Tasking constructs that preclude analysis, either technically or economically, are disallowed. You can use the `pragma Profile (Ravenscar)` to indicate that the Ravenscar restrictions must be observed in your program.

Some of the examples we've seen above will be rejected by the compiler when using the Ravenscar profile. For example:

[Ada]

```
package My_Tasks is

   task type My_Task (First : Character);
```

(continues on next page)

(continued from previous page)

```
end My_Tasks;

with Ada.Text_IO; use Ada.Text_IO;

package body My_Tasks is

   task body My_Task is
   begin
      for C in First .. 'Z' loop
         Put (C);
      end loop;
   end My_Task;

end My_Tasks;

pragma Profile (Ravenscar);

with My_Tasks; use My_Tasks;

procedure Main is
   Tab : array (0 .. 9) of My_Task ('G');
begin
   null;
end Main;
```

This code violates the *No_Task_Hierarchy* restriction of the Ravenscar profile. This is due to the declaration of `Tab` in the `Main` procedure. Ravenscar requires task declarations to be done at the library level. Therefore, a simple solution is to create a separate package and reference it in the main application:

[Ada]

```
with My_Tasks; use My_Tasks;

package My_Task_Inst is

   Tab : array (0 .. 9) of My_Task ('G');

end My_Task_Inst;

pragma Profile (Ravenscar);

with My_Task_Inst;

procedure Main is
begin
   null;
end Main;
```

Also, Ravenscar prohibits entries for tasks. For example, we're not allowed to write this declaration:

```
task type My_Task (First : Character) is
   entry Start;
end My_Task;
```

You can use, however, one entry per protected object. As an example, the declaration of the `Binary_Semaphore` type that we've discussed before compiles fine with Ravenscar:

```
protected type Binary_Semaphore is
   entry Wait;
```

(continues on next page)

(continued from previous page)

```
   procedure Signal;
private
   Signaled : Boolean := False;
end Binary_Semaphore;
```

We could add more procedures and functions to the declaration of `Binary_Semaphore`, but we wouldn't be able to add another entry when using Ravenscar.

Similar to the previous example with the task array declaration, objects of `Binary_Semaphore` cannot be declared in the main application:

```
procedure Main is
   B : Binary_Semaphore;
begin
   null;
end Main;
```

This violates the *No_Local_Protected_Objects* restriction. Again, Ravenscar expects this declaration to be done on a library level, so a solution to make this code compile is to have this declaration in a separate package and reference it in the `Main` procedure.

Ravenscar offers many additional restrictions. Covering those would exceed the scope of this chapter. You can find more examples using the Ravenscar profile on this blog post[10].

[10] https://blog.adacore.com/theres-a-mini-rtos-in-my-language

CHAPTER

FOUR

WRITING ADA ON EMBEDDED SYSTEMS

4.1 Understanding the Ada Run-Time

Ada supports a high level of abstractness and expressiveness. In some cases, the compiler translates those constructs directly into machine code. However, there are many high-level constructs for which a direct compilation would be difficult. In those cases, the compiler links to a library containing an implementation of those high-level constructs: this is the so-called run-time library.

One typical example of high-level constructs that can be cumbersome for direct machine code generation is Ada source-code using tasking. In this case, linking to a low-level implementation of multithreading support — for example, an implementation using POSIX threads — is more straightforward than trying to make the compiler generate all the machine code.

In the case of GNAT, the run-time library is implemented using both C and Ada source-code. Also, depending on the operating system, the library will interface with low-level functionality from the target operating system.

There are basically two types of run-time libraries:

- the **standard** run-time library: in many cases, this is the run-time library available on desktop operating systems or on some embedded platforms (such as ARM-Linux on a Raspberry-Pi).
- the **configurable** run-time library: this is a capability that is used to create custom run-time libraries for specific target devices.

Configurable run-time libraries are usually used for constrained target devices where support for the full library would be difficult or even impossible. In this case, configurable run-time libraries may support just a subset of the full Ada language. There are many reasons that speak for this approach:

- Some aspects of the Ada language may not translate well to limited operating systems.
- Memory constraints may require reducing the size of the run-time library, so that developers may need to replace or even remove parts of the library.
- When certification is required, those parts of the library that would require too much certification effort can be removed.

When using a configurable run-time library, the compiler checks whether the library supports certain features of the language. If a feature isn't supported, the compiler will give an error message.

You can find further information about the run-time library on this chapter of the GNAT User's Guide Supplement for Cross Platforms[11]

[11] https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx/the_gnat_configurable_run_time_facility.html

4.2 Low Level Programming

4.2.1 Representation Clauses

We've seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let's start with the following example:

[Ada]

```
type R is record
   V  : Integer range 0 .. 255;
   B1 : Boolean;
   B2 : Boolean;
end record
  with Pack;
```

[C]

```
struct R {
    unsigned int v:8;
    bool b1;
    bool b2;
};
```

The Ada and the C++ code above both represent efforts to create an object that's as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C++ and Ada code are equivalent in this particular example, there's an interesting semantic difference. In C++, the number of bits required by each field needs to be specified. Here, we're stating that v is only 8 bits, effectively representing values from 0 to 255. In Ada, it's the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The `Pack` aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components. We'll see more details about the `Pack` aspect in the sections about *bitwise operations* (page 104) and *mapping structures to bit-fields* (page 106) in chapter 6.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here's how to specify a specific data layout based on the previous example:

[Ada]

```
type R is record
   V  : Integer range 0 .. 255;
   B1 : Boolean;
   B2 : Boolean;
end record;

for R use record
   --  Occupy the first bit of the first byte.
   B1 at 0 range 0 .. 0;
```

(continues on next page)

(continued from previous page)

```
   --  Occupy the last 7 bits of the first byte,
   --  as well as the first bit of the second byte.
   V  at 0 range 1 .. 8;

   --  Occupy the second bit of the second byte.
   B2 at 1 range 1 .. 1;
end record;
```

We omit the `with Pack` directive and instead use a record representation clause following the record declaration. The compiler is directed to spread objects of type R across two bytes. The layout we're specifying here is fairly inefficient to work with on any machine, but you can have the compiler construct the most efficient methods for access, rather than coding your own machine-dependent bit-level methods manually.

4.2.2 Embedded Assembly Code

When performing low-level development, such as at the kernel or hardware driver level, there can be times when it is necessary to implement functionality with assembly code.

Every Ada compiler has its own conventions for embedding assembly code, based on the hardware platform and the supported assembler(s). Our examples here will work with GNAT and GCC on the x86 architecture.

All x86 processors since the Intel Pentium offer the `rdtsc` instruction, which tells us the number of cycles since the last processor reset. It takes no inputs and places an unsigned 64-bit value split between the `edx` and `eax` registers.

GNAT provides a subprogram called `System.Machine_Code.Asm` that can be used for assembly code insertion. You can specify a string to pass to the assembler as well as source-level variables to be used for input and output:

[Ada]

```
with System.Machine_Code; use System.Machine_Code;
with Interfaces;          use Interfaces;

function Get_Processor_Cycles return Unsigned_64 is
   Low, High : Unsigned_32;
   Counter   : Unsigned_64;
begin
   Asm ("rdtsc",
        Outputs =>
          (Unsigned_32'Asm_Output ("=a", High),
           Unsigned_32'Asm_Output ("=d", Low)),
        Volatile => True);

   Counter :=
     Unsigned_64 (High) * 2 ** 32 +
     Unsigned_64 (Low);

   return Counter;
end Get_Processor_Cycles;
```

The `Unsigned_32'Asm_Output` clauses above provide associations between machine registers and source-level variables to be updated. `=a` and `=d` refer to the `eax` and `edx` machine registers, respectively. The use of the `Unsigned_32` and `Unsigned_64` types from package `Interfaces` ensures correct representation of the data. We assemble the two 32-bit values to form a single 64-bit value.

We set the `Volatile` parameter to `True` to tell the compiler that invoking this instruction multiple

times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the `eax` and `edx` registers to implement the `High` and `Low` variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User's Guide, and the GNAT Reference manual.

4.3 Interrupt Handling

Handling interrupts is an important aspect when programming embedded devices. Interrupts are used, for example, to indicate that a hardware or software event has happened. Therefore, by handling interrupts, an application can react to external events.

Ada provides built-in support for handling interrupts. We can process interrupts by attaching a handler — which must be a protected procedure — to it. In the declaration of the protected procedure, we use the `Attach_Handler` aspect and indicate which interrupt we want to handle.

Let's look into a code example that *traps* the quit interrupt (`SIGQUIT`) on Linux:

[Ada]

```
with System.OS_Interface;

package Signal_Handlers is

   protected type Quit_Handler is
      function Requested return Boolean;
   private
      Quit_Request : Boolean := False;

      --
      --  Declaration of an interrupt handler for the "quit" interrupt:
      --
      procedure Handle_Quit_Signal
        with Attach_Handler => System.OS_Interface.SIGQUIT;
   end Quit_Handler;

end Signal_Handlers;
```

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

package body Signal_Handlers is

   protected body Quit_Handler is

      function Requested return Boolean is
        (Quit_Request);

      procedure Handle_Quit_Signal is
      begin
         Put_Line ("Quit request detected!");
         Quit_Request := True;
      end Handle_Quit_Signal;

   end Quit_Handler;

end Signal_Handlers;
```

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

procedure Test_Quit_Handler is
   Quit : Signal_Handlers.Quit_Handler;

begin
   while True loop
      delay 1.0;
      exit when Quit.Requested;
   end loop;

   Put_Line ("Exiting application...");
end Test_Quit_Handler;
```

The specification of the `Signal_Handlers` package from this example contains the declaration of `Quit_Handler`, which is a protected type. In the private part of this protected type, we declare the `Handle_Quit_Signal` procedure. By using the `Attach_Handler` aspect in the declaration of `Handle_Quit_Signal` and indicating the quit interrupt (`System.OS_Interface.SIGQUIT`), we're instructing the operating system to call this procedure for any quit request. So when the user presses CTRL+\ on their keyboard, for example, the application will behave as follows:

- the operating system calls the `Handle_Quit_Signal` procedure , which displays a message to the user ("Quit request detected!") and sets a Boolean variable — `Quit_Request`, which is declared in the `Quit_Handler` type;
- the main application checks the status of the quit handler by calling the `Requested` function as part of the `while True` loop;
  - This call is in the `exit when Quit.Requested` line.
  - The `Requested` function returns `True` in this case because the `Quit_Request` flag was set by the `Handle_Quit_Signal` procedure.
- the main applications exits the loop, displays a message and finishes.

Note that the code example above isn't portable because it makes use of interrupts from the Linux operating system. When programming embedded devices, we would use instead the interrupts available on those specific devices.

Also note that, in the example above, we're declaring a static handler at compilation time. If you need to make use of dynamic handlers, which can be configured at runtime, you can use the subprograms from the `Ada.Interrupts` package. This package includes not only a version of `Attach_Handler` as a procedure, but also other procedures such as:

- `Exchange_Handler`, which lets us exchange, at runtime, the current handler associated with a specific interrupt by a different handler;
- `Detach_Handler`, which we can use to remove the handler currently associated with a given interrupt.

Details about the `Ada.Interrupts` package are out of scope for this course. We'll discuss them in a separate, more advanced course in the future. You can find some information about it in the Interrupts appendix of the Ada Reference Manual[12].

Todo: Once available, add link to section from a more advanced embedded course that explains the `Ada.Interrupts` package.

---

[12] https://www.adaic.org/resources/add_content/standards/12aarm/html/AA-C-3-2.html

4.4 Dealing with Absence of FPU with Fixed Point

Many numerical applications typically use floating-point types to compute values. However, in some platforms, a floating-point unit may not be available. Other platforms may have a floating-point unit, but using it in certain numerical algorithms can be prohibitive in terms of performance. For those cases, fixed-point arithmetic can be a good alternative.

The difference between fixed-point and floating-point types might not be so obvious when looking at this code snippet:

[Ada]

```
package Fixed_Definitions is

   D : constant := 2.0 ** (-31);

   type Fixed is delta D range -1.0 .. 1.0 - D;

end Fixed_Definitions;
```

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

with Fixed_Definitions; use Fixed_Definitions;

procedure Show_Float_And_Fixed_Point is
   Float_Value : Float := 0.25;
   Fixed_Value : Fixed := 0.25;
begin

   Float_Value := Float_Value + 0.25;
   Fixed_Value := Fixed_Value + 0.25;

   Put_Line ("Float_Value = " & Float'Image (Float_Value));
   Put_Line ("Fixed_Value = " & Fixed'Image (Fixed_Value));
end Show_Float_And_Fixed_Point;
```

In this example, the application will show the value 0.5 for both `Float_Value` and `Fixed_Value`.

The major difference between floating-point and fixed-point types is in the way the values are stored. Values of ordinary fixed-point types are, in effect, scaled integers. The scaling used for ordinary fixed-point types is defined by the type's `small`, which is derived from the specified `delta` and, by default, is a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types. In that sense, ordinary fixed-point types can be thought of being close to the actual representation on the machine. In fact, ordinary fixed-point types make use of the available integer shift instructions, for example.

Another difference between floating-point and fixed-point types is that Ada doesn't provide standard fixed-point types — except for the `Duration` type, which is used to represent an interval of time in seconds. While the Ada standard specifies floating-point types such as `Float` and `Long_Float`, we have to declare our own fixed-point types. Note that, in the previous example, we have used a fixed-point type named `Fixed`: this type isn't part of the standard, but must be declared somewhere in the source-code of our application.

The syntax for an ordinary fixed-point type is

```
type <type_name> is delta <delta_value> range <lower_bound> .. <upper_bound>;
```

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

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

[Ada]

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

procedure Normalized_Fixed_Point_Type is
   D : constant := 2.0 ** (-31);
   type TQ31 is delta D range -1.0 .. 1.0 - D;
begin
   Put_Line ("TQ31 requires " & Integer'Image (TQ31'Size) & " bits");
   Put_Line ("The delta    value of TQ31 is " & TQ31'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 exactly one. This is a typical effect of fixed-point data types — you can find more details in this discussion about the Q format[13]. We may also rewrite this code with an exact type definition:

[Ada]

```
package Normalized_Adapted_Fixed_Point_Type is

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

end Normalized_Adapted_Fixed_Point_Type;
```

We may also use any other range. For example:

[Ada]

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

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

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

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

[Ada]

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

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

   A, B, R : TQ31;
begin
   A := 0.25;
   B := 0.50;
```

(continues on next page)

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

(continued from previous page)

```
   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 the case of C, since the language doesn't support fixed-point arithmetic, we need to emulate it using integer types and custom operations via functions. Let's look at this very rudimentary example:

[C]

```
#include <stdio.h>
#include <math.h>

#define SHIFT_FACTOR  32

#define TO_FIXED(x)     ((int)   ((x) * pow (2.0, SHIFT_FACTOR - 1)))
#define TO_FLOAT(x)     ((float) ((double)(x) * (double)pow (2.0, -(SHIFT_FACTOR -␣
↪1))))

typedef int fixed;

fixed add (fixed a, fixed b)
{
    return a + b;
}

fixed mult (fixed a, fixed b)
{
    return (fixed)(((long)a * (long)b) >> (SHIFT_FACTOR - 1));
}

void display_fixed (fixed x)
{
    printf("value (integer) = %d\n",    x);
    printf("value (float)   = %3.5f\n\n", TO_FLOAT (x));
}

int main(int argc, const char * argv[])
{
    int fixed_value = TO_FIXED(0.25);

    printf("Original value\n");
    display_fixed(fixed_value);

    printf("... + 0.25\n");
    fixed_value = add(fixed_value, TO_FIXED(0.25));
    display_fixed(fixed_value);

    printf("... * 0.5\n");
    fixed_value = mult(fixed_value, TO_FIXED(0.5));
    display_fixed(fixed_value);
}
```

Here, we declare the fixed-point type `fixed` based on `int` and two operations for it: addition (via the `add` function) and multiplication (via the `mult` function). Note that, while fixed-point addition is quite straightforward, multiplication requires right-shifting to match the correct internal representation. In Ada, since fixed-point operations are part of the language specification, they don't need to be emulated. Therefore, no extra effort is required from the programmer.

Also note that the example above is very rudimentary, so it doesn't take some of the side-effects of

fixed-point arithmetic into account. In C, you have to manually take all side-effects deriving from fixed-point arithmetic into account, while in Ada, the compiler takes care of selecting the right operations for you.

4.5 Volatile and Atomic data

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

4.5.1 Volatile

A volatile[14] object can be described as an object in memory whose value may change between two consecutive memory accesses of a process A — even if process A itself hasn't changed the value. This situation may arise when an object in memory is being shared by multiple threads. For example, a thread *B* may modify the value of that object between two read accesses of a thread *A*. Another typical example is the one of memory-mapped I/O[15], where the hardware might be constantly changing the value of an object in memory.

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

Let's look at a simple example of a volatile variable in C:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    volatile double val = 0.0;
    int i;

    for (i = 0; i < 1000; i++)
    {
        val += i * 2.0;
    }
    printf ("val: %5.3f\n", val);
}
```

In this example, `val` has the modifier `volatile`, which indicates that the compiler must handle `val` as a volatile object. Therefore, each read and write access in the loop is performed by accessing the value of `val` in then memory.

This is the corresponding implementation in Ada:

[Ada]

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

procedure Show_Volatile_Object is
   Val : Long_Float with Volatile;
begin
```

(continues on next page)

[14] https://en.wikipedia.org/wiki/Volatile_(computer_programming)
[15] https://en.wikipedia.org/wiki/Memory-mapped_I/O

(continued from previous page)

```
   Val := 0.0;
   for I in 0 .. 999 loop
      Val := Val + 2.0 * Long_Float (I);
   end loop;

   Put_Line ("Val: " & Long_Float'Image (Val));
end Show_Volatile_Object;
```

In this example, `Val` has the `Volatile` aspect, which makes the object volatile. We can also use the `Volatile` aspect in type declarations. For example:

[Ada]

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

procedure Show_Volatile_Type is
   type Volatile_Long_Float is new Long_Float with Volatile;

   Val : Volatile_Long_Float;
begin
   Val := 0.0;
   for I in 0 .. 999 loop
      Val := Val + 2.0 * Volatile_Long_Float (I);
   end loop;

   Put_Line ("Val: " & Volatile_Long_Float'Image (Val));
end Show_Volatile_Type;
```

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

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

[Ada]

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

procedure Show_Volatile_Array_Components is
   Arr : array (1 .. 2) of Long_Float with Volatile_Components;
begin
   Arr := (others => 0.0);

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

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

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

[Ada]

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

4.5.2 Atomic

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

In other contexts

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

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

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

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

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

[Ada]

```
with System;

procedure Show_Shared_HW_Register is
   R   : Integer
     with Atomic, Address => System'To_Address (16#FFFF00A0#);
begin
   null;
end Show_Shared_HW_Register;
```

Note that the `Address` aspect allows for assigning a variable to a specific location in the memory. In this example, we're using this aspect to specify the address of the memory-mapped register. We'll discuss more about the `Address` aspect later in the section about *mapping structures to bit-fields* (page 106) (in chapter 6).

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

[Ada]

```
with System;

procedure Show_Shared_HW_Register is
   type Atomic_Integer is new Integer with Atomic;

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

(continues on next page)

(continued from previous page)

```
   Arr : array (1 .. 2) of Integer with Atomic_Components;
begin
   null;
end Show_Shared_HW_Register;
```

In this example, we're declaring the `Atomic_Integer` type, which is an atomic type. Objects of this type — such as R in this example — are automatically atomic. This example also includes the declaration of the `Arr` array, which has atomic components.

4.6 Interfacing with Devices

Previously, we've seen that we can use *representation clauses* (page 62) to specify a particular layout for a record type. As mentioned before, this is useful when interfacing with hardware, drivers, or communication protocols. In this section, we'll extend this concept for two specific use-cases: register overlays and data streams. Before we discuss those use-cases, though, we'll first explain the `Size` aspect and the `Size` attribute.

4.6.1 `Size` aspect and attribute

The `Size` aspect indicates the minimum number of bits required to represent an object. When applied to a type, the `Size` aspect is telling the compiler to not make record or array components of a type T any smaller than X bits. Therefore, a common usage for this aspect is to just confirm expectations: developers specify `'Size` to tell the compiler that T should fit X bits, and the compiler will tell them if they are right (or wrong).

When the specified size value is larger than necessary, it can cause objects to be bigger in memory than they would be otherwise. For example, for some enumeration types, we could say `for type Enum'Size use 32;` when the number of literals would otherwise have required only a byte. That's useful for unchecked conversions because the sizes of the two types need to be the same. Likewise, it's useful for interfacing with C, where `enum` types are just mapped to the `int` type, and thus larger than Ada might otherwise require. We'll discuss unchecked conversions *later in the course* (page 116).

Let's look at an example from an earlier chapter:

[Ada]

```
package My_Device_Types is

   type UInt10 is mod 2 ** 10
     with Size => 10;

end My_Device_Types;
```

Here, we're saying that objects of type `UInt10` must have at least 10 bits. In this case, if the code compiles, it is a confirmation that such values can be represented in 10 bits when packed into an enclosing record or array type.

If the size specified was larger than what the compiler would use by default, then it could affect the size of objects. For example, for `UInt10`, anything up to and including 16 would make no difference on a typical machine. However, anything over 16 would then push the compiler to use a larger object representation. That would be important for unchecked conversions, for example.

The `Size` attribute indicates the number of bits required to represent a type or an object. We can use the size attribute to retrieve the size of a type or of an object:

[Ada]

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

with My_Device_Types; use My_Device_Types;

procedure Show_Device_Types is
   UInt10_Obj : constant UInt10 := 0;
begin
   Put_Line ("Size of UInt10 type:   " & Positive'Image (UInt10'Size));
   Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_Obj'Size));
end Show_Device_Types;
```

Here, we're retrieving the actual sizes of the `UInt10` type and an object of that type. Note that the sizes don't necessarily need to match. For example, although the size of `UInt10` type is expected to be 10 bits, the size of `UInt10_Obj` may be 16 bits, depending on the platform. Also, components of this type within composite types (arrays, records) will probably be 16 bits as well unless they are packed.

4.6.2 Register overlays

Register overlays make use of representation clauses to create a structure that facilitates manipulating bits from registers. Let's look at a simplified example of a power management controller containing registers such as a system clock enable register. Note that this example is based on an actual architecture:

[Ada]

```
with System;

package Registers is

   type Bit    is mod 2 ** 1
     with Size => 1;
   type UInt5  is mod 2 ** 5
     with Size => 5;
   type UInt10 is mod 2 ** 10
     with Size => 10;

   subtype USB_Clock_Enable is Bit;

   --  System Clock Enable Register
   type PMC_SCER_Register is record
      --  Reserved bits
      Reserved_0_4   : UInt5           := 16#0#;
      --  Write-only. Enable USB FS Clock
      USBCLK         : USB_Clock_Enable := 16#0#;
      --  Reserved bits
      Reserved_6_15  : UInt10          := 16#0#;
   end record
     with
       Volatile,
       Size      => 16,
       Bit_Order => System.Low_Order_First;

   for PMC_SCER_Register use record
      Reserved_0_4   at 0 range 0 .. 4;
      USBCLK         at 0 range 5 .. 5;
      Reserved_6_15  at 0 range 6 .. 15;
   end record;
```

(continues on next page)

(continued from previous page)

```
   --  Power Management Controller
   type PMC_Peripheral is record
      --  System Clock Enable Register
      PMC_SCER       : aliased PMC_SCER_Register;
      --  System Clock Disable Register
      PMC_SCDR       : aliased PMC_SCER_Register;
   end record
     with Volatile;

   for PMC_Peripheral use record
      --  16-bit register at byte 0
      PMC_SCER       at 16#0# range 0 .. 15;
      --  16-bit register at byte 2
      PMC_SCDR       at 16#2# range 0 .. 15;
   end record;

   --  Power Management Controller
   PMC_Periph : aliased PMC_Peripheral
     with Import, Address => System'To_Address (16#400E0600#);

end Registers;
```

First, we declare the system clock enable register — this is `PMC_SCER_Register` type in the code example. Most of the bits in that register are reserved. However, we're interested in bit #5, which is used to activate or deactivate the system clock. To achieve a correct representation of this bit, we do the following:

- We declare the `USBCLK` component of this record using the `USB_Clock_Enable` type, which has a size of one bit; and
- we use a representation clause to indicate that the `USBCLK` component is specifically at bit #5 of byte #0.

After declaring the system clock enable register and specifying its individual bits as components of a record type, we declare the power management controller type — `PMC_Peripheral` record type in the code example. Here, we declare two 16-bit registers as record components of `PMC_Peripheral`. These registers are used to enable or disable the system clock. The strategy we use in the declaration is similar to the one we've just seen above:

- We declare these registers as components of the `PMC_Peripheral` record type;
- we use a representation clause to specify that the `PMC_SCER` register is at byte #0 and the `PMC_SCDR` register is at byte #2.
  - Since these registers have 16 bits, we use a range of bits from 0 to 15.

The actual power management controller becomes accessible by the declaration of the `PMC_Periph` object of `PMC_Peripheral` type. Here, we specify the actual address of the memory-mapped registers (*400E0600* in hexadecimal) using the `Address` aspect in the declaration. When we use the `Address` aspect in an object declaration, we're indicating the address in memory of that object.

Because we specify the address of the memory-mapped registers in the declaration of `PMC_Periph`, this object is now an overlay for those registers. This also means that any operation on this object corresponds to an actual operation on the registers of the power management controller. We'll discuss more details about overlays in the section about *mapping structures to bit-fields* (page 106) (in chapter 6).

Finally, in a test application, we can access any bit of any register of the power management controller with simple record component selection. For example, we can set the `USBCLK` bit of the `PMC_SCER` register by using `PMC_Periph.PMC_SCER.USBCLK`:

[Ada]

```
with Registers;

procedure Enable_USB_Clock is
begin
   Registers.PMC_Periph.PMC_SCER.USBCLK := 1;
end Enable_USB_Clock;
```

This code example makes use of many aspects and keywords of the Ada language. One of them is the `Volatile` aspect, which we've discussed in the section about *volatile and atomic objects* (page 69). Using the `Volatile` aspect for the `PMC_SCER_Register` type ensures that objects of this type won't be stored in a register.

In the declaration of the `PMC_SCER_Register` record type of the example, we use the `Bit_Order` aspect to specify the bit ordering of the record type. Here, we can select one of these options:

- `High_Order_First`: first bit of the record is the most significant bit;
- `Low_Order_First`: first bit of the record is the least significant bit.

The declarations from the `Registers` package also makes use of the `Import`, which is sometimes necessary when creating overlays. When used in the context of object declarations, it avoids default initialization (for data types that have it.). Aspect `Import` will be discussed in the section that explains how to *map structures to bit-fields* (page 106) in chapter 6. Please refer to that chapter for more details.

Details about `'Size`

In the example above, we're using the `Size` aspect in the declaration of the `PMC_SCER_Register` type. In this case, the effect is that it has the compiler confirm that the record type will fit into the expected 16 bits.

That's what the aspect does for type `PMC_SCER_Register` in the example above, as well as for the types `Bit`, `UInt5` and `UInt10`. For example, we may declare a stand-alone object of type `Bit`:

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

procedure Show_Bit_Declaration is

   type Bit    is mod 2 ** 1
     with Size => 1;

   B : constant Bit := 0;
   --  ^ Although Bit'Size is 1, B'Size is almost certainly 8
begin
   Put_Line ("Bit'Size = " & Positive'Image (Bit'Size));
   Put_Line ("B'Size   = " & Positive'Image (B'Size));
end Show_Bit_Declaration;
```

In this case, `B` is almost certainly going to be 8-bits wide on a typical machine, even though the language requires that `Bit'Size` is 1 by default.

In the declaration of the components of the `PMC_Peripheral` record type, we use the `aliased` keyword to specify that those record components are accessible via other paths besides the component name. Therefore, the compiler won't store them in registers. This makes sense because we want to ensure that we're accessing specific memory-mapped registers, and not registers assigned by the compiler. Note that, for the same reason, we also use the `aliased` keyword in the declaration of the `PMC_Periph` object.

4.6.3 Data streams

Creating data streams — in the context of interfacing with devices — means the serialization of arbitrary information and its transmission over a communication channel. For example, we might want to transmit the content of memory-mapped registers as byte streams using a serial port. To do this, we first need to get a serialized representation of those registers as an array of bytes, which we can then transmit over the serial port.

Serialization of arbitrary record types — including register overlays — can be achieved by declaring an array of bytes as an overlay. By doing this, we're basically interpreting the information from those record types as bytes while ignoring their actual structure — i.e. their components and representation clause. We'll discuss details about overlays in the section about *mapping structures to bit-fields* (page 106) (in chapter 6).

Let's look at a simple example of serialization of an arbitrary record type:

[Ada]

```
package Arbitrary_Types is

   type Arbitrary_Record is record
      A : Integer;
      B : Integer;
      C : Integer;
   end record;

end Arbitrary_Types;
```

```
with Arbitrary_Types;

procedure Serialize_Data (Some_Object : Arbitrary_Types.Arbitrary_Record);
```

```
with Arbitrary_Types;

procedure Serialize_Data (Some_Object : Arbitrary_Types.Arbitrary_Record) is
   type UByte is new Natural range 0 .. 255
     with Size => 8;

   type UByte_Array is array (Positive range <>) of UByte;

   --
   --  We can access the serialized data in Raw_TX, which is our overlay
   --
   Raw_TX : UByte_Array (1 .. Some_Object'Size / 8)
     with Address => Some_Object'Address;
begin
   null;
   --
   --  Now, we could stream the data from Some_Object.
   --
   --  For example, we could send the bytes (from Raw_TX) via the
   --  serial port.
   --
end Serialize_Data;
```

```
with Arbitrary_Types;
with Serialize_Data;

procedure Data_Stream_Declaration is
   Dummy_Object : Arbitrary_Types.Arbitrary_Record;
```

(continues on next page)

(continued from previous page)

```
begin
    Serialize_Data (Dummy_Object);
end Data_Stream_Declaration;
```

The most important part of this example is the implementation of the `Serialize_Data` procedure, where we declare `Raw_TX` as an overlay for our arbitrary object (`Some_Object` of `Arbitrary_Record` type). In simple terms, by writing `with Address => Some_Object'Address;` in the declaration of `Raw_TX`, we're specifying that `Raw_TX` and `Some_Object` have the same address in memory. Here, we are:

- taking the address of `Some_Object` — using the `Address` attribute —, and then
- using it as the address of `Raw_TX` — which is specified with the `Address` aspect.

By doing this, we're essentially saying that both `Raw_TX` and `Some_Object` are different representations of the same object in memory.

Because the `Raw_TX` overlay is completely agnostic about the actual structure of the record type, the `Arbitrary_Record` type could really be anything. By declaring `Raw_TX`, we create an array of bytes that we can use to stream the information from `Some_Object`.

We can use this approach and create a data stream for the register overlay example that we've seen before. This is the corresponding implementation:

[Ada]

```
with System;

package Registers is

   type Bit is mod 2 ** 1
     with Size => 1;
   type UInt5 is mod 2 ** 5
     with Size => 5;
   type UInt10 is mod 2 ** 10
     with Size => 10;

   subtype USB_Clock_Enable is Bit;

   --  System Clock Register
   type PMC_SCER_Register is record
      --  Reserved bits
      Reserved_0_4   : UInt5 := 16#0#;
      --  Write-only. Enable USB FS Clock
      USBCLK         : USB_Clock_Enable := 16#0#;
      --  Reserved bits
      Reserved_6_15  : UInt10 := 16#0#;
   end record
     with
       Volatile,
       Size      => 16,
       Bit_Order => System.Low_Order_First;

   for PMC_SCER_Register use record
      Reserved_0_4   at 0 range 0 .. 4;
      USBCLK         at 0 range 5 .. 5;
      Reserved_6_15  at 0 range 6 .. 15;
   end record;

   --  Power Management Controller
   type PMC_Peripheral is record
      --  System Clock Enable Register
```

(continues on next page)

(continued from previous page)

```
      PMC_SCER       : aliased PMC_SCER_Register;
      --  System Clock Disable Register
      PMC_SCDR       : aliased PMC_SCER_Register;
   end record
     with Volatile;

   for PMC_Peripheral use record
      --  16-bit register at byte 0
      PMC_SCER       at 16#0# range 0 .. 15;
      --  16-bit register at byte 2
      PMC_SCDR       at 16#2# range 0 .. 15;
   end record;

   --  Power Management Controller
   PMC_Periph : aliased PMC_Peripheral;
--     with Import, Address => System'To_Address (16#400E0600#);

end Registers;
```

```
package Serial_Ports is

   type UByte is new Natural range 0 .. 255
     with Size => 8;

   type UByte_Array is array (Positive range <>) of UByte;

   type Serial_Port is null record;

   procedure Read (Port : in out Serial_Port;
                   Data :    out UByte_Array);

   procedure Write (Port : in out Serial_Port;
                    Data :        UByte_Array);

end Serial_Ports;
```

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

package body Serial_Ports is

   procedure Display (Data : UByte_Array) is
   begin
      Put_Line ("---- Data ----");
      for E of Data loop
         Put_Line (UByte'Image (E));
      end loop;
      Put_Line ("--------------");
   end Display;

   procedure Read (Port : in out Serial_Port;
                   Data :    out UByte_Array) is
      pragma Unreferenced (Port);
   begin
      Put_Line ("Reading data...");
      Data := (0, 0, 32, 0);
   end Read;

   procedure Write (Port : in out Serial_Port;
                    Data :        UByte_Array) is
      pragma Unreferenced (Port);
```

(continues on next page)

(continued from previous page)

```
   begin
      Put_Line ("Writing data...");
      Display (Data);
   end Write;

end Serial_Ports;
```

```
with Serial_Ports; use Serial_Ports;
with Registers;    use Registers;

package Data_Stream is

   procedure Send (Port : in out Serial_Port;
                   PMC  :        PMC_Peripheral);

   procedure Receive (Port : in out Serial_Port;
                      PMC  :    out PMC_Peripheral);

end Data_Stream;
```

```
package body Data_Stream is

   procedure Send (Port : in out Serial_Port;
                   PMC  :        PMC_Peripheral)
   is
      Raw_TX : UByte_Array (1 .. PMC'Size / 8)
        with Address => PMC'Address;
   begin
      Write (Port => Port,
             Data => Raw_TX);
   end Send;

   procedure Receive (Port : in out Serial_Port;
                      PMC  :    out PMC_Peripheral)
   is
      Raw_TX : UByte_Array (1 .. PMC'Size / 8)
        with Address => PMC'Address;
   begin
      Read (Port => Port,
            Data => Raw_TX);
   end Receive;

end Data_Stream;
```

```
with Ada.Text_IO;

with Registers;
with Data_Stream;
with Serial_Ports;

procedure Test_Data_Stream is

   procedure Display_Registers is
      use Ada.Text_IO;
   begin
      Put_Line ("---- Registers ----");
      Put_Line ("PMC_SCER.USBCLK: "
                & Registers.PMC_Periph.PMC_SCER.USBCLK'Image);
      Put_Line ("PMC_SCDR.USBCLK: "
                & Registers.PMC_Periph.PMC_SCDR.USBCLK'Image);
```

(continues on next page)

(continued from previous page)

```
      Put_Line ("-------------- ----");
   end Display_Registers;

   Port : Serial_Ports.Serial_Port;
begin
   Registers.PMC_Periph.PMC_SCER.USBCLK := 1;
   Registers.PMC_Periph.PMC_SCDR.USBCLK := 1;

   Display_Registers;

   Data_Stream.Send (Port => Port,
                     PMC  => Registers.PMC_Periph);

   Data_Stream.Receive (Port => Port,
                        PMC  => Registers.PMC_Periph);

   Display_Registers;
end Test_Data_Stream;
```

In this example, we can find the overlay in the implementation of the `Send` and `Receive` procedures from the `Data_Stream` package. Because the overlay doesn't need to know the internals of the `PMC_Peripheral` type, we're declaring it in the same way as in the previous example (where we created an overlay for `Some_Object`). In this case, we're creating an overlay for the `PMC` parameter.

Note that, for this section, we're not really interested in the details about the serial port. Thus, package `Serial_Ports` in this example is just a stub. However, because the `Serial_Port` type in that package only *sees* arrays of bytes, after implementing an actual serial port interface for a specific device, we could create data streams for any type.

4.7 ARM and svd2ada

As we've seen in the previous section about *interfacing with devices* (page 72), Ada offers powerful features to describe low-level details about the hardware architecture without giving up its strong typing capabilities. However, it can be cumbersome to create a specification for all those low-level details when you have a complex architecture. Fortunately, for ARM Cortex-M devices, the GNAT toolchain offers an Ada binding generator called **svd2ada**, which takes CMSIS-SVD descriptions for those devices and creates Ada specifications that match the architecture. CMSIS-SVD description files are based on the Cortex Microcontroller Software Interface Standard (CMSIS), which is a hardware abstraction layer for ARM Cortex microcontrollers.

Please refer to the svd2ada project page[16] for details about this tool.

[16] https://github.com/AdaCore/svd2ada

CHAPTER

FIVE

ENHANCING VERIFICATION WITH SPARK AND ADA

5.1 Understanding Exceptions and Dynamic Checks

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering ”exceptions” that interrupt the normal flow of execution. For example, an exception is raised by an attempt to access an array component via an index that is out of bounds. This simple check precludes exploits based on buffer overflow. Several other cases also raise language-defined exceptions, such as scalar range constraint violations and null pointer dereferences. Developers may declare and raise their own application-specific exceptions too. (Exceptions are software artifacts, although an implementation may map hardware events to exceptions.)

Exceptions are raised during execution of what we will loosely define as a ”frame.” A frame is a language construct that has a call stack entry when called, for example a procedure or function body. There are a few other constructs that are also pertinent but this definition will suffice for now.

Frames have a sequence of statements implementing their functionality. They can also have optional ”exception handlers” that specify the response when exceptions are ”raised” by those statements. These exceptions could be raised directly within the statements, or indirectly via calls to other procedures and functions.

For example, the frame below is a procedure including three exceptions handlers:

```
procedure P is
begin
   Statements_That_Might_Raise_Exceptions;
exception
   when A =>
      Handle_A;
   when B =>
      Handle_B;
   when C =>
      Handle_C;
end P;
```

The three exception handlers each start with the word `when` (lines 5, 7, and 9). Next comes one or more exception identifiers, followed by the so-called ”arrow.” In Ada, the arrow always associates something on the left side with something on the right side. In this case, the left side is the exception name and the right side is the handler’s code for that exception.

Each handler’s code consists of an arbitrary sequence of statements, in this case specific procedures called in response to those specific exceptions. If exception `A` is raised we call procedure `Handle_A` (line 6), dedicated to doing the actual work of handling that exception. The other two exceptions are dealt with similarly, on lines 8 and 10.

Structurally, the exception handlers are grouped together and textually separated from the rest of the code in a frame. As a result, the sequence of statements representing the normal flow of execution is distinct from the section representing the error handling. The reserved word `exception`

separates these two sections (line 4 above). This separation helps simplify the overall flow, increasing understandability. In particular, status result codes are not required so there is no mixture of error checking and normal processing. If no exception is raised the exception handler section is automatically skipped when the frame exits.

Note how the syntactic structure of the exception handling section resembles that of an Ada case statement. The resemblance is intentional, to suggest similar behavior. When something in the statements of the normal execution raises an exception, the corresponding exception handler for that specific exception is executed. After that, the routine completes. The handlers do not "fall through" to the handlers below. For example, if exception `B` is raised, procedure `Handle_B` is called but `Handle_C` is not called. There's no need for a `break` statement, just as there is no need for it in a case statement. (There's no break statement in Ada anyway.)

So far, we've seen a frame with three specific exceptions handled. What happens if a frame has no handler for the actual exception raised? In that case the run-time library code goes "looking" for one.

Specifically, the active exception is propagated up the dynamic call chain. At each point in the chain, normal execution in that caller is abandoned and the handlers are examined. If that caller has a handler for the exception, the handler is executed. That caller then returns normally to its caller and execution continues from there. Otherwise, propagation goes up one level in the call chain and the process repeats. The search continues until a matching handler is found or no callers remain. If a handler is never found the application terminates abnormally. If the search reaches the main procedure and it has a matching handler it will execute the handler, but, as always, the routine completes so once again the application terminates.

For a concrete example, consider the following:

```
package Arrays is

   type List is array (Natural range <>) of Integer;

   function Value (A : List; X, Y : Integer) return Integer;

end Arrays;
```

```
package body Arrays is

   function Value (A : List; X, Y : Integer) return Integer is
   begin
      return A (X + Y * 10);
   end Value;

end Arrays;
```

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

procedure Some_Process is
   L : constant List (1 .. 100) := (others => 42);
begin
   Put_Line (Integer'Image (Value (L, 1, 10)));
exception
   when Constraint_Error =>
      Put_Line ("Constraint_Error caught in Some_Process");
      Put_Line ("Some_Process completes normally");
end Some_Process;
```

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

(continues on next page)

(continued from previous page)

```
procedure Main is
begin
   Some_Process;
   Put_Line ("Main completes normally");
end Main;
```

Procedure `Main` calls `Some_Process`, which in turn calls function `Value` (line 7). `Some_Process` declares the array object L of type `List` on line 5, with bounds 1 through 100. The call to `Value` has arguments, including variable L, leading to an attempt to access an array component via an out-of-bounds index (`1 + 10 * 10 = 101`, beyond the last index of L). This attempt will trigger an exception in `Value` prior to actually accessing the array object's memory. Function `Value` doesn't have any exception handlers so the exception is propagated up to the caller `Some_Process`. Procedure `Some_Process` has an exception handler for `Constraint_Error` and it so happens that `Constraint_Error` is the exception raised in this case. As a result, the code for that handler will be executed, printing some messages on the screen. Then procedure `Some_Process` will return to `Main` normally. `Main` then continues to execute normally after the call to `Some_Process` and prints its completion message.

If procedure `Some_Process` had also not had a handler for `Constraint_Error`, that procedure call would also have returned abnormally and the exception would have been propagated further up the call chain to procedure `Main`. Normal execution in `Main` would likewise be abandoned in search of a handler. But `Main` does not have any handlers so `Main` would have completed abnormally, immediately, without printing its closing message.

This semantic model is the same as with many other programming languages, in which the execution of a frame's sequence of statements is unavoidably abandoned when an exception becomes active. The model is a direct reaction to the use of status codes returned from functions as in C, where it is all too easy to forget (intentionally or otherwise) to check the status values returned. With the exception model errors cannot be ignored.

However, full exception propagation as described above is not the norm for embedded applications when the highest levels of integrity are required. The run-time library code implementing exception propagation can be rather complex and expensive to certify. Those problems apply to the application code too, because exception propagation is a form of control flow without any explicit construct in the source. Instead of the full exception model, designers of high-integrity applications often take alternative approaches.

One alternative consists of deactivating exceptions altogether, or more precisely, deactivating language-defined checks, which means that the compiler will not generate code checking for conditions giving rise to exceptions. Of course, this makes the code vulnerable to attacks, such as buffer overflow, unless otherwise verified (e.g. through static analysis). Deactivation can be applied at the unit level, through the `-gnatp` compiler switch, or locally within a unit via the pragma `Suppress`. (Refer to the GNAT User's Guide for Native Platforms[17] for more details about the switch.)

For example, we can write the following. Note the pragma on line 4 of `arrays.adb` within function `Value`:

```
package Arrays is

   type List is array (Natural range <>) of Integer;

   function Value (A : List; X, Y : Integer) return Integer;

end Arrays;
```

```
package body Arrays is

   function Value (A : List; X, Y : Integer) return Integer is
```

(continues on next page)

[17] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html

(continued from previous page)

```
      pragma Suppress (All_Checks);
   begin
      return A (X + Y * 10);
   end Value;

end Arrays;
```

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

procedure Some_Process is
   L : constant List (1 .. 100) := (others => 42);
begin
   Put_Line (Integer'Image (Value (L, 1, 10)));
exception
   when Constraint_Error =>
      Put_Line ("FAILURE");
end Some_Process;
```

This placement of the pragma will only suppress checks in the function body. However, that is where the exception would otherwise have been raised, leading to incorrect and unpredictable execution. (Run the program more than once. If it prints the right answer (42), or even the same value each time, it's just a coincidence.) As you can see, suppressing checks negates the guarantee of errors being detected and addressed at run-time.

Another alternative is to leave checks enabled but not retain the dynamic call-chain propagation. There are a couple of approaches available in this alternative.

The first approach is for the run-time library to invoke a global "last chance handler" (LCH) when any exception is raised. Instead of the sequence of statements of an ordinary exception handler, the LCH is actually a procedure intended to perform "last-wishes" before the program terminates. No exception handlers are allowed. In this scheme "propagation" is simply a direct call to the LCH procedure. The default LCH implementation provided by GNAT does nothing other than loop infinitely. Users may define their own replacement implementation.

The availability of this approach depends on the run-time library. Typically, *Zero Footprint* and *Ravenscar SFP* run-times will provide this mechanism because they are intended for certification.

A user-defined LCH handler can be provided either in C or in Ada, with the following profiles:

[Ada]

```
procedure Last_Chance_Handler (Source_Location : System.Address; Line : Integer);
pragma Export (C,
               Last_Chance_Handler,
               "__gnat_last_chance_handler");
```

[C]

```
void __gnat_last_chance_handler (char *source_location,
                                 int line);
```

We'll go into the details of the pragma `Export` in a further section on language interfacing. For now, just know that the symbol `__gnat_last_chance_handler` is what the run-time uses to branch immediately to the last-chance handler. Pragma `Export` associates that symbol with this replacement procedure so it will be invoked instead of the default routine. As a consequence, the actual procedure name in Ada is immaterial.

Here is an example implementation that simply blinks an LED forever on the target:

```
procedure Last_Chance_Handler (Msg : System.Address; Line : Integer) is
   pragma Unreferenced (Msg, Line);

   Next_Release : Time := Clock;
   Period       : constant Time_Span := Milliseconds (500);
begin
   Initialize_LEDs;
   All_LEDs_Off;

   loop
      Toggle (LCH_LED);
      Next_Release := Next_Release + Period;
      delay until Next_Release;
   end loop;
end Last_Chance_Handler;
```

The LCH_LED is a constant referencing the LED used by the last-chance handler, declared elsewhere. The infinite loop is necessary because a last-chance handler must never return to the caller (hence the term "last-chance"). The LED changes state every half-second.

Unlike the approach in which there is only the last-chance handler routine, the other approach allows exception handlers, but in a specific, restricted manner. Whenever an exception is raised, the only handler that can apply is a matching handler located in the same frame in which the exception is raised. Propagation in this context is simply an immediate branch instruction issued by the compiler, going directly to the matching handler's sequence of statements. If there is no matching local handler the last chance handler is invoked. For example consider the body of function `Value` in the body of package `Arrays`:

```
package Arrays is

   type List is array (Natural range <>) of Integer;

   function Value (A : List; X, Y : Integer) return Integer;

end Arrays;
```

```
package body Arrays is

   function Value (A : List; X, Y : Integer) return Integer is
   begin
      return A (X + Y * 10);
   exception
      when Constraint_Error =>
         return 0;
   end Value;

end Arrays;
```

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

procedure Some_Process is
   L : constant List (1 .. 100) := (others => 42);
begin
   Put_Line (Integer'Image (Value (L, 1, 10)));
exception
   when Constraint_Error =>
      Put_Line ("FAILURE");
end Some_Process;
```

In both procedure `Some_Process` and function `Value` we have an exception handler for

`Constraint_Error`. In this example the exception is raised in `Value` because the index check fails there. A local handler for that exception is present so the handler applies and the function returns zero, normally. Because the call to the function returns normally, the execution of `Some_Process` prints zero and then completes normally.

Let's imagine, however, that function `Value` did *not* have a handler for `Constraint_Error`. In the context of full exception propagation, the function call would return to the caller, i.e., `Some_Process`, and would be handled in that procedure's handler. But only local handlers are allowed under the second alternative so the lack of a local handler in `Value` would result in the last-chance handler being invoked. The handler for `Constraint_Error` in `Some_Process` under this alternative approach.

So far we've only illustrated handling the `Constraint_Error` exception. It's possible to handle other language-defined and user-defined exceptions as well, of course. It is even possible to define a single handler for all other exceptions that might be encountered in the handled sequence of statements, beyond those explicitly named. The "name" for this otherwise anonymous exception is the Ada reserved word `others`. As in case statements, it covers all other choices not explicitly mentioned, and so must come last. For example:

```
package Arrays is

   type List is array (Natural range <>) of Integer;

   function Value (A : List; X, Y : Integer) return Integer;

end Arrays;
```

```
package body Arrays is

   function Value (A : List; X, Y : Integer) return Integer is
   begin
      return A (X + Y * 10);
   exception
      when Constraint_Error =>
         return 0;
      when others =>
         return -1;
   end Value;

end Arrays;
```

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

procedure Some_Process is
   L : constant List (1 .. 100) := (others => 42);
begin
   Put_Line (Integer'Image (Value (L, 1, 10)));
exception
   when Constraint_Error =>
      Put_Line ("FAILURE");
end Some_Process;
```

In the code above, the `Value` function has a handler specifically for `Constraint_Error` as before, but also now has a handler for all other exceptions. For any exception other than `Constraint_Error`, function `Value` returns -1. If you remove the function's handler for `Constraint_Error` (lines 7 and 8) then the other "anonymous" handler will catch the exception and -1 will be returned instead of zero.

There are additional capabilities for exceptions, but for now you have a good basic understanding of how exceptions work, especially their dynamic nature at run-time.

5.2 Understanding Dynamic Checks versus Formal Proof

So far, we have discussed language-defined checks inserted by the compiler for verification at run-time, leading to exceptions being raised. We saw that these dynamic checks verified semantic conditions ensuring proper execution, such as preventing writing past the end of a buffer, or exceeding an application-specific integer range constraint, and so on. These checks are defined by the language because they apply generally and can be expressed in language-defined terms.

Developers can also define dynamic checks. These checks specify component-specific or application-specific conditions, expressed in terms defined by the component or application. We will refer to these checks as "user-defined" for convenience. (Be sure you understand that we are not talking about user-defined *exceptions* here.)

Like the language-defined checks, user-defined checks must be true at run-time. All checks consist of Boolean conditions, which is why we can refer to them as assertions: their conditions are asserted to be true by the compiler or developer.

Assertions come in several forms, some relatively low-level, such as a simple pragma `Assert`, and some high-level, such as type invariants and contracts. These forms will be presented in detail in a later section, but we will illustrate some of them here.

User-defined checks can be enabled at run-time in GNAT with the `-gnata` switch, as well as with pragma `Assertion_Policy`. The switch enables all forms of these assertions, whereas the pragma can be used to control specific forms. The switch is typically used but there are reasonable use-cases in which some user-defined checks are enabled, and others, although defined, are disabled.

By default in GNAT, language-defined checks are enabled but user-defined checks are disabled. Here's an example of a simple program employing a low-level assertion. We can use it to show the effects of the switches, including the defaults:

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

procedure Main is
   X : Positive := 10;
begin
   X := X * 5;
   pragma Assert (X > 99);
   X := X - 99;
   Put_Line (Integer'Image (X));
end Main;
```

If we compiled this code we would get a warning about the assignment on line 8 after the pragma `Assert`, but not one about the `Assert` itself on line 7.

```
gprbuild -q -P main.gpr
main.adb:8:11: warning: value not in range of type "Standard.Positive"
main.adb:8:11: warning: "Constraint_Error" will be raised at run time
```

No code is generated for the user-defined check expressed via pragma `Assert` but the language-defined check is emitted. In this case the range constraint on X excludes zero and negative numbers, but `X * 5 = 50`, `X - 99 = -49`. As a result, the check for the last assignment would fail, raising `Constraint_Error` when the program runs. These results are the expected behavior for the default switch settings.

But now let's enable user-defined checks and build it. Different compiler output will appear.

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

procedure Main is
   X : Positive := 10;
```

(continues on next page)

(continued from previous page)

```
begin
   X := X * 5;
   pragma Assert (X > 99);
   X := X - 99;
   Put_Line (Integer'Image (X));
end Main;
```

Now we also get the compiler warning about the pragma `Assert` condition. When run, the failure of pragma `Assert` on line 7 raises the exception `Ada.Assertions.Assertion_Error`. According to the expression in the assertion, X is expected (incorrectly) to be above 99 after the multiplication. (The exception name in the error message, SYSTEM.ASSERTIONS.ASSERT_FAILURE, is a GNAT-specific alias for `Ada.Assertions.Assertion_Error`.)

It's interesting to see in the output that the compiler can detect some violations at compile-time:

```
main.adb:7:19: warning: assertion will fail at run time
main.adb:7:21: warning: condition can only be True if invalid values present
main.adb:8:11: warning: value not in range of type "Standard.Positive"
```

Generally speaking, a complete analysis is beyond the scope of compilers and they may not find all errors prior to execution, even those we might detect ourselves by inspection. More errors can be found by tools dedicated to that purpose, known as static analyzers. But even an automated static analysis tool cannot guarantee it will find all potential problems.

A much more powerful alternative is formal proof, a form of static analysis that can (when possible) give strong guarantees about the checks, for all possible conditions and all possible inputs. Proof can be applied to both language-defined and user-defined checks.

Be sure you understand that formal proof, as a form of static analysis, verifies conditions prior to execution, even prior to compilation. That earliness provides significant cost benefits. Removing bugs earlier is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after deployment. Preventing bug introduction into the deployed system is the least expensive approach of all. Furthermore, cost savings during the initial development will be possible as well, for reasons specific to proof. We will revisit this topic later in this section.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the **gnatprove** verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof. As a disclaimer, this course is not aimed at providing a full introduction to proof and the SPARK language, but rather to present in a few examples what it is about and what it can do for us.

As it turns out, our procedure `Main` is already SPARK compliant so we can start verifying it.

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

procedure Main is
   X : Positive := 10;
begin
   X := X * 5;
   pragma Assert (X > 99);
   X := X - 99;
   Put_Line (Integer'Image (X));
end Main;
```

The "Prove" button invokes **gnatprove** on `main.adb`. You can ignore the parameters to the invocation. For the purpose of this demonstration, the interesting output is this message:

```
main.adb:7:19: medium: assertion might fail, cannot prove X > 99 (e.g. when X = 50)
```

gnatprove can tell that the assertion `X > 99` may have a problem. There's indeed a bug here, and **gnatprove** even gives us the counterexample (when X is 50). As a result the code is not proven

and we know we have an error to correct.

Notice that the message says the assertion "might fail" even though clearly **gnatprove** has an example for when failure is certain. That wording is a reflection of the fact that SPARK gives strong guarantees when the assertions are proven to hold, but does not guarantee that flagged problems are indeed problems. In other words, **gnatprove** does not give false positives but false negatives are possible. The result is that if **gnatprove** does not indicate a problem for the code under analysis we can be sure there is no problem, but if **gnatprove** does indicate a problem the tool may be wrong.

5.3 Initialization and Correct Data Flow

An immediate benefit from having our code compatible with the SPARK subset is that we can ask **gnatprove** to verify initialization and correct data flow, as indicated by the absence of messages during SPARK "flow analysis." Flow analysis detects programming errors such as reading uninitialized data, problematic aliasing between formal parameters, and data races between concurrent tasks.

In addition, **gnatprove** checks unit specifications for the actual data read or written, and the flow of information from inputs to outputs. As you can imagine, this verification provides significant benefits, and it can be reached with comparatively low cost.

For example, the following illustrates an initialization failure:

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

procedure Main is
   B : Integer;
begin
   Increment (B);
   Put_Line (B'Image);
end Main;
```

```
procedure Increment (Value : in out Integer) is
begin
    Value := Value + 1;
end Increment;
```

Granted, `Increment` is a silly procedure as-is, but imagine it did useful things, and, as part of that, incremented the argument. **gnatprove** tells us that the caller has not assigned a value to the argument passed to `Increment`.

Consider this next routine, which contains a serious coding error. Flow analysis will find it for us.

```
with Ada.Numerics.Elementary_Functions;  use Ada.Numerics.Elementary_Functions;

procedure Compute_Offset (K : Float; Z : out Integer; Flag : out Boolean) is
   X : constant Float := Sin (K);
begin
   if X < 0.0 then
      Z := 0;
      Flag := True;
   elsif X > 0.0 then
      Z := 1;
      Flag := True;
   else
      Flag := False;
   end if;
end Compute_Offset;
```

gnatprove tells us that Z might not be initialized (assigned a value) in `Compute_Offset`, and indeed that is correct. Z is a mode `out` parameter so the routine should assign a value to it: Z is an output, after all. The fact that `Compute_Offset` does not do so is a significant and nasty bug. Why is it so nasty? In this case, formal parameter Z is of the scalar type `Integer`, and scalar parameters are always passed by copy in Ada and SPARK. That means that, when returning to the caller, an integer value is copied to the caller's argument passed to Z. But this procedure doesn't always assign the value to be copied back, and in that case an arbitrary value — whatever is on the stack — is copied to the caller's argument. The poor programmer must debug the code to find the problem, yet the effect could appear well downstream from the call to `Compute_Offset`. That's not only painful, it is expensive. Better to find the problem before we even compile the code.

5.4 Contract-Based Programming

So far, we've seen assertions in a routine's sequence of statements, either through implicit language-defined checks (is the index in the right range?) or explicit user-defined checks. These checks are already useful by themselves but they have an important limitation: the assertions are in the implementation, hidden from the callers of the routine. For example, a call's success or failure may depend upon certain input values but the caller doesn't have that information.

Generally speaking, Ada and SPARK put a lot of emphasis on strong, complete specifications for the sake of abstraction and analysis. Callers need not examine the implementations to determine whether the arguments passed to it are changed, for example. It is possible to go beyond that, however, to specify implementation constraints and functional requirements. We use contracts to do so.

At the language level, contracts are higher-level forms of assertions associated with specifications and declarations rather than sequences of statements. Like other assertions they can be activated or deactivated at run-time, and can be statically proven. We'll concentrate here on two kinds of contracts, both associated especially (but not exclusively) with procedures and functions:

- *Preconditions*, those Boolean conditions required to be true *prior* to a call of the corresponding subprogram
- *Postconditions*, those Boolean conditions required to be true *after* a call, as a result of the corresponding subprogram's execution

In particular, preconditions specify the initial conditions, if any, required for the called routine to correctly execute. Postconditions, on the other hand, specify what the called routine's execution must have done, at least, on normal completion. Therefore, preconditions are obligations on callers (referred to as "clients") and postconditions are obligations on implementers. By the same token, preconditions are guarantees to the implementers, and postconditions are guarantees to clients.

Contract-based programming, then, is the specification and rigorous enforcement of these obligations and guarantees. Enforcement is rigorous because it is not manual, but tool-based: dynamically at run-time with exceptions, or, with SPARK, statically, prior to build.

Preconditions are specified via the "Pre" aspect. Postconditions are specified via the "Post" aspect. Usually subprograms have separate declarations and these aspects appear with those declarations, even though they are *about* the bodies. Placement on the declarations allows the obligations and guarantees to be visible to all parties. For example:

```
function Mid (X, Y : Integer) return Integer with
   Pre  => X + Y /= 0,
   Post => Mid'Result > X;
```

The precondition on line 2 specifies that, for any given call, the sum of the values passed to parameters X and Y must not be zero. (Perhaps we're dividing by X + Y in the body.) The declaration also provides a guarantee about the function call's result, via the postcondition on line 3: for any given call, the value returned will be greater than the value passed to X.

Consider a client calling this function:

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

procedure Demo is
   A, B, C : Integer;
begin
   A := Mid (1, 2);
   B := Mid (1, -1);
   C := Mid (A, B);
   Put_Line (C'Image);
end Demo;
```

gnatprove indicates that the assignment to B (line 8) might fail because of the precondition, i.e., the sum of the inputs shouldn't be 0, yet `-1 + 1 = 0`. (We will address the other output message elsewhere.)

Let's change the argument passed to Y in the second call (line 8). Instead of -1 we will pass -2:

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

procedure Demo is
   A, B, C : Integer;
begin
   A := Mid (1, 2);
   B := Mid (1, -2);
   C := Mid (A, B);
   Put_Line (C'Image);
end Demo;
```

```
function Mid (X, Y : Integer) return Integer with
   Pre  => X + Y /= 0,
   Post => Mid'Result > X;
```

The second call will no longer be flagged for the precondition. In addition, **gnatprove** will know from the postcondition that A has to be greater than 1, as does B, because in both calls 1 was passed to X. Therefore, **gnatprove** can deduce that the precondition will hold for the third call `C := Mid (A, B);` because the sum of two numbers greater than 1 will never be zero.

Postconditions can also compare the state prior to a call with the state after a call, using the `'Old` attribute. For example:

```
procedure Increment (Value : in out Integer) with
  Pre  => Value < Integer'Last,
  Post => Value = Value'Old + 1;
```

```
procedure Increment (Value : in out Integer) is
begin
   Value := Value + 1;
end Increment;
```

The postcondition specifies that, on return, the argument passed to the parameter `Value` will be one greater than it was immediately prior to the call (`Value'Old`).

5.5 Replacing Defensive Code

One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations. For example, the *Push* operation for a stack type would need to ensure that the

given stack is not already full. The body of the routine would first check that, explicitly, and perhaps raise an exception or set a status code. With preconditions we can make the requirement explicit and **gnatprove** will verify that the requirement holds at all call sites.

This reduction has a number of advantages:

- The implementation is simpler, removing validation code that is often difficult to test, makes the code more complex and leads to behaviors that are difficult to define.
- The precondition documents the conditions under which it's correct to call the subprogram, moving from an implementer responsibility to mitigate invalid input to a user responsibility to fulfill the expected interface.
- Provides the means to verify that this interface is properly respected, through code review, dynamic checking at run-time, or formal static proof.

As an example, consider a procedure Read that returns a component value from an array. Both the `Data` and `Index` are objects visible to the procedure so they are not formal parameters.

```
package P is

   type List is array (Integer range <>) of Character;

   Data  : List (1 .. 100);
   Index : Integer := Data'First;

   procedure Read (V : out Character);

end P;
```

```
package body P is

   procedure Read (V : out Character) is
   begin
      if Index not in Data'Range then
         V := Character'First;
         return;
      end if;

      V := Data (Index);
      Index := Index + 1;
   end Read;
end P;
```

In addition to procedure Read we would also have a way to load the array components in the first place, but we can ignore that for the purpose of this discussion.

Procedure Read is responsible for reading an element of the array and then incrementing the index. What should it do in case of an invalid index? In this implementation there is defensive code that returns a value arbitrarily chosen. We could also redesign the code to return a status in this case, or — better — raise an exception.

An even more robust approach would be instead to ensure that this subprogram is only called when `Index` is within the indexing boundaries of `Data`. We can express that requirement with a precondition (line 9).

```
package P is

   type List is array (Integer range <>) of Character;

   Data : List (1 .. 100);
   Index : Integer := 1;
```

(continues on next page)

(continued from previous page)

```
   procedure Read (V : out Character)
     with Pre => Index in Data'Range;

end P;
```

```
package body P is

   procedure Read (V : out Character) is
   begin
      V := Data (Index);
      Index := Index + 1;
   end Read;

end P;
```

Now we don't need the defensive code in the procedure body. That's safe because SPARK will attempt to prove statically that the check will not fail at the point of each call.

Assuming that procedure Read is intended to be the only way to get values from the array, in a real application (where the principles of software engineering apply) we would take advantage of the compile-time visibility controls that packages offer. Specifically, we would move all the variables' declarations to the private part of the package, or even the package body, so that client code could not possibly access the array directly. Only procedure Read would remain visible to clients, thus remaining the only means of accessing the array. However, that change would entail others, and in this chapter we are only concerned with introducing the capabilities of SPARK. Therefore, we keep the examples as simple as possible.

5.6 Proving Absence of Run-Time Errors

Earlier we said that **gnatprove** will verify both language-defined and user-defined checks. Proving that the language-defined checks will not raise exceptions at run-time is known as proving "Absence of Run-Time Errors" or AoRTE for short. Successful proof of these checks is highly significant in itself.

One of the major resulting benefits is that we can deploy the final executable with checks disabled. That has obvious performance benefits, but it is also a safety issue. If we disable the checks we also disable the run-time library support for them, but in that case the language does not define what happens if indeed an exception is raised. Formally speaking, anything could happen. We must have good reason for thinking that exceptions cannot be raised.

This is such an important issue that proof of AoRTE can be used to comply with the objectives of certification standards in various high-integrity domains (for example, DO-178B/C in avionics, EN 50128 in railway, IEC 61508 in many safety-related industries, ECSS-Q-ST-80C in space, IEC 60880 in nuclear, IEC 62304 in medical, and ISO 26262 in automotive).

As a result, the quality of the program can be guaranteed to achieve higher levels of integrity than would be possible in other programming languages.

However, successful proof of AoRTE may require additional assertions, especially preconditions. We can see that with procedure `Increment`, the procedure that takes an Integer argument and increments it by one. But of course, if the incoming value of the argument is the largest possible positive value, the attempt to increment it would overflow, raising `Constraint_Error`. (As you have likely already concluded, `Constraint_Error` is the most common exception you will have to deal with.) We added a precondition to allow only the integer values up to, but not including, the largest positive value:

```
procedure Increment (Value : in out Integer) with
  Pre  => Value < Integer'Last,
  Post => Value = Value'Old + 1;
```

```
procedure Increment (Value : in out Integer) is
begin
   Value := Value + 1;
end Increment;
```

Prove it, then comment-out the precondition and try proving it again. Not only will **gnatprove** tell us what is wrong, it will suggest a solution as well.

Without the precondition the check it provides would have to be implemented as defensive code in the body. One or the other is critical here, but note that we should never need both.

5.7 Proving Abstract Properties

The postcondition on `Increment` expresses what is, in fact, a unit-level requirement. Successfully proving such requirements is another significant robustness and cost benefit. Together with the proofs for initialization and AoRTE, these proofs ensure program integrity, that is, the program executes within safe boundaries: the control flow of the program is correctly programmed and cannot be circumvented through run-time errors, and data cannot be corrupted.

We can go even further. We can use contracts to express arbitrary abstract properties when such exist. Safety and security properties, for instance, could be expressed as postconditions and then proven by **gnatprove**.

For example, imagine we have a procedure to move a train to a new position on the track, and we want to do so safely, without leading to a collision with another train. Procedure `Move`, therefore, takes two inputs: a train identifier specifying which train to move, and the intended new position. The procedure's output is a value indicating a motion command to be given to the train in order to go to that new position. If the train cannot go to that new position safely the output command is to stop the train. Otherwise the command is for the train to continue at an indicated speed:

```
type Move_Result is (Full_Speed, Slow_Down, Keep_Going, Stop);

procedure Move
   (Train        : in  Train_Id;
    New_Position : in  Train_Position;
    Result       : out Move_Result)
with
   Pre  => Valid_Id (Train) and
           Valid_Move (Trains (Train), New_Position) and
           At_Most_One_Train_Per_Track and
           Safe_Signaling,
   Post => At_Most_One_Train_Per_Track and
           Safe_Signaling;

function At_Most_One_Train_Per_Track return Boolean;

function Safe_Signaling return Boolean;
```

The preconditions specify that, given a safe initial state and a valid move, the result of the call will also be a safe state: there will be at most one train per track section and the track signaling system will not allow any unsafe movements.

5.8 Final Comments

Make sure you understand that **gnatprove** does not attempt to prove the program correct as a whole. It attempts to prove language-defined and user-defined assertions about parts of the program, especially individual routines and calls to those routines. Furthermore, **gnatprove** proves

the routines correct only to the extent that the user-defined assertions correctly and sufficiently describe and constrain the implementation of the corresponding routines.

Although we are not proving whole program correctness, as you will have seen — and done — we can prove properties than make our software far more robust and bug-free than is possible otherwise. But in addition, consider what proving the unit-level requirements for your procedures and functions would do for the cost of unit testing and system integration. The tests would pass the first time.

However, within the scope of what SPARK can do, not everything can be proven. In some cases that is because the software behavior is not amenable to expression as boolean conditions (for example, a mouse driver). In other cases the source code is beyond the capabilities of the analyzers that actually do the mathematical proof. In these cases the combination of proof and actual test is appropriate, and still less expensive that testing alone.

There is, of course, much more to be said about what can be done with SPARK and **`gnatprove`**. Those topics are reserved for the Introduction to SPARK[18] course.

[18] https://learn.adacore.com/courses/intro-to-spark/index.html

CHAPTER

SIX

C TO ADA TRANSLATION PATTERNS

6.1 Naming conventions and casing considerations

One question that may arise relatively soon when converting from C to Ada is the style of source code presentation. The Ada language doesn't impose any particular style and for many reasons, it may seem attractive to keep a C-like style — for example, camel casing — to the Ada program.

However, the code in the Ada language standard, most third-party code, and the libraries provided by GNAT follow a specific style for identifiers and reserved words. Using a different style for the rest of the program leads to inconsistencies, thereby decreasing readability and confusing automatic style checkers. For those reasons, it's usually advisable to adopt the Ada style — in which each identifier starts with an upper case letter, followed by lower case letters (or digits), with an underscore separating two "distinct" words within the identifier. Acronyms within identifiers are in upper case. For example, there is a language-defined package named `Ada.Text_IO`. Reserved words are all lower case.

Following this scheme doesn't preclude adding additional, project-specific rules.

6.2 Interfacing C and Ada

6.2.1 Manual Interfacing

Before even considering translating code from C to Ada, it's worthwhile to evaluate the possibility of keeping a portion of the C code intact, and only translating selected modules to Ada. This is a necessary evil when introducing Ada to an existing large C codebase, where re-writing the entire code upfront is not practical nor cost-effective.

Fortunately, Ada has a dedicated set of features for interfacing with other languages. The `Interfaces` package hierarchy and the pragmas `Convention`, `Import`, and `Export` allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

[C]

```c
#include <stdio.h>

struct my_struct {
    int A, B;
};

void call (struct my_struct *p) {
    printf ("%d", p->A);
}
```

To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C `struct my_struct` is represented, we can use the following record along with a `pragma Convention`. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

[Ada]

```
with Ada.Text_IO;  use Ada.Text_IO;
with Interfaces.C;

procedure Use_My_Struct is

   type my_struct is record
     A : Interfaces.C.int;
     B : Interfaces.C.int;
   end record;
   pragma Convention (C, my_struct);

   V : my_struct := (A => 1, B => 2);
begin
   Put_Line ("V = ("
             & Interfaces.C.int'Image (V.A)
             & Interfaces.C.int'Image (V.B)
             & ")");
end Use_My_Struct;
```

Describing a foreign subprogram call to Ada code is called *binding* and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a void function maps to an Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a `pragma Import`:

```
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

The `Import` pragma specifies that whenever `Call` is invoked by Ada code, it should invoke the `Call` function with the C calling convention.

And that's all that's necessary. Here's an example of a call to `Call`:

[Ada]

```
with Interfaces.C;

procedure Use_My_Struct is

   type my_struct is record
     A : Interfaces.C.int;
     B : Interfaces.C.int;
   end record;
   pragma Convention (C, my_struct);

   procedure Call (V : my_struct);
   pragma Import (C, Call, "call"); -- Third argument optional

   V : my_struct := (A => 1, B => 2);
begin
   Call (V);
end Use_My_Struct;
```

6.2.2 Building and Debugging mixed language code

The easiest way to build an application using mixed C / Ada code is to create a simple project file for **gprbuild** and specify C as an additional language. By default, when using **gprbuild** we only compile Ada source files. To compile C code files as well, we use the `Languages` attribute and specify `c` as an option, as in the following example of a project file named *default.gpr*:

```
project Default is

   for Languages use ("ada", "c");
   for Main use ("main.adb");

end Default;
```

Then, we use this project file to build the application by simply calling `gprbuild`. Alternatively, we can specify the project file on the command-line with the `-P` option — for example, `gprbuild -P default.gpr`. In both cases, `gprbuild` compiles all C source-code file found in the directory and links the corresponding object files to build the executable.

In order to include debug information, you can use `gprbuild -cargs -g`. This option adds debug information based on both C and Ada code to the executable. Alternatively, you can specify a `Builder` package in the project file and include global compilation switches for each language using the `Global_Compilation_Switches` attribute. For example:

```
project Default is

   for Languages use ("ada", "c");
   for Main use ("main.adb");

   package Builder is
      for Global_Compilation_Switches ("Ada") use ("-g");
      for Global_Compilation_Switches ("C") use ("-g");
   end Builder;

end Default;
```

In this case, you can simply run `gprbuild -P default.gpr` to build the executable.

To debug the executable, you can use programs such as **gdb** or **ddd**, which are suitable for debugging both C and Ada source-code. If you prefer a complete IDE, you may want to look into **GNAT Studio**, which supports building and debugging an application within a single environment, and remotely running applications loaded to various embedded devices. You can find more information about **gprbuild** and **GNAT Studio** in the Introduction to GNAT Toolchain[19] course.

6.2.3 Automatic interfacing

It may be useful to start interfacing Ada and C by using automatic binding generators. These can be done either by invoking **gcc** `-fdump-ada-spec` option (to generate an Ada binding to a C header file) or `-gnatceg` option (to generate a C binding to an Ada specification file). For example:

```
gcc -c -fdump-ada-spec my_header.h
gcc -c -gnatceg spec.ads
```

The level of interfacing is very low level and typically requires either massaging (changing the generated files) or wrapping (calling the generated files from a higher level interface). For example, numbers bound from C to Ada are only standard numbers where user-defined types may be desirable. C uses a lot of by-pointer parameters which may be better replaced by other parameter modes, etc.

[19] https://learn.adacore.com/courses/GNAT_Toolchain_Intro/index.html

However, the automatic binding generator helps having a starting point which ensures compatibility of the Ada and the C code.

6.2.4 Using Arrays in C interfaces

It is relatively straightforward to pass an array from Ada to C. In particular, with the GNAT compiler, passing an array is equivalent to passing a pointer to its first element. Of course, as there's no notion of boundaries in C, the length of the array needs to be passed explicitly. For example:

[C]

```
void p (int * a, int length);
```

[Ada]

```
procedure Main is
   type Arr is array (Integer range <>) of Integer;

   procedure P (V : Arr; Length : Integer);
   pragma Import (C, P);

   X : Arr (5 .. 15);
begin
   P (X, X'Length);
end Main;
```

The other way around — that is, retrieving an array that has been creating on the C side — is more difficult. Because C doesn't explicitly carry boundaries, they need to be recreated in some way.

The first option is to actually create an Ada array without boundaries. This is the most flexible, but also the least safe option. It involves creating an array with indices over the full range of `Integer` without ever creating it from Ada, but instead retrieving it as an access from C. For example:

[C]

```
int * f ();
```

[Ada]

```
procedure Main is
   type Arr is array (Integer) of Integer;
   type Arr_A is access all Arr;

   function F return Arr_A;
   pragma Import (C, F);
begin
   null;
end Main;
```

Note that `Arr` is a constrained type (it doesn't have the `range <>` notation for indices). For that reason, as it would be for C, it's possible to iterate over the whole range of integer, beyond the memory actually allocated for the array.

A somewhat safer way is to overlay an Ada array over the C one. This requires having access to the length of the array. This time, let's consider two cases, one with an array and its size accessible through functions, another one on global variables. This time, as we're using an overlay, the function will be directly mapped to an Ada function returning an address:

[C]

```
int * f_arr (void);
int f_size (void);

int * g_arr;
int g_size;
```

[Ada]

```
with System;

package Fg is

   type Arr is array (Integer range <>) of Integer;

   function F_Arr return System.Address;
   pragma Import (C, F_Arr, "f_arr");

   function F_Size return Integer;
   pragma Import (C, F_Size, "f_size");

   F : Arr (0 .. F_Size - 1) with Address => F_Arr;

   G_Size : Integer;
   pragma Import (C, G_Size, "g_size");

   G_Arr : Arr (0 .. G_Size - 1);
   pragma Import (C, G_Arr, "g_arr");

end Fg;
```

```
with Fg;

procedure Main is
begin
   null;
end Main;
```

With all solutions though, importing an array from C is a relatively unsafe pattern, as there's only so much information on the array as there would be on the C side in the first place. These are good places for careful peer reviews.

6.2.5 By-value vs. by-reference types

When interfacing Ada and C, the rules of parameter passing are a bit different with regards to what's a reference and what's a copy. Scalar types and pointers are passed by value, whereas record and arrays are (almost) always passed by reference. However, there may be cases where the C interface also passes values and not pointers to objects. Here's a slightly modified version of a previous example to illustrate this point:

[C]

```
typedef struct my_struct {
    int A, B;
};

void call (struct my_struct p) {
    printf ("%d", p.A);
}
```

In Ada, a type can be modified so that parameters of this type can always be passed by copy.

[Ada]

```
with Interfaces.C;

procedure Main is
   type my_struct is record
      A : Interfaces.C.int;
      B : Interfaces.C.int;
   end record
     with Convention => C_Pass_By_Copy;

   procedure Call (V : my_struct);
   pragma Import (C, Call, "call");
begin
   null;
end Main;
```

Note that this cannot be done at the subprogram declaration level, so if there is a mix of by-copy and by-reference calls, two different types need to be used on the Ada side.

6.2.6 Naming and prefixes

Because of the absence of namespaces, any global name in C tends to be very long. And because of the absence of overloading, they can even encode type names in their type.

In Ada, the package is a namespace — two entities declared in two different packages are clearly identified and can always be specifically designated. The C names are usually a good indication of the names of the future packages and should be stripped — it is possible to use the full name if useful. For example, here's how the following declaration and call could be translated:

[C]

```
void registerInterface_Initialize (int size);
```

```
#include "reg_interface.h"

int main(int argc, const char * argv[])
{
    registerInterface_Initialize(15);
}
```

[Ada]

```
package Register_Interface is
   procedure Initialize (Size : Integer)
     with Import     => True,
          Convention => C,
          External_Name => "registerInterface_Initialize";

end Register_Interface;
```

```
with Register_Interface;

procedure Main is
begin
   Register_Interface.Initialize (15);
end Main;
```

Note that in the above example, a `use` clause on `Register_Interface` could allow us to omit the prefix.

6.2.7 Pointers

The first thing to ask when translating pointers from C to Ada is: are they needed in the first place? In Ada, pointers (or access types) should only be used with complex structures that cannot be allocated at run-time — think of a linked list or a graph for example. There are many other situations that would need a pointer in C, but do not in Ada, in particular:

- Arrays, even when dynamically allocated
- Results of functions
- Passing large structures as parameters
- Access to registers
- ... others

This is not to say that pointers aren't used in these cases but, more often than not, the pointer is hidden from the user and automatically handled by the code generated by the compiler; thus avoiding possible mistakes from being made. Generally speaking, when looking at C code, it's good practice to start by analyzing how many pointers are used and to translate as many as possible into *pointerless* Ada structures.

Here are a few examples of such patterns — additional examples can be found throughout this document.

Dynamically allocated arrays can be directly allocated on the stack:

[C]

```c
#include <stdlib.h>

int main() {
    int *a = malloc(sizeof(int) * 10);
}
```

[Ada]

```
procedure Main is
   type Arr is array (Integer range <>) of Integer;
   A : Arr (0 .. 9);
begin
   null;
end Main;
```

It's even possible to create a such an array within a structure, provided that the size of the array is known when instantiating this object, using a type discriminant:

[C]

```c
#include <stdlib.h>

typedef struct {
    int * a;
} S;

int main(int argc, const char * argv[])
{
    S v;

    v.a = malloc(sizeof(int) * 10);
}
```

[Ada]

```
procedure Main is
   type Arr is array (Integer range <>) of Integer;

   type S (Last : Integer) is record
      A : Arr (0 .. Last);
   end record;

   V : S (9);
begin
   null;
end Main;
```

With regards to parameter passing, usage mode (input / output) should be preferred to implementation mode (by copy or by reference). The Ada compiler will automatically pass a reference when needed. This works also for smaller objects, so that the compiler will copy in an out when needed. One of the advantages of this approach is that it clarifies the nature of the object: in particular, it differentiates between arrays and scalars. For example:

[C]

```
void p (int * a, int * b);
```

[Ada]

```
package Array_Types is
   type Arr is array (Integer range <>) of Integer;

   procedure P (A : in out Integer; B : in out Arr);
end Array_Types;
```

Most of the time, access to registers end up in some specific structures being mapped onto a specific location in memory. In Ada, this can be achieved through an `Address` clause associated to a variable, for example:

[C]

```
int main(int argc, const char * argv[])
{
    int * r = (int *)0xFFFF00A0;
}
```

[Ada]

```
with System;

procedure Test is
   R : Integer with Address => System'To_Address (16#FFFF00A0#);
begin
   null;
end Test;
```

These are some of the most common misuse of pointers in Ada. Previous sections of the document deal with specifically using access types if absolutely necessary.

6.2.8 Bitwise Operations

Bitwise operations such as masks and shifts in Ada should be relatively rarely needed, and, when translating C code, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other.

Consider the case of using a C primitive type as a container for single bit boolean flags. In C, this would be done through masks, e.g.:

[C]

```
#define FLAG_1 0b0001
#define FLAG_2 0b0010
#define FLAG_3 0b0100
#define FLAG_4 0b1000

int main(int argc, const char * argv[])
{
    int value = 0;

    value |= FLAG_2 | FLAG_4;
}
```

In Ada, the above can be represented through a Boolean array of enumerate values:

[Ada]

```
procedure Main is
   type Values is (Flag_1, Flag_2, Flag_3, Flag_4);
   type Value_Array is array (Values) of Boolean
     with Pack;

   Value : Value_Array :=
      (Flag_2 => True,
       Flag_4 => True,
       others => False);
begin
   null;
end Main;
```

Note the Pack directive for the array, which requests that the array takes as little space as possible.

It is also possible to map records on memory when additional control over the representation is needed or more complex data are used:

[C]

```
int main(int argc, const char * argv[])
{
    int value = 0;

    value = (2 << 1) | 1;
}
```

[Ada]

```
procedure Main is
   type Value_Rec is record
     V1 : Boolean;
     V2 : Integer range 0 .. 3;
   end record;

   for Value_Rec use record
      V1 at 0 range 0 .. 0;
      V2 at 0 range 1 .. 2;
   end record;

   Value : Value_Rec := (V1 => True, V2 => 2);
begin
```

(continues on next page)

(continued from previous page)

```
   null;
end Main;
```

The benefit of using Ada structure instead of bitwise operations is threefold:

- The code is simpler to read / write and less error-prone
- Individual fields are named
- The compiler can run consistency checks (for example, check that the value indeed fit in the expected size).

Note that, in cases where bitwise operators are needed, Ada provides modular types with `and`, `or` and `xor` operators. Further shift operators can also be provided upon request through a `pragma`. So the above could also be literally translated to:

[C]

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

procedure Main is
   type Value_Type is mod 2 ** 32;
   pragma Provide_Shift_Operators (Value_Type);

   Value : Value_Type;
begin
   Value := Shift_Left (2, 1) or 1;
   Put_Line ("Value = " & Value_Type'Image (Value));
end Main;
```

6.2.9 Mapping Structures to Bit-Fields

In the previous section, we've seen how to perform bitwise operations. In this section, we look at how to interpret a data type as a bit-field and perform low-level operations on it.

In general, you can create a bit-field from any arbitrary data type. First, we declare a bit-field type like this:

[Ada]

```
type Bit_Field is array (Natural range <>) of Boolean with Pack;
```

As we've seen previously, the `Pack` aspect declared at the end of the type declaration indicates that the compiler should optimize for size. We must use this aspect to be able to interpret data types as a bit-field.

Then, we can use the `Size` and the `Address` attributes of an object of any type to declare a bit-field for this object. We've discussed the `Size` attribute *earlier in this course* (page 72).

The `Address` attribute indicates the address in memory of that object. For example, assuming we've declare a variable `V`, we can declare an actual bit-field object by referring to the `Address` attribute of `V` and using it in the declaration of the bit-field, as shown here:

[Ada]

```
B : Bit_Field (0 .. V'Size - 1) with Address => V'Address;
```

Note that, in this declaration, we're using the `Address` attribute of `V` for the `Address` aspect of `B`.

This technique is called overlays for serialization. Now, any operation that we perform on `B` will have a direct impact on `V`, since both are using the same memory location.

The approach that we use in this section relies on the `Address` aspect. Another approach would be to use unchecked conversions, which we'll discuss in the *next section* (page 116).

We should add the `Volatile` aspect to the declaration to cover the case when both objects can still be changed independently — they need to be volatile, otherwise one change might be missed. This is the updated declaration:

[Ada]

```
B : Bit_Field (0 .. V'Size - 1) with Address => V'Address, Volatile;
```

Using the `Volatile` aspect is important at high level of optimizations. You can find further details about this aspect in the section about the *Volatile and Atomic aspects* (page 69).

Another important aspect that should be added is `Import`. When used in the context of object declarations, it'll avoid default initialization which could overwrite the existing content while creating the overlay — see an example in the admonition below. The declaration now becomes:

```
B : Bit_Field (0 .. V'Size - 1)
  with
    Address => V'Address, Import, Volatile;
```

Let's look at a simple example:

[Ada]

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

procedure Simple_Bitfield is
   type Bit_Field is array (Natural range <>) of Boolean with Pack;

   V : Integer := 0;
   B : Bit_Field (0 .. V'Size - 1)
     with Address => V'Address, Import, Volatile;
begin
   B (2) := True;
   Put_Line ("V = " & Integer'Image (V));
end Simple_Bitfield;
```

In this example, we first initialize `V` with zero. Then, we use the bit-field `B` and set the third element (`B (2)`) to `True`. This automatically sets bit #3 of `V` to 1. Therefore, as expected, the application displays the message `V = 4`, which corresponds to $2^2 = 4$.

Note that, in the declaration of the bit-field type above, we could also have used a positive range. For example:

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

B : Bit_Field (1 .. V'Size)
  with Address => V'Address, Import, Volatile;
```

The only difference in this case is that the first bit is `B (1)` instead of `B (0)`.

In C, we would rely on bit-shifting and masking to set that specific bit:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int v = 0;

    v = v | (1 << 2);
```

(continues on next page)

(continued from previous page)

```
    printf("v = %d\n", v);
}
```

Important

Ada has the concept of default initialization. For example, you may set the default value of record components:

[Ada]

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

procedure Main is

   type Rec is record
      X : Integer := 10;
      Y : Integer := 11;
   end record;

   R : Rec;
begin
   Put_Line ("R.X = " & Integer'Image (R.X));
   Put_Line ("R.Y = " & Integer'Image (R.Y));
end Main;
```

In the code above, we don't explicitly initialize the components of R, so they still have the default values 10 and 11, which are displayed by the application.

Likewise, the `Default_Value` aspect can be used to specify the default value in other kinds of type declarations. For example:

[Ada]

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

procedure Main is

   type Percentage is range 0 .. 100
     with Default_Value => 10;

   P : Percentage;
begin
   Put_Line ("P = " & Percentage'Image (P));
end Main;
```

When declaring an object whose type has a default value, the object will automatically be initialized with the default value. In the example above, P is automatically initialized with 10, which is the default value of the `Percentage` type.

Some types have an implicit default value. For example, access types have a default value of `null`.

As we've just seen, when declaring objects for types with associated default values, automatic initialization will happen. This can also happens when creating an overlay with the `Address` aspect. The default value is then used to overwrite the content at the memory location indicated by the address. However, in most situations, this isn't the behavior we expect, since overlays are usually created to analyze and manipulate existing values. Let's look at an example where this happens:

[Ada]

```
package P is
```

(continues on next page)

(continued from previous page)

```
   type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;

   type Byte_Field is array (Natural range <>) of Unsigned_8;

   procedure Display_Bytes_Increment (V : in out Integer);
end P;
```

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

package body P is

   procedure Display_Bytes_Increment (V : in out Integer) is
      BF  : Byte_Field (1 .. V'Size / 8)
        with Address => V'Address, Volatile;
   begin
      for B of BF loop
         Put_Line ("Byte = " & Unsigned_8'Image (B));
      end loop;
      Put_Line ("Now incrementing...");
      V := V + 1;
   end Display_Bytes_Increment;

end P;
```

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

with P; use P;

procedure Main is
   V : Integer := 10;
begin
   Put_Line ("V = " & Integer'Image (V));
   Display_Bytes_Increment (V);
   Put_Line ("V = " & Integer'Image (V));
end Main;
```

In this example, we expect `Display_Bytes_Increment` to display each byte of the `V` parameter and then increment it by one. Initially, `V` is set to 10, and the call to `Display_Bytes_Increment` should change it to 11. However, due to the default value associated to the `Unsigned_8` type — which is set to 0 — the value of `V` is overwritten in the declaration of `BF` (in `Display_Bytes_Increment`). Therefore, the value of `V` is 1 after the call to `Display_Bytes_Increment`. Of course, this is not the behavior that we originally intended.

Using the `Import` aspect solves this problem. This aspect tells the compiler to not apply default initialization in the declaration because the object is imported. Let's look at the corrected example:

[Ada]

```
package P is

   type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;

   type Byte_Field is array (Natural range <>) of Unsigned_8;

   procedure Display_Bytes_Increment (V : in out Integer);
end P;
```

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

package body P is
```

(continues on next page)

(continued from previous page)

```
   procedure Display_Bytes_Increment (V : in out Integer) is
      BF  : Byte_Field (1 .. V'Size / 8)
        with Address => V'Address, Import, Volatile;
   begin
      for B of BF loop
         Put_Line ("Byte = " & Unsigned_8'Image (B));
      end loop;
      Put_Line ("Now incrementing...");
      V := V + 1;
   end Display_Bytes_Increment;

end P;
```

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

with P; use P;

procedure Main is
   V : Integer := 10;
begin
   Put_Line ("V = " & Integer'Image (V));
   Display_Bytes_Increment (V);
   Put_Line ("V = " & Integer'Image (V));
end Main;
```

This unwanted side-effect of the initialization by the `Default_Value` aspect that we've just seen can also happen in these cases:

- when we set a default value for components of a record type declaration,
- when we use the `Default_Component_Value` aspect for array types, or
- when we set use the `Initialize_Scalars` pragma for a package.

Again, using the `Import` aspect when declaring the overlay eliminates this side-effect.

We can use this pattern for objects of more complex data types like arrays or records. For example:

[Ada]

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

procedure Int_Array_Bitfield is
   type Bit_Field is array (Natural range <>) of Boolean with Pack;

   A : array (1 .. 2) of Integer := (others => 0);
   B : Bit_Field (0 .. A'Size - 1)
     with Address => A'Address, Import, Volatile;
begin
   B (2) := True;
   for I in A'Range loop
      Put_Line ("A (" & Integer'Image (I)
                & ")= " & Integer'Image (A (I)));
   end loop;
end Int_Array_Bitfield;
```

In the Ada example above, we're using the bit-field to set bit #3 of the first element of the array (`A (1)`). We could set bit #4 of the second element by using the size of the data type (in this case, `Integer'Size`):

[Ada]

```
B (Integer'Size + 3) := True;
```

In C, we would select the specific array position and, again, rely on bit-shifting and masking to set that specific bit:

[C]

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int i;
    int a[2] = {0, 0};

    a[0] = a[0] | (1 << 2);

    for (i = 0; i < 2; i++)
    {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Since we can use this pattern for any arbitrary data type, this allows us to easily create a subprogram to serialize data types and, for example, transmit complex data structures as a bitstream. For example:

[Ada]

```
package Serializer is

   type Bit_Field is array (Natural range <>) of Boolean with Pack;

   procedure Transmit (B : Bit_Field);

end Serializer;
```

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

package body Serializer is

   procedure Transmit (B : Bit_Field) is

      procedure Show_Bit (V : Boolean) is
      begin
         case V is
            when False => Put ("0");
            when True  => Put ("1");
         end case;
      end Show_Bit;

   begin
      Put ("Bits: ");
      for E of B loop
         Show_Bit (E);
      end loop;
      New_Line;
   end Transmit;

end Serializer;
```

```
package My_Recs is
```

(continues on next page)

(continued from previous page)

```
   type Rec is record
      V : Integer;
      S : String (1 .. 3);
   end record;

end My_Recs;
```

```
with Serializer;  use Serializer;
with My_Recs;     use My_Recs;

procedure Main is
   R : Rec := (5, "abc");
   B : Bit_Field (0 .. R'Size - 1)
     with Address => R'Address, Import, Volatile;
begin
   Transmit (B);
end Main;
```

In this example, the `Transmit` procedure from `Serializer` package displays the individual bits of a bit-field. We could have used this strategy to actually transmit the information as a bitstream. In the main application, we call `Transmit` for the object R of record type `Rec`. Since `Transmit` has the bit-field type as a parameter, we can use it for any type, as long as we have a corresponding bit-field representation.

In C, we interpret the input pointer as an array of bytes, and then use shifting and masking to access the bits of that byte. Here, we use the `char` type because it has a size of one byte in most platforms.

[C]

```
typedef struct {
    int v;
    char s[4];
} rec;
```

```
void transmit (void *bits, int len);
```

```
#include "serializer.h"

#include <stdio.h>
#include <assert.h>

void transmit (void *bits, int len)
{
    int i, j;
    char *c = (char *)bits;

    assert(sizeof(char) == 1);

    printf("Bits: ");
    for (i = 0; i < len / (sizeof(char) * 8); i++)
    {
        for (j = 0; j < sizeof(char) * 8; j++)
        {
            printf("%d", c[i] >> j & 1);
        }
    }
    printf("\n");
}
```

```c
#include <stdio.h>

#include "my_recs.h"
#include "serializer.h"

int main(int argc, const char * argv[])
{
    rec r = {5, "abc"};

    transmit(&r, sizeof(r) * 8);
}
```

Similarly, we can write a subprogram that converts a bit-field — which may have been received as a bitstream — to a specific type. We can add a `To_Rec` subprogram to the `My_Recs` package to convert a bit-field to the `Rec` type. This can be used to convert a bitstream that we received into the actual data type representation.

As you know, we may write the `To_Rec` subprogram as a procedure or as a function. Since we need to use slightly different strategies for the implementation, the following example has both versions of `To_Rec`.

This is the updated code for the `My_Recs` package and the `Main` procedure:

[Ada]

```
package Serializer is

   type Bit_Field is array (Natural range <>) of Boolean with Pack;

   procedure Transmit (B : Bit_Field);

end Serializer;
```

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

package body Serializer is

   procedure Transmit (B : Bit_Field) is

      procedure Show_Bit (V : Boolean) is
      begin
         case V is
            when False => Put ("0");
            when True  => Put ("1");
         end case;
      end Show_Bit;

   begin
      Put ("Bits: ");
      for E of B loop
         Show_Bit (E);
      end loop;
      New_Line;
   end Transmit;

end Serializer;
```

```
with Serializer;  use Serializer;

package My_Recs is

   type Rec is record
```

(continues on next page)

(continued from previous page)

```
      V : Integer;
      S : String (1 .. 3);
   end record;

   procedure To_Rec (B :     Bit_Field;
                     R : out Rec);

   function To_Rec (B : Bit_Field) return Rec;

   procedure Display (R : Rec);

end My_Recs;
```

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

package body My_Recs is

   procedure To_Rec (B :     Bit_Field;
                     R : out Rec) is
      B_R : Rec
        with Address => B'Address, Import, Volatile;
   begin
      --  Assigning data from overlayed record B_R to output parameter R.
      R := B_R;
   end To_Rec;

   function To_Rec (B : Bit_Field) return Rec is
      R   : Rec;
      B_R : Rec
        with Address => B'Address, Import, Volatile;
   begin
      --  Assigning data from overlayed record B_R to local record R.
      R := B_R;

      return R;
   end To_Rec;

   procedure Display (R : Rec) is
   begin
      Put ("("  & Integer'Image (R.V) & ", "
           & (R.S) & ")");
   end Display;

end My_Recs;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Serializer;  use Serializer;
with My_Recs;     use My_Recs;

procedure Main is
   R1 : Rec := (5, "abc");
   R2 : Rec := (0, "zzz");

   B1 : Bit_Field (0 .. R1'Size - 1)
     with Address => R1'Address, Import, Volatile;
begin
   Put ("R2 = ");
   Display (R2);
   New_Line;
```

(continues on next page)

(continued from previous page)

```
   --  Getting Rec type using data from B1, which is a bit-field
   --  representation of R1.
   To_Rec (B1, R2);

   --  We could use the function version of To_Rec:
   --  R2 := To_Rec (B1);

   Put_Line ("New bitstream received!");
   Put ("R2 = ");
   Display (R2);
   New_Line;
end Main;
```

In both versions of `To_Rec`, we declare the record object `B_R` as an overlay of the input bit-field. In the procedure version of `To_Rec`, we then simply copy the data from `B_R` to the output parameter `R`. In the function version of `To_Rec`, however, we need to declare a local record object `R`, which we return after the assignment.

In C, we can interpret the input pointer as an array of bytes, and copy the individual bytes. For example:

[C]

```
typedef struct {
    int v;
    char s[3];
} rec;

void to_r (void *bits, int len, rec *r);

void display_r (rec *r);
```

```
#include "my_recs.h"

#include <stdio.h>
#include <assert.h>

void to_r (void *bits, int len, rec *r)
{
    int i;
    char *c1 = (char *)bits;
    char *c2 = (char *)r;

    assert(len == sizeof(rec) * 8);

    for (i = 0; i < len / (sizeof(char) * 8); i++)
    {
        c2[i] = c1[i];
    }
}

void display_r (rec *r)
{
    printf("{%d, %c%c%c}", r->v, r->s[0], r->s[1], r->s[2]);
}
```

```
#include <stdio.h>
#include "my_recs.h"

int main(int argc, const char * argv[])
{
```

(continues on next page)

(continued from previous page)

```
    rec r1 = {5, "abc"};
    rec r2 = {0, "zzz"};

    printf("r2 = ");
    display_r (&r2);
    printf("\n");

    to_r(&r1, sizeof(r1) * 8, &r2);

    printf("New bitstream received!\n");
    printf("r2 = ");
    display_r (&r2);
    printf("\n");
}
```

Here, `to_r` casts both pointer parameters to pointers to `char` to get a byte-aligned pointer. Then, it simply copies the data byte-by-byte.

6.2.9.1 Overlays vs. Unchecked Conversions

Unchecked conversions are another way of converting between unrelated data types. This conversion is done by instantiating the generic `Unchecked_Conversions` function for the types you want to convert. Let's look at a simple example:

[Ada]

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

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

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

   function As_Integer is new Ada.Unchecked_Conversion (Source => State,
                                                        Target => Integer);

   I : Integer;
begin
   I := As_Integer (State_2);
   Put_Line ("I = " & Integer'Image (I));
end Simple_Unchecked_Conversion;
```

In this example, `As_Integer` is an instantiation of `Unchecked_Conversion` to convert between the `State` enumeration and the `Integer` type. Note that, in order to ensure safe conversion, we're declaring `State` to have the same size as the `Integer` type we want to convert to.

This is the corresponding implementation using overlays:

[Ada]

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

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

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

(continues on next page)

(continued from previous page)

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

Let's look at another example of converting between different numeric formats. In this case, we want to convert between a 16-bit fixed-point and a 16-bit integer data type. This is how we can do it using `Unchecked_Conversion`:

[Ada]

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

procedure Fixed_Int_Unchecked_Conversion is
   Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
   Max_16   : constant := 2 ** 15;

   type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
     with Size => 16;
   type Int_16   is range -Max_16 .. Max_16 - 1
     with Size => 16;

   function As_Int_16 is new Ada.Unchecked_Conversion (Source => Fixed_16,
                                                       Target => Int_16);
   function As_Fixed_16 is new Ada.Unchecked_Conversion (Source => Int_16,
                                                         Target => Fixed_16);

   I : Int_16   := 0;
   F : Fixed_16 := 0.0;
begin
   F := Fixed_16'Last;
   I := As_Int_16 (F);

   Put_Line ("F = " & Fixed_16'Image (F));
   Put_Line ("I = " & Int_16'Image (I));
end Fixed_Int_Unchecked_Conversion;
```

Here, we instantiate `Unchecked_Conversion` for the `Int_16` and `Fixed_16` types, and we call the instantiated functions explicitly. In this case, we call `As_Int_16` to get the integer value corresponding to `Fixed_16'Last`.

This is how we can rewrite the implementation above using overlays:

[Ada]

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

procedure Fixed_Int_Overlay is
   Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
   Max_16   : constant := 2 ** 15;

   type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
     with Size => 16;
   type Int_16   is range -Max_16 .. Max_16 - 1
     with Size => 16;

   I : Int_16   := 0;
   F : Fixed_16
```

(continues on next page)

(continued from previous page)

```
     with Address => I'Address, Import, Volatile;
begin
   F := Fixed_16'Last;

   Put_Line ("F = " & Fixed_16'Image (F));
   Put_Line ("I = " & Int_16'Image (I));
end Fixed_Int_Overlay;
```

Here, the conversion to the integer value is implicit, so we don't need to call a conversion function.

Using `Unchecked_Conversion` has the advantage of making it clear that a conversion is happening, since the conversion is written explicitly in the code. With overlays, that conversion is automatic and therefore implicit. In that sense, using `Unchecked_Conversion` is a cleaner and safer approach. On the other hand, `Unchecked_Conversion` requires a copy, so it's less efficient than overlays, where no copy is performed — because one change in the source object is automatically reflected in the target object (and vice-versa). In the end, the choice between unchecked conversions and overlays depends on the level of performance that you want to achieve.

Also note that `Unchecked_Conversion` can only be instantiated for constrained types. In order to rewrite the examples using bit-fields that we've seen in the previous section, we cannot simply instantiate `Unchecked_Conversion` with the `Target` indicating the *unconstrained* bit-field, such as:

```
Ada.Unchecked_Conversion (Source => Integer,
                          Target => Bit_Field);
```

Instead, we have to declare a subtype for the specific range we're interested in. This is how we can rewrite one of the previous examples:

[Ada]

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

procedure Simple_Bitfield_Conversion is
   type Bit_Field is array (Natural range <>) of Boolean with Pack;

   V : Integer := 4;

   --  Declaring subtype that takes the size of V into account.
   --
   subtype Integer_Bit_Field is Bit_Field (0 .. V'Size - 1);

   --  NOTE: we could also use the Integer type in the declaration:
   --
   --    subtype Integer_Bit_Field is Bit_Field (0 .. Integer'Size - 1);
   --

   --  Using the Integer_Bit_Field subtype as the target
   function As_Bit_Field is new
     Ada.Unchecked_Conversion (Source => Integer,
                               Target => Integer_Bit_Field);

   B : Integer_Bit_Field;
begin
   B := As_Bit_Field (V);

   Put_Line ("V = " & Integer'Image (V));
end Simple_Bitfield_Conversion;
```

In this example, we first declare the subtype `Integer_Bit_Field` as a bit-field with a length that fits the `V` variable we want to convert to. Then, we can use that subtype in the instantiation of

`Unchecked_Conversion`.

CHAPTER

SEVEN

HANDLING VARIABILITY AND RE-USABILITY

7.1 Understanding static and dynamic variability

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. For example, the same application may need to be portable between two different architectures (ARM and x86), or two different platforms with different set of devices available. Maybe the same application is used for two different generations of the product, so it needs to account for absence or presence of new features, or it's used for different projects which may select different components or configurations. All these cases, and many others, require variability in the software in order to ensure its reusability.

In C, variability is usually achieved through macros and function pointers, the former being tied to static variability (variability in different builds) the latter to dynamic variability (variability within the same build decided at run-time).

Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. When Ada isn't enough, the GNAT compilation system also provides a layer of capabilities, in particular selection of alternate bodies.

If you're familiar with object-oriented programming (OOP) — supported in languages such as C++ and Java —, you might also be interested in knowing that OOP is supported by Ada and can be used to implement variability. This should, however, be used with care, as OOP brings its own set of problems, such as loss of efficiency — dispatching calls can't be inlined and require one level of indirection — or loss of analyzability — the target of a dispatching call isn't known at run time. As a rule of thumb, OOP should be considered only for cases of dynamic variability, where several versions of the same object need to exist concurrently in the same application.

7.2 Handling variability & reusability statically

7.2.1 Genericity

One usage of C macros involves the creation of functions that works regardless of the type they're being called upon. For example, a swap macro may look like:

[C]

```
#include <stdio.h>
#include <stdlib.h>

#define SWAP(t, a, b) ({\
                           t tmp = a; \
                           a = b; \
                           b = tmp; \
                       })
```

(continues on next page)

(continued from previous page)

```
int main()
{
    int a = 10;
    int b = 42;

    printf("a = %d, b = %d\n", a, b);

    SWAP (int, a, b);

    printf("a = %d, b = %d\n", a, b);
}
```

Ada offers a way to declare this kind of functions as a generic, that is, a function that is written after static arguments, such as a parameter:

[Ada]

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

procedure Main is

   generic
      type A_Type is private;
   procedure Swap (Left, Right : in out A_Type);

   procedure Swap (Left, Right : in out A_Type) is
      Temp : constant A_Type := Left;
   begin
      Left  := Right;
      Right := Temp;
   end Swap;

   procedure Swap_I is new Swap (Integer);

   A : Integer := 10;
   B : Integer := 42;

begin
   Put_Line ("A = "
             & Integer'Image (A)
             & ", B = "
             & Integer'Image (B));

   Swap_I (A, B);

   Put_Line ("A = "
             & Integer'Image (A)
             & ", B = "
             & Integer'Image (B));
end Main;
```

There are a few key differences between the C and the Ada version here. In C, the macro can be used directly and essentially get expanded by the preprocessor without any kind of checks. In Ada, the generic will first be checked for internal consistency. It then needs to be explicitly instantiated for a concrete type. From there, it's exactly as if there was an actual version of this Swap function, which is going to be called as any other function. All rules for parameter modes and control will apply to this instance.

In many respects, an Ada generic is a way to provide a safe specification and implementation of such macros, through both the validation of the generic itself and its usage.

Subprograms aren't the only entities that can me made generic. As a matter of fact, it's much more common to render an entire package generic. In this case the instantiation creates a new version of all the entities present in the generic, including global variables. For example:

[Ada]

```
generic
   type T is private;
package Gen is
   type C is tagged record
      V : T;
   end record;

   G : Integer;
end Gen;
```

The above can be instantiated and used the following way:

```
with Gen;

procedure Main is
   package I1 is new Gen (Integer);
   package I2 is new Gen (Integer);
   subtype Str10 is String (1 .. 10);
   package I3 is new Gen (Str10);
begin
   I1.G := 0;
   I2.G := 1;
   I3.G := 2;
end Main;
```

Here, `I1.G`, `I2.G` and `I3.G` are three distinct variables.

So far, we've only looked at generics with one kind of parameter: a so-called private type. There's actually much more that can be described in this section, such as variables, subprograms or package instantiations with certain properties. For example, the following provides a sort algorithm for any kind of structurally compatible array type:

[Ada]

```
generic
   type Component is private;
   type Index is (<>);
   with function "<" (Left, Right : Component) return Boolean;
   type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

The declaration above states that we need a type (`Component`), a discrete type (`Index`), a comparison subprogram (`"<"`), and an array definition (`Array_Type`). Given these, it's possible to write an algorithm that can sort any `Array_Type`. Note the usage of the with reserved word in front of the function name: it exists to differentiate between the generic parameter and the beginning of the generic subprogram.

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

```
type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type e.g. String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<>); -- T is a discrete type
type T is range <>; -- T is an integer type
type T is digits <>; -- T is a floating point type
type T is access T2; -- T is an access type to T2
```

For a more complete list please reference the Generic Formal Types Appendix[20].

7.2.2 Simple derivation

Let's take a case where a codebase needs to handle small variations of a given device, or maybe different generations of a device, depending on the platform it's running on. In this example, we're assuming that each platform will lead to a different binary, so the code can statically resolve which set of services are available. However, we want an easy way to implement a new device based on a previous one, saying "this new device is the same as this previous device, with these new services and these changes in existing services".

We can implement such patterns using Ada's simple derivation — as opposed to tagged derivation, which is OOP-related and discussed in a later section.

Let's start from the following example:

[Ada]

```
package Drivers_1 is

   type Device_1 is null record;
   procedure Startup (Device : Device_1);
   procedure Send (Device : Device_1; Data : Integer);
   procedure Send_Fast (Device : Device_1; Data : Integer);
   procedure Receive (Device : Device_1; Data : out Integer);

end Drivers_1;
```

```
package body Drivers_1 is

   --  NOTE: unimplemented procedures: Startup, Send, Send_Fast
   --        mock-up implementation: Receive

   procedure Startup (Device : Device_1) is null;

   procedure Send (Device : Device_1; Data : Integer) is null;

   procedure Send_Fast (Device : Device_1; Data : Integer) is null;

   procedure Receive (Device : Device_1; Data : out Integer) is
   begin
      Data := 42;
   end Receive;

end Drivers_1;
```

In the above example, `Device_1` is an empty record type. It may also have some fields if required, or be a different type such as a scalar. Then the four procedures `Startup`, `Send`, `Send_Fast` and `Receive` are primitives of this type. A primitive is essentially a subprogram that has a parameter or return type directly referencing this type and declared in the same scope. At this stage, there's nothing special with this type: we're using it as we would use any other type. For example:

[Ada]

```
with Ada.Text_IO; use Ada.Text_IO;
with Drivers_1;   use Drivers_1;

procedure Main is
   D : Device_1;
```

(continues on next page)

[20] https://learn.adacore.com/courses/intro-to-ada/chapters/appendices.html#appendix-a-generic-formal-types

(continued from previous page)

```
   I : Integer;
begin
   Startup (D);
   Send_Fast (D, 999);
   Receive (D, I);
   Put_Line (Integer'Image (I));
end Main;
```

Let's now assume that we need to implement a new generation of device, `Device_2`. This new device works exactly like the first one, except for the startup code that has to be done differently. We can create a new type that operates exactly like the previous one, but modifies only the behavior of `Startup`:

[Ada]

```
with Drivers_1; use Drivers_1;

package Drivers_2 is

   type Device_2 is new Device_1;

   overriding
   procedure Startup (Device : Device_2);

end Drivers_2;
```

```
package body Drivers_2 is

   overriding
   procedure Startup (Device : Device_2) is null;

end Drivers_2;
```

Here, `Device_2` is derived from `Device_1`. It contains all the exact same properties and primitives, in particular, `Startup`, `Send`, `Send_Fast` and `Receive`. However, here, we decided to change the `Startup` function and to provide a different implementation. We override this function. The main subprogram doesn't change much, except for the fact that it now relies on a different type:

[Ada]

```
with Ada.Text_IO; use Ada.Text_IO;
with Drivers_2;   use Drivers_2;

procedure Main is
   D : Device_2;
   I : Integer;
begin
   Startup (D);
   Send_Fast (D, 999);
   Receive (D, I);
   Put_Line (Integer'Image (I));
end Main;
```

We can continue with this approach and introduce a new generation of devices. This new device doesn't implement the `Send_Fast` service so we want to remove it from the list of available services. Furthermore, for the purpose of our example, let's assume that the hardware team went back to the `Device_1` way of implementing `Startup`. We can write this new device the following way:

[Ada]

```
with Drivers_1; use Drivers_1;

package Drivers_3 is

   type Device_3 is new Device_1;

   overriding
   procedure Startup (Device : Device_3);

   procedure Send_Fast (Device : Device_3; Data : Integer)
   is abstract;

end Drivers_3;
```

```
package body Drivers_3 is

   overriding
   procedure Startup (Device : Device_3) is null;

end Drivers_3;
```

The `is abstract` definition makes illegal any call to a function, so calls to `Send_Fast` on `Device_3` will be flagged as being illegal. To then implement `Startup` of `Device_3` as being the same as the `Startup` of `Device_1`, we can convert the type in the implementation:

[Ada]

```
package body Drivers_3 is

   overriding
   procedure Startup (Device : Device_3) is
   begin
      Drivers_1.Startup (Device_1 (Device));
   end Startup;

end Drivers_3;
```

Our `Main` now looks like:

[Ada]

```
with Ada.Text_IO; use Ada.Text_IO;
with Drivers_3;   use Drivers_3;

procedure Main is
   D : Device_3;
   I : Integer;
begin
   Startup (D);
   Send_Fast (D, 999);
   Receive (D, I);
   Put_Line (Integer'Image (I));
end Main;
```

Here, the call to `Send_Fast` will get flagged by the compiler.

Note that the fact that the code of `Main` has to be changed for every implementation isn't necessarily satisfactory. We may want to go one step further, and isolate the selection of the device kind to be used for the whole application in one unique file. One way to do this is to use the same name for all types, and use a renaming to select which package to use. Here's a simplified example to illustrate that:

[Ada]

```
package Drivers_1 is

   type Transceiver is null record;
   procedure Send (Device : Transceiver; Data : Integer);
   procedure Receive (Device : Transceiver; Data : out Integer);

end Drivers_1;
```

```
package body Drivers_1 is

   procedure Send (Device : Transceiver; Data : Integer) is null;

   procedure Receive (Device : Transceiver; Data : out Integer) is
      pragma Unreferenced (Device);
   begin
      Data := 42;
   end Receive;

end Drivers_1;
```

```
with Drivers_1;

package Drivers_2 is

   type Transceiver is new Drivers_1.Transceiver;
   procedure Send (Device : Transceiver; Data : Integer);
   procedure Receive (Device : Transceiver; Data : out Integer);

end Drivers_2;
```

```
package body Drivers_2 is

   procedure Send (Device : Transceiver; Data : Integer) is null;

   procedure Receive (Device : Transceiver; Data : out Integer) is
      pragma Unreferenced (Device);
   begin
      Data := 42;
   end Receive;

end Drivers_2;
```

```
with Drivers_1;

package Drivers renames Drivers_1;
```

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

procedure Main is
   D : Transceiver;
   I : Integer;
begin
   Send (D, 999);
   Receive (D, I);
   Put_Line (Integer'Image (I));
end Main;
```

In the above example, the whole code can rely on `drivers.ads`, instead of relying on the specific driver. Here, `Drivers` is another name for `Driver_1`. In order to switch to `Driver_2`, the project only has to replace that one `drivers.ads` file.

In the following section, we'll go one step further and demonstrate that this selection can be done through a configuration switch selected at build time instead of a manual code modification.

7.2.3 Configuration pragma files

Configuration pragmas are a set of pragmas that modify the compilation of source-code files. You may use them to either relax or strengthen requirements. For example:

```
pragma Suppress (Overflow_Check);
```

In this example, we're suppressing the overflow check, thereby relaxing a requirement. Normally, the following program would raise a constraint error due to a failed overflow check:

[Ada]

```
package P is
   function Add_Max (A : Integer) return Integer;
end P;
```

```
package body P is
   function Add_Max (A : Integer) return Integer is
   begin
      return A + Integer'Last;
   end Add_Max;
end P;
```

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

procedure Main is
   I : Integer := Integer'Last;
begin
   I := Add_Max (I);
   Put_Line ("I = " & Integer'Image (I));
end Main;
```

When suppressing the overflow check, however, the program doesn't raise an exception, and the value that `Add_Max` returns is `-2`, which is a wraparound of the sum of the maximum integer values (`Integer'Last + Integer'Last`).

We could also strengthen requirements, as in this example:

```
pragma Restrictions (No_Floating_Point);
```

Here, the restriction forbids the use of floating-point types and objects. The following program would violate this restriction, so the compiler isn't able to compile the program when the restriction is used:

```
procedure Main is
   F : Float := 0.0;
   --  Declaration is not possible with No_Floating_Point restriction.
begin
   null;
end Main;
```

Restrictions are especially useful for high-integrity applications. In fact, the Ada Reference Manual has a separate section for them[21].

[21] http://www.ada-auth.org/standards/12rm/html/RM-H-4.html

When creating a project, it is practical to list all configuration pragmas in a separate file. This is called a configuration pragma file, and it usually has an *.adc* file extension. If you use **GPRbuild** for building Ada applications, you can specify the configuration pragma file in the corresponding project file. For example, here we indicate that `gnat.adc` is the configuration pragma file for our project:

```
project Default is

   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Main use ("main.adb");

   package Compiler is
      for Local_Configuration_Pragmas use "gnat.adc";
   end Compiler;

end Default;
```

7.2.4 Configuration packages

In C, preprocessing flags are used to create blocks of code that are only compiled under certain circumstances. For example, we could have a block that is only used for debugging:

[C]

```
#include <stdio.h>
#include <stdlib.h>

int func(int x)
{
    return x % 4;
}

int main()
{
    int a, b;

    a = 10;
    b = func(a);

#ifdef DEBUG
    printf("func(%d) => %d\n", a, b);
#endif
}
```

Here, the block indicated by the DEBUG flag is only included in the build if we define this preprocessing flag, which is what we expect for a debug version of the build. In the release version, however, we want to keep debug information out of the build, so we don't use this flag during the build process.

Ada doesn't define a preprocessor as part of the language. Some Ada toolchains — like the GNAT toolchain — do have a preprocessor that could create code similar to the one we've just seen. When programming in Ada, however, the recommendation is to use configuration packages to select code blocks that are meant to be included in the application.

When using a configuration package, the example above can be written as:

[Ada]

```
package Config is

```

(continues on next page)

(continued from previous page)

```
   Debug : constant Boolean := False;

end Config;
```

```
function Func (X : Integer) return Integer;
```

```
function Func (X : Integer) return Integer is
begin
    return X mod 4;
end Func;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Config;
with Func;

procedure Main is
   A, B : Integer;
begin
   A := 10;
   B := Func (A);

   if Config.Debug then
      Put_Line ("Func(" & Integer'Image (A) & ") => "
                & Integer'Image (B));
   end if;
end Main;
```

In this example, `Config` is a configuration package. The version of `Config` we're seeing here is the release version. The debug version of the `Config` package looks like this:

```
package Config is

   Debug : constant Boolean := True;

end Config;
```

The compiler makes sure to remove dead code. In the case of the release version, since `Config.Debug` is constant and set to `False`, the compiler is smart enough to remove the call to `Put_Line` from the build.

As you can see, both versions of `Config` are very similar to each other. The general idea is to create packages that declare the same constants, but using different values.

In C, we differentiate between the debug and release versions by selecting the appropriate preprocessing flags, but in Ada, we select the appropriate configuration package during the build process. Since the file name is usually the same (`config.ads` for the example above), we may want to store them in distinct directories. For the example above, we could have:

- `src/debug/config.ads` for the debug version, and
- `src/release/config.ads` for the release version.

Then, we simply select the appropriate configuration package for each version of the build by indicating the correct path to it. When using **`GPRbuild`**, we can select the appropriate directory where the `config.ads` file is located. We can use scenario variables in our project, which allow for creating different versions of a build. For example:

```
project Default is

   type Mode_Type is ("debug", "release");
```

(continues on next page)

(continued from previous page)

```
   Mode : Mode_Type := external ("mode", "debug");

   for Source_Dirs use ("src", "src/" & Mode);
   for Object_Dir use "obj";
   for Main use ("main.adb");

end Default;
```

In this example, we're defining a scenario type called `Mode_Type`. Then, we're declaring the scenario variable `Mode` and using it in the `Source_Dirs` declaration to complete the path to the subdirectory containing the `config.ads` file. The expression `"src/" & Mode` concatenates the user-specified mode to select the appropriate subdirectory.

We can then set the mode on the command-line. For example:

```
gprbuild -P default.gpr -Xmode=release
```

In addition to selecting code blocks for the build, we could also specify values that depend on the target build. For our example above, we may want to create two versions of the application, each one having a different version of a `MOD_VALUE` that is used in the implementation of `func()`. In C, we can achieve this by using preprocessing flags and defining the corresponding version in `APP_VERSION`. Then, depending on the value of `APP_VERSION`, we define the corresponding value of `MOD_VALUE`.

[C]

```
#ifndef APP_VERSION
#define APP_VERSION     1
#endif

#if APP_VERSION == 1
#define MOD_VALUE       4
#endif

#if APP_VERSION == 2
#define MOD_VALUE       5
#endif
```

```
#include <stdio.h>
#include <stdlib.h>

#include "defs.h"

int func(int x)
{
    return x % MOD_VALUE;
}

int main()
{
    int a, b;

    a = 10;
    b = func(a);
}
```

If not defined outside, the code above will compile version #1 of the application. We can change this by specifying a value for `APP_VERSION` during the build (e.g. as a Makefile switch).

For the Ada version of this code, we can create two configuration packages for each version of the application. For example:

[Ada]

```
--  ./src/app_1/app_defs.ads

package App_Defs is

   Mod_Value : constant Integer := 4;

end App_Defs;
```

```
function Func (X : Integer) return Integer;
```

```
with App_Defs;

function Func (X : Integer) return Integer is
begin
    return X mod App_Defs.Mod_Value;
end Func;
```

```
with Func;

procedure Main is
   A, B : Integer;
begin
   A := 10;
   B := Func (A);
end Main;
```

The code above shows the version #1 of the configuration package. The corresponding implementation for version #2 looks like this:

```
--  ./src/app_2/app_defs.ads

package App_Defs is

   Mod_Value : constant Integer := 5;

end App_Defs;
```

Again, we just need to select the appropriate configuration package for each version of the build, which we can easily do when using **GPRbuild**.

7.3 Handling variability & reusability dynamically

7.3.1 Records with discriminants

In basic terms, records with discriminants are records that include ”parameters” in their type definitions. This allows for adding more flexibility to the type definition. In the section about *pointers* (page 103), we've seen this example:

[Ada]

```
procedure Main is
   type Arr is array (Integer range <>) of Integer;

   type S (Last : Positive) is record
      A : Arr (0 .. Last);
   end record;
```

(continues on next page)

(continued from previous page)

```
   V : S (9);
begin
   null;
end Main;
```

Here, `Last` is the discriminant for type S. When declaring the variable V as `S (9)`, we specify the actual index of the last position of the array component A by setting the `Last` discriminant to 9.

We can create an equivalent implementation in C by declaring a `struct` with a pointer to an array:

[C]

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int       * a;
    const int   last;
} S;

S init_s (int last)
{
    S v = { malloc (sizeof(int) * last + 1), last };
    return v;
}

int main(int argc, const char * argv[])
{
    S v = init_s (9);
}
```

Here, we need to explicitly allocate the `a` array of the S struct via a call to `malloc()`, which allocates memory space on the heap. In the Ada version, in contrast, the array (`V.A`) is allocated on the stack and we don't need to explicitly allocate it.

Note that the information that we provide as the discriminant to the record type (in the Ada code) is constant, so we cannot assign a value to it. For example, we cannot write:

[Ada]

```
V.Last := 10;        --  COMPILATION ERROR!
```

In the C version, we declare the `last` field constant to get the same behavior.

[C]

```
v.last = 10;         //  COMPILATION ERROR!
```

Note that the information provided as discriminants is visible. In the example above, we could display `Last` by writing:

[Ada]

```
Put_Line ("Last : " & Integer'Image (V.Last));
```

Also note that, even if a type is private, we can still access the information of the discriminants if they are visible in the *public* part of the type declaration. Let's rewrite the example above:

[Ada]

```
package Array_Definition is
   type Arr is array (Integer range <>) of Integer;
```

(continues on next page)

(continued from previous page)

```
   type S (Last : Integer) is private;

private
   type S (Last : Integer) is record
      A : Arr (0 .. Last);
   end record;

end Array_Definition;
```

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

procedure Main is
   V : S (9);
begin
   Put_Line ("Last : " & Integer'Image (V.Last));
end Main;
```

Even though the S type is now private, we can still display `Last` because this discriminant is visible in the *non-private* part of package `Array_Definition`.

7.3.2 Variant records

In simple terms, a variant record — a *discriminated record* in Ada terminology — is a record with discriminants that allows for changing its structure. Basically, it's a record containing a `case`. This is the general structure:

[Ada]

```
type Var_Rec (V : F) is record

   case V is
     when Opt_1 => F1 : Type_1;
     when Opt_2 => F2 : Type_2;
   end case;

end record;
```

Let's look at this example:

[Ada]

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

procedure Main is

   type Float_Int (Use_Float : Boolean) is record
      case Use_Float is
        when True  => F : Float;
        when False => I : Integer;
      end case;
   end record;

   procedure Display (V : Float_Int) is
   begin
      if V.Use_Float then
         Put_Line ("Float value:   " & Float'Image (V.F));
      else
```

(continues on next page)

(continued from previous page)

```
         Put_Line ("Integer value: " & Integer'Image (V.I));
      end if;
   end Display;

   F : constant Float_Int := (Use_Float => True,  F => 10.0);
   I : constant Float_Int := (Use_Float => False, I => 9);

begin
   Display (F);
   Display (I);
end Main;
```

Here, we declare `F` containing a floating-point value, and `I` containing an integer value. In the `Display` procedure, we present the correct information to the user according to the `Use_Float` discriminant of the `Float_Int` type.

We can implement this example in C by using unions:

[C]

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int use_float;
    union {
        float f;
        int   i;
    };
} float_int;

float_int init_float (float f)
{
    float_int v;

    v.use_float = 1;
    v.f         = f;
    return v;
}

float_int init_int (int i)
{
    float_int v;

    v.use_float = 0;
    v.i         = i;
    return v;
}

void display (float_int v)
{
    if (v.use_float) {
        printf("Float value   : %f\n", v.f);
    }
    else {
        printf("Integer value : %d\n", v.i);
    }
}

int main(int argc, const char * argv[])
{
    float_int f = init_float (10.0);
```

(continues on next page)

(continued from previous page)

```
    float_int i = init_int (9);

    display (f);
    display (i);
}
```

Similar to the Ada code, we declare `f` containing a floating-point value, and `i` containing an integer value. One difference is that we use the `init_float()` and `init_int()` functions to initialize the `float_int` struct. These functions initialize the correct field of the union and set the `use_float` field accordingly.

7.3.2.1 Variant records and unions

There is, however, a difference in accessibility between variant records in Ada and unions in C. In C, we're allowed to access any field of the union regardless of the initialization:

[C]

```
float_int v = init_float (10.0);

printf("Integer value : %d\n", v.i);
```

This feature is useful to create overlays. In this specific example, however, the information displayed to the user doesn't make sense, since the union was initialized with a floating-point value (`v.f`) and, by accessing the integer field (`v.i`), we're displaying it as if it was an integer value.

In Ada, accessing the wrong component would raise an exception at run-time ("discriminant check failed"), since the component is checked before being accessed:

[Ada]

```
   V : constant Float_Int := (Use_Float => True,  F => 10.0);
begin
   Put_Line ("Integer value: " & Integer'Image (V.I));
   --                                            ^ Constraint_Error is raised!
```

Using this method prevents wrong information being used in other parts of the program.

To get the same behavior in Ada as we do in C, we need to explicitly use the `Unchecked_Union` aspect in the type declaration. This is the modified example:

[Ada]

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

procedure Main is

   type Float_Int_Union (Use_Float : Boolean) is record
      case Use_Float is
        when True  => F : Float;
        when False => I : Integer;
      end case;
   end record
     with Unchecked_Union;

   V : constant Float_Int_Union := (Use_Float => True,  F => 10.0);

begin
   Put_Line ("Integer value: " & Integer'Image (V.I));
end Main;
```

Now, we can display the integer component (`V.I`) even though we initialized the floating-point component (`V.F`). As expected, the information displayed by the test application in this case doesn't make sense.

Note that, when using the `Unchecked_Union` aspect in the declaration of a variant record, the reference discriminant is not available anymore, since it isn't stored as part of the record. Therefore, we cannot access the `Use_Float` discriminant as in the following code:

[Ada]

```
   V : constant Float_Int_Union := (Use_Float => True,  F => 10.0);
begin
   if V.Use_Float then       --  COMPILATION ERROR!
      --  Do something...
   end if;
```

We could, however, declare another record with discriminants and use the `Float_Int_Union` type for one of its components. For example:

[Ada]

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

procedure Main is

   type Float_Int_Union (Use_Float : Boolean) is record
      case Use_Float is
        when True  => F : Float;
        when False => I : Integer;
      end case;
   end record
     with Unchecked_Union;

   type Float_Int (Use_Float : Boolean) is record
      U : Float_Int_Union (Use_Float);
   end record;

   V : constant Float_Int := (Use_Float => True,
                              U         => (Use_Float => True,  F => 10.0));

begin
   Put_Line ("Using float:   " & Boolean'Image (V.Use_Float));
   Put_Line ("Integer value: " & Integer'Image (V.U.I));
end Main;
```

Unchecked unions are particularly useful in Ada when creating bindings for C code.

7.3.2.2 Optional components

We can also use variant records to specify optional components of a record. For example:

[Ada]

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

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

   type Extra_Info is (No, Yes);

   type S_Var (Last : Integer; Has_Extra_Info : Extra_Info) is record
      A : Arr (0 .. Last);
```

(continues on next page)

(continued from previous page)

```
      case Has_Extra_Info is
        when No   => null;
        when Yes  => B : Arr (0 .. Last);
      end case;
   end record;

   V1 : S_Var (Last => 9, Has_Extra_Info => Yes);
   V2 : S_Var (Last => 9, Has_Extra_Info => No);
begin
   Put_Line ("Size of V1 is: " & Integer'Image (V1'Size));
   Put_Line ("Size of V2 is: " & Integer'Image (V2'Size));
end Main;
```

Here, in the declaration of `S_Var`, we don't have any component in case `Has_Extra_Info` is false. The component is simply set to `null` in this case.

When running the example above, we see that the size of `V1` is greater than the size of `V2` due to the extra `B` component — which is only included when `Has_Extra_Info` is true.

7.3.2.3 Optional output information

We can use optional components to prevent subprograms from generating invalid information that could be misused by the caller. Consider the following example:

[C]

```
#include <stdio.h>
#include <stdlib.h>

float calculate (float  f1,
                 float  f2,
                 int   *success)
{
    if (f1 < f2) {
        *success = 1;
        return f2 - f1;
    }
    else {
        *success = 0;
        return 0.0;
    }
}

void display (float v,
              int   success)
{
    if (success) {
        printf("Value = %f\n", v);
    }
    else {
        printf("Calculation error!\n");
    }
}

int main(int argc, const char * argv[])
{
    float f;
    int success;
```

(continues on next page)

(continued from previous page)

```
    f = calculate (1.0, 0.5, &success);
    display (f, success);

    f = calculate (0.5, 1.0, &success);
    display (f, success);
}
```

In this code, we're using the output parameter `success` of the `calculate()` function to indicate whether the calculation was successful or not. This approach has a major problem: there's no way to prevent that the invalid value returned by `calculate()` in case of an error is misused in another computation. For example:

[C]

```
int main(int argc, const char * argv[])
{
    float f;
    int success;

    f = calculate (1.0, 0.5, &success);

    f = f * 0.25;   // Using f in another computation even though
                    // calculate() returned a dummy value due to error!
                    // We should have evaluated "success", but we didn't.
}
```

We cannot prevent access to the returned value or, at least, force the caller to evaluate `success` before using the returned value.

This is the corresponding code in Ada:

[Ada]

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

procedure Main is

   function Calculate (F1, F2  : Float;
                       Success : out Boolean) return Float is
   begin
      if F1 < F2 then
         Success := True;
         return F2 - F1;
      else
        Success := False;
        return 0.0;
      end if;
   end Calculate;

   procedure Display (V : Float; Success : Boolean) is
   begin
      if Success then
         Put_Line ("Value = " & Float'Image (V));
      else
         Put_Line ("Calculation error!");
      end if;
   end Display;

   F       : Float;
   Success : Boolean;
begin
   F := Calculate (1.0, 0.5, Success);
```

(continues on next page)

(continued from previous page)

```
   Display (F, Success);

   F := Calculate (0.5, 1.0, Success);
   Display (F, Success);
end Main;
```

The Ada code above suffers from the same drawbacks as the C code. Again, there's no way to prevent misuse of the invalid value returned by `Calculate` in case of errors.

However, in Ada, we can use variant records to make the component unavailable and therefore prevent misuse of this information. Let's rewrite the original example and *wrap* the returned value in a variant record:

[Ada]

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

procedure Main is

   type Opt_Float (Success : Boolean) is record
      case Success is
        when False => null;
        when True  => F : Float;
      end case;
   end record;

   function Calculate (F1, F2 : Float) return Opt_Float is
   begin
      if F1 < F2 then
         return (Success => True, F => F2 - F1);
      else
         return (Success => False);
      end if;
   end Calculate;

   procedure Display (V : Opt_Float) is
   begin
      if V.Success then
         Put_Line ("Value = " & Float'Image (V.F));
      else
         Put_Line ("Calculation error!");
      end if;
   end Display;

begin
   Display (Calculate (1.0, 0.5));
   Display (Calculate (0.5, 1.0));
end Main;
```

In this example, we can determine whether the calculation was successful or not by evaluating the `Success` component of the `Opt_Float`. If the calculation wasn't successful, we won't be able to access the F component of the `Opt_Float`. As mentioned before, trying to access the component in this case would raise an exception. Therefore, in case of errors, we can ensure that no information is misused after the call to `Calculate`.

7.3.3 Object orientation

In the *previous section* (page 132), we've seen that we can add variability to records by using discriminants. Another approach is to use *tagged* records, which are the base for object-oriented programming in Ada.

7.3.3.1 Type extension

A tagged record type is declared by adding the `tagged` keyword. For example:

[Ada]

```
procedure Main is

   type Rec is record
      V : Integer;
   end record;

   type Tagged_Rec is tagged record
      V : Integer;
   end record;

   R1 : Rec;
   R2 : Tagged_Rec;

   pragma Unreferenced (R1, R2);
begin
   R1 := (V => 0);
   R2 := (V => 0);
end Main;
```

In this simple example, there isn't much difference between the `Rec` and `Tagged_Rec` type. However, tagged types can be derived and extended. For example:

[Ada]

```
procedure Main is

   type Rec is record
      V : Integer;
   end record;

   --  We cannot declare this:
   --
   --  type Ext_Rec is new Rec with record
   --     V : Integer;
   --  end record;

   type Tagged_Rec is tagged record
      V : Integer;
   end record;

   --  But we can declare this:
   --
   type Ext_Tagged_Rec is new Tagged_Rec with record
      V2 : Integer;
   end record;

   R1 : Rec;
   R2 : Tagged_Rec;
   R3 : Ext_Tagged_Rec;

   pragma Unreferenced (R1, R2, R3);
begin
   R1 := (V => 0);
   R2 := (V => 0);
   R3 := (V => 0, V2 => 0);
end Main;
```

As indicated in the example, a type derived from an untagged type cannot have an extension. The

compiler indicates this error if you uncomment the declaration of the `Ext_Rec` type above. In contrast, we can extend a tagged type, as we did in the declaration of `Ext_Tagged_Rec`. In this case, `Ext_Tagged_Rec` has all the components of the `Tagged_Rec` type (V, in this case) plus the additional components from its own type declaration (V2, in this case).

7.3.3.2 Overriding subprograms

Previously, we've seen that subprograms can be overriden. For example, if we had implemented a `Reset` and a `Display` procedure for the `Rec` type that we declared above, these procedures would be available for an `Ext_Rec` type derived from `Rec`. Also, we could override these procedures for the `Ext_Rec` type. In Ada, we don't need object-oriented programming features to do that: simple (untagged) records can be used to derive types, inherit operations and override them. However, in applications where the actual subprogram to be called is determined dynamically at run-time, we need dispatching calls. In this case, we must use tagged types to implement this.

7.3.3.3 Comparing untagged and tagged types

Let's discuss the similarities and differences between untagged and tagged types based on this example:

[Ada]

```
package P is

   type Rec is record
      V : Integer;
   end record;

   procedure Display (R : Rec);
   procedure Reset (R : out Rec);

   type New_Rec is new Rec;

   overriding procedure Display (R : New_Rec);
   not overriding procedure New_Op (R : in out New_Rec);

   type Tagged_Rec is tagged record
      V : Integer;
   end record;

   procedure Display (R : Tagged_Rec);
   procedure Reset (R : out Tagged_Rec);

   type Ext_Tagged_Rec is new Tagged_Rec with record
      V2 : Integer;
   end record;

   overriding procedure Display (R : Ext_Tagged_Rec);
   overriding procedure Reset (R : out Ext_Tagged_Rec);
   not overriding procedure New_Op (R : in out Ext_Tagged_Rec);

end P;
```

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

package body P is

   procedure Display (R : Rec) is
```

(continues on next page)

(continued from previous page)

```
   begin
      Put_Line ("TYPE: REC");
      Put_Line ("Rec.V = " & Integer'Image (R.V));
      New_Line;
   end Display;

   procedure Reset (R : out Rec) is
   begin
      R.V := 0;
   end Reset;

   procedure Display (R : New_Rec) is
   begin
      Put_Line ("TYPE: NEW_REC");
      Put_Line ("New_Rec.V = " & Integer'Image (R.V));
      New_Line;
   end Display;

   procedure New_Op (R : in out New_Rec) is
   begin
      R.V := R.V + 1;
   end New_Op;

   procedure Display (R : Tagged_Rec) is
   begin
      --  Using External_Tag attribute to retrieve the tag as a string
      Put_Line ("TYPE: " & Tagged_Rec'External_Tag);
      Put_Line ("Tagged_Rec.V = " & Integer'Image (R.V));
      New_Line;
   end Display;

   procedure Reset (R : out Tagged_Rec) is
   begin
      R.V := 0;
   end Reset;

   procedure Display (R : Ext_Tagged_Rec) is
   begin
      --  Using External_Tag attribute to retrieve the tag as a string
      Put_Line ("TYPE: " & Ext_Tagged_Rec'External_Tag);
      Put_Line ("Ext_Tagged_Rec.V  = " & Integer'Image (R.V));
      Put_Line ("Ext_Tagged_Rec.V2 = " & Integer'Image (R.V2));
      New_Line;
   end Display;

   procedure Reset (R : out Ext_Tagged_Rec) is
   begin
      Tagged_Rec (R).Reset;
      R.V2 := 0;
   end Reset;

   procedure New_Op (R : in out Ext_Tagged_Rec) is
   begin
      R.V := R.V + 1;
   end New_Op;

end P;
```

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

(continues on next page)

(continued from previous page)

```
procedure Main is
   X_Rec              : Rec;
   X_New_Rec          : New_Rec;

   X_Tagged_Rec       : aliased Tagged_Rec;
   X_Ext_Tagged_Rec   : aliased Ext_Tagged_Rec;

   X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class
                         := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
begin
   --
   --  Reset all objects
   --
   Reset (X_Rec);
   Reset (X_New_Rec);
   X_Tagged_Rec.Reset;    --  we could write "Reset (X_Tagged_Rec)" as well
   X_Ext_Tagged_Rec.Reset;

   --
   --  Use new operations when available
   --
   New_Op (X_New_Rec);
   X_Ext_Tagged_Rec.New_Op;

   --
   --  Display all objects
   --
   Display (X_Rec);
   Display (X_New_Rec);
   X_Tagged_Rec.Display;  --  we could write "Display (X_Tagged_Rec)" as well
   X_Ext_Tagged_Rec.Display;

   --
   --  Resetting and display objects of Tagged_Rec'Class
   --
   Put_Line ("Operations on Tagged_Rec'Class");
   Put_Line ("------------------------------");
   for E of X_Tagged_Rec_Array loop
      E.Reset;
      E.Display;
   end loop;
end Main;
```

These are the similarities between untagged and tagged types:

- We can derive types and inherit operations in both cases.
  - Both `X_New_Rec` and `X_Ext_Tagged_Rec` inherit the `Display` and `Reset` procedures from their respective ancestors.
- We can override operations in both cases.
- We can implement new operations in both cases.
  - Both `X_New_Rec` and `X_Ext_Tagged_Rec` implement a procedure called `New_Op`, which is not available for their respective ancestors.

Now, let's look at the differences between untagged and tagged types:

- We can dispatch calls for a given type class.
  - This is what we do when we iterate over objects of the `Tagged_Rec` class — in the loop over `X_Tagged_Rec_Array` at the last part of the `Main` procedure.
- We can use the dot notation.

- We can write both `E.Reset` or `Reset (E)` forms: they're equivalent.

7.3.3.4 Dispatching calls

Let's look more closely at the dispatching calls implemented above. First, we declare the `X_Tagged_Rec_Array` array and initialize it with the access to objects of both parent and derived tagged types:

[Ada]

```
X_Tagged_Rec       : aliased Tagged_Rec;
X_Ext_Tagged_Rec   : aliased Ext_Tagged_Rec;

X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class
                       := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
```

Here, we use the `aliased` keyword to be able to get access to the objects (via the `'Access` attribute).

Then, we loop over this array and call the `Reset` and `Display` procedures:

[Ada]

```
for E of X_Tagged_Rec_Array loop
   E.Reset;
   E.Display;
end loop;
```

Since we're using dispatching calls, the actual procedure that is selected depends on the type of the object. For the first element (`X_Tagged_Rec_Array (1)`), this is `Tagged_Rec`, while for the second element (`X_Tagged_Rec_Array (2)`), this is `Ext_Tagged_Rec`.

Dispatching calls are only possible for a type class — for example, the `Tagged_Rec'Class`. When the type of an object is known at compile time, the calls won't dispatch at runtime. For example, the call to the `Reset` procedure of the `X_Ext_Tagged_Rec` object (`X_Ext_Tagged_Rec.Reset`) will always take the overriden `Reset` procedure of the `Ext_Tagged_Rec` type. Similarly, if we perform a view conversion by writing `Tagged_Rec (A_Ext_Tagged_Rec).Display`, we're instructing the compiler to interpret `A_Ext_Tagged_Rec` as an object of type `Tagged_Rec`, so that the compiler selects the `Display` procedure of the `Tagged_Rec` type.

7.3.3.5 Interfaces

Another useful feature of object-oriented programming is the use of interfaces. In this case, we can define abstract operations, and implement them in the derived tagged types. We declare an interface by simply writing `type T is interface`. For example:

[Ada]

```
type My_Interface is interface;

procedure Op (Obj : My_Interface) is abstract;

--  We cannot declare actual objects of an interface:
--
--  Obj : My_Interface;  --   ERROR!
```

All operations on an interface type are abstract, so we need to write `is abstract` in the signature — as we did in the declaration of `Op` above. Also, since interfaces are abstract types and don't have an actual implementation, we cannot declare objects for it.

We can derive tagged types from an interface and implement the actual operations of that interface:

[Ada]

```
type My_Derived is new My_Interface with null record;

procedure Op (Obj : My_Derived);
```

Note that we're not using the `tagged` keyword in the declaration because any type derived from an interface is automatically tagged.

Let's look at an example with an interface and two derived tagged types:

[Ada]

```
package P is

   type Display_Interface is interface;
   procedure Display (D : Display_Interface) is abstract;

   type Small_Display_Type is new Display_Interface with null record;
   procedure Display (D : Small_Display_Type);

   type Big_Display_Type is new Display_Interface with null record;
   procedure Display (D : Big_Display_Type);

end P;
```

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

package body P is

   procedure Display (D : Small_Display_Type) is
      pragma Unreferenced (D);
   begin
      Put_Line ("Using Small_Display_Type");
   end Display;

   procedure Display (D : Big_Display_Type) is
      pragma Unreferenced (D);
   begin
      Put_Line ("Using Big_Display_Type");
   end Display;

end P;
```

```
with P; use P;

procedure Main is
   D_Small : Small_Display_Type;
   D_Big   : Big_Display_Type;

   procedure Dispatching_Display (D : Display_Interface'Class) is
   begin
      D.Display;
   end Dispatching_Display;

begin
   Dispatching_Display (D_Small);
   Dispatching_Display (D_Big);
end Main;
```

In this example, we have an interface type `Display_Interface` and two tagged types that are derived from `Display_Interface`: `Small_Display_Type` and `Big_Display_Type`.

Both types (`Small_Display_Type` and `Big_Display_Type`) implement the interface by overriding the `Display` procedure. Then, in the inner procedure `Dispatching_Display` of the `Main` procedure, we perform a dispatching call depending on the actual type of `D`.

7.3.3.6 Deriving from multiple interfaces

We may derive a type from multiple interfaces by simply writing `type Derived_T is new T1 and T2 with null record`. For example:

[Ada]

```
package Transceivers is

   type Send_Interface is interface;

   procedure Send (Obj : in out Send_Interface) is abstract;

   type Receive_Interface is interface;

   procedure Receive (Obj : in out Receive_Interface) is abstract;

   type Transceiver is new Send_Interface and Receive_Interface
     with null record;

   procedure Send (D : in out Transceiver);
   procedure Receive (D : in out Transceiver);

end Transceivers;
```

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

package body Transceivers is

   procedure Send (D : in out Transceiver) is
      pragma Unreferenced (D);
   begin
      Put_Line ("Sending data...");
   end Send;

   procedure Receive (D : in out Transceiver) is
      pragma Unreferenced (D);
   begin
      Put_Line ("Receiving data...");
   end Receive;

end Transceivers;
```

```
with Transceivers; use Transceivers;

procedure Main is
   D : Transceiver;
begin
   D.Send;
   D.Receive;
end Main;
```

In this example, we're declaring two interfaces (`Send_Interface` and `Receive_Interface`) and the tagged type `Transceiver` that derives from both interfaces. Since we need to implement the

interfaces, we implement both Send and Receive for Transceiver.

7.3.3.7 Abstract tagged types

We may also declare abstract tagged types. Note that, because the type is abstract, we cannot use it to declare objects for it — this is the same as for interfaces. We can only use it to derive other types. Let's look at the abstract tagged type declared in the Abstract_Transceivers package:

[Ada]

```
with Transceivers; use Transceivers;

 package Abstract_Transceivers is

    type Abstract_Transceiver is abstract new Send_Interface and
      Receive_Interface with null record;

    procedure Send (D : in out Abstract_Transceiver);
    --  We don't implement Receive for Abstract_Transceiver!

 end Abstract_Transceivers;
```

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

 package body Abstract_Transceivers is

    procedure Send (D : in out Abstract_Transceiver) is
       pragma Unreferenced (D);
    begin
       Put_Line ("Sending data...");
    end Send;

 end Abstract_Transceivers;
```

```
with Abstract_Transceivers; use Abstract_Transceivers;

 procedure Main is
    D : Abstract_Transceiver;
 begin
    D.Send;
    D.Receive;
 end Main;
```

In this example, we declare the abstract tagged type Abstract_Transceiver. Here, we're only partially implementing the interfaces from which this type is derived: we're implementing Send, but we're skipping the implementation of Receive. Therefore, Receive is an abstract operation of Abstract_Transceiver. Since any tagged type that has abstract operations is abstract, we must indicate this by adding the abstract keyword in type declaration.

Also, when compiling this example, we get an error because we're trying to declare an object of Abstract_Transceiver (in the Main procedure), which is not possible. Naturally, if we derive another type from Abstract_Transceiver and implement Receive as well, then we can declare objects of this derived type. This is what we do in the Full_Transceivers below:

[Ada]

```
with Abstract_Transceivers; use Abstract_Transceivers;

package Full_Transceivers is

```

(continues on next page)

(continued from previous page)

```
   type Full_Transceiver is new Abstract_Transceiver with null record;
   procedure Receive (D : in out Full_Transceiver);

end Full_Transceivers;
```

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

package body Full_Transceivers is

   procedure Receive (D : in out Full_Transceiver) is
      pragma Unreferenced (D);
   begin
      Put_Line ("Receiving data...");
   end Receive;

end Full_Transceivers;
```

```
with Full_Transceivers; use Full_Transceivers;

procedure Main is
   D : Full_Transceiver;
begin
   D.Send;
   D.Receive;
end Main;
```

Here, we implement the `Receive` procedure for the `Full_Transceiver`. Therefore, the type doesn't have any abstract operation, so we can use it to declare objects.

7.3.3.8 From simple derivation to OOP

In the *section about simple derivation* (page 124), we've seen an example where the actual selection was done at *implementation* time by renaming one of the packages:

[Ada]

```
with Drivers_1;

package Drivers renames Drivers_1;
```

Although this approach is useful in many cases, there might be situations where we need to select the actual driver dynamically at runtime. Let's look at how we could rewrite that example using interfaces, tagged types and dispatching calls:

[Ada]

```
package Drivers_Base is

   type Transceiver is interface;

   procedure Send (Device : Transceiver; Data : Integer) is abstract;
   procedure Receive (Device : Transceiver; Data : out Integer) is abstract;
   procedure Display (Device : Transceiver) is abstract;

end Drivers_Base;
```

```
with Drivers_Base;
```

(continues on next page)

(continued from previous page)

```
package Drivers_1 is

   type Transceiver is new Drivers_Base.Transceiver with null record;

   procedure Send (Device : Transceiver; Data : Integer);
   procedure Receive (Device : Transceiver; Data : out Integer);
   procedure Display (Device : Transceiver);

end Drivers_1;
```

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

package body Drivers_1 is

   procedure Send (Device : Transceiver; Data : Integer) is null;

   procedure Receive (Device : Transceiver; Data : out Integer) is
      pragma Unreferenced (Device);
   begin
      Data := 42;
   end Receive;

   procedure Display (Device : Transceiver) is
      pragma Unreferenced (Device);
   begin
      Put_Line ("Using Drivers_1");
   end Display;

end Drivers_1;
```

```
with Drivers_Base;

package Drivers_2 is

   type Transceiver is new Drivers_Base.Transceiver with null record;

   procedure Send (Device : Transceiver; Data : Integer);
   procedure Receive (Device : Transceiver; Data : out Integer);
   procedure Display (Device : Transceiver);

end Drivers_2;
```

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

package body Drivers_2 is

   procedure Send (Device : Transceiver; Data : Integer) is null;

   procedure Receive (Device : Transceiver; Data : out Integer) is
      pragma Unreferenced (Device);
   begin
      Data := 7;
   end Receive;

   procedure Display (Device : Transceiver) is
      pragma Unreferenced (Device);
   begin
      Put_Line ("Using Drivers_2");
   end Display;
```

(continues on next page)

(continued from previous page)

```
end Drivers_2;
```

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

with Drivers_Base;
with Drivers_1;
with Drivers_2;

procedure Main is
   D1 : aliased Drivers_1.Transceiver;
   D2 : aliased Drivers_2.Transceiver;
   D  : access Drivers_Base.Transceiver'Class;

   I  : Integer;

   type Driver_Number is range 1 .. 2;

   procedure Select_Driver (N : Driver_Number) is
   begin
      if N = 1 then
         D := D1'Access;
      else
         D := D2'Access;
      end if;
      D.Display;
   end Select_Driver;

begin
   Select_Driver (1);
   D.Send (999);
   D.Receive (I);
   Put_Line (Integer'Image (I));

   Select_Driver (2);
   D.Send (999);
   D.Receive (I);
   Put_Line (Integer'Image (I));
end Main;
```

In this example, we declare the `Transceiver` interface in the `Drivers_Base` package. This interface is then used to derive the tagged types `Transceiver` from both `Drivers_1` and `Drivers_2` packages.

In the `Main` procedure, we use the access to `Transceiver'Class` — from the interface declared in the `Drivers_Base` package — to declare `D`. This object `D` contains the access to the actual driver loaded at any specific time. We select the driver at runtime in the inner `Select_Driver` procedure, which initializes `D` (with the access to the selected driver). Then, any operation on `D` triggers a dispatching call to the selected driver.

7.3.3.9 Further resources

In the appendices, we have a step-by-step *hands-on overview of object-oriented programming* (page 179) that discusses how to translate a simple system written in C to an equivalent system in Ada using object-oriented programming.

7.3.4 Pointer to subprograms

Pointers to subprograms allow us to dynamically select an appropriate subprogram at runtime. This selection might be triggered by an external event, or simply by the user. This can be useful when multiple versions of a routine exist, and the decision about which one to use cannot be made at compilation time.

This is an example on how to declare and use pointers to functions in C:

[C]

```
#include <stdio.h>
#include <stdlib.h>

void show_msg_v1 (char *msg)
{
    printf("Using version #1: %s\n", msg);
}

void show_msg_v2 (char *msg)
{
    printf("Using version #2:\n %s\n", msg);
}

int main()
{
    int selection = 1;
    void (*current_show_msg) (char *);

    switch (selection)
    {
        case 1:  current_show_msg = &show_msg_v1;    break;
        case 2:  current_show_msg = &show_msg_v2;    break;
        default: current_show_msg = NULL;            break;
    }

    if (current_show_msg != NULL)
    {
        current_show_msg ("Hello there!");
    }
    else
    {
        printf("ERROR: no version of show_msg() selected!\n");
    }
}
```

The example above contains two versions of the `show_msg()` function: `show_msg_v1()` and `show_msg_v2()`. The function is selected depending on the value of `selection`, which initializes the function pointer `current_show_msg`. If there's no corresponding value, `current_show_msg` is set to `null` — alternatively, we could have selected a default version of `show_msg()` function. By calling `current_show_msg ("Hello there!")`, we're calling the function that `current_show_msg` is pointing to.

This is the corresponding implementation in Ada:

[Ada]

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

procedure Show_Subprogram_Selection is

   procedure Show_Msg_V1 (Msg : String) is
   begin
```

(continues on next page)

(continued from previous page)

```
      Put_Line ("Using version #1: " & Msg);
   end Show_Msg_V1;

   procedure Show_Msg_V2 (Msg : String) is
   begin
      Put_Line ("Using version #2: ");
      Put_Line (Msg);
   end Show_Msg_V2;

   type Show_Msg_Proc is access procedure (Msg : String);

   Current_Show_Msg : Show_Msg_Proc;
   Selection        : Natural;

begin
   Selection := 1;

   case Selection is
      when 1      => Current_Show_Msg := Show_Msg_V1'Access;
      when 2      => Current_Show_Msg := Show_Msg_V2'Access;
      when others => Current_Show_Msg := null;
   end case;

   if Current_Show_Msg /= null then
      Current_Show_Msg ("Hello there!");
   else
      Put_Line ("ERROR: no version of Show_Msg selected!");
   end if;

end Show_Subprogram_Selection;
```

The structure of the code above is very similar to the one used in the C code. Again, we have two version of `Show_Msg`: `Show_Msg_V1` and `Show_Msg_V2`. We set `Current_Show_Msg` according to the value of `Selection`. Here, we use `'Access` to get access to the corresponding procedure. If no version of `Show_Msg` is available, we set `Current_Show_Msg` to `null`.

Pointers to subprograms are also typically used as callback functions. This approach is extensively used in systems that process events, for example. Here, we could have a two-layered system:

- A layer of the system (an event manager) triggers events depending on information from sensors.
  - For each event, callback functions can be registered.
  - The event manager calls registered callback functions when an event is triggered.
- Another layer of the system registers callback functions for specific events and decides what to do when those events are triggered.

This approach promotes information hiding and component decoupling because:

- the layer of the system responsible for managing events doesn't need to know what the callback function actually does, while
- the layer of the system that implements callback functions remains agnostic to implementation details of the event manager — for example, how events are implemented in the event manager.

Let's see an example in C where we have a `process_values()` function that calls a callback function (`process_one`) to process a list of values:

[C]

```
typedef int (*process_one_callback) (int);

void process_values (int                  *values,
                     int                   len,
                     process_one_callback  process_one);
```

```
#include "process_values.h"

#include <assert.h>
#include <stdio.h>

void process_values (int                  *values,
                     int                   len,
                     process_one_callback  process_one)
{
    int i;

    assert (process_one != NULL);

    for (i = 0; i < len; i++)
    {
        values[i] = process_one (values[i]);
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>

#include "process_values.h"

int proc_10 (int val)
{
    return val + 10;
}

# define LEN_VALUES     5

int main()
{

    int values[LEN_VALUES] = { 1, 2, 3, 4, 5 };
    int i;

    process_values (values, LEN_VALUES, &proc_10);

    for (i = 0; i < LEN_VALUES; i++)
    {
        printf("Value [%d] = %d\n", i, values[i]);
    }
}
```

As mentioned previously, `process_values()` doesn't have any knowledge about what `process_one()` does with the integer value it receives as a parameter. Also, we could replace `proc_10()` by another function without having to change the implementation of `process_values()`.

Note that `process_values()` calls an `assert()` for the function pointer to compare it against `null`. Here, instead of checking the validity of the function pointer, we're expecting the caller of `process_values()` to provide a valid pointer.

This is the corresponding implementation in Ada:

[Ada]

```
package Values_Processing is

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

   type Process_One_Callback is not null access
     function (Value : Integer) return Integer;

   procedure Process_Values (Values      : in out Integer_Array;
                             Process_One :        Process_One_Callback);

end Values_Processing;
```

```
package body Values_Processing is

   procedure Process_Values (Values      : in out Integer_Array;
                             Process_One :        Process_One_Callback) is
   begin
      for I in Values'Range loop
         Values (I) := Process_One (Values (I));
      end loop;
   end Process_Values;

end Values_Processing;
```

```
function Proc_10 (Value : Integer) return Integer;
```

```
function Proc_10 (Value : Integer) return Integer is
begin
   return Value + 10;
end Proc_10;
```

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

with Values_Processing; use Values_Processing;
with Proc_10;

procedure Show_Callback is
   Values : Integer_Array := (1, 2, 3, 4, 5);
begin
   Process_Values (Values, Proc_10'Access);

   for I in Values'Range loop
      Put_Line ("Value ["
                & Positive'Image (I)
                & "] = "
                & Integer'Image (Values (I)));
   end loop;
end Show_Callback;
```

Similar to the implementation in C, the `Process_Values` procedure receives the access to a callback routine, which is then called for each value of the `Values` array.

Note that the declaration of `Process_One_Callback` makes use of the `not null access` declaration. By using this approach, we ensure that any parameter of this type has a valid value, so we can always call the callback routine.

7.4 Design by components using dynamic libraries

In the previous sections, we have shown how to use packages to create separate components of a system. As we know, when designing a complex system, it is advisable to separate concerns into distinct units, so we can use Ada packages to represent each unit of a system. In this section, we go one step further and create separate dynamic libraries for each component, which we'll then link to the main application.

Let's suppose we have a main system (`Main_System`) and a component *A* (`Component_A`) that we want to use in the main system. For example:

[Ada]

```
--
--  File: component_a.ads
--
package Component_A is

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

   function Average (Data : Float_Array) return Float;

end Component_A;
```

```
--
--  File: component_a.adb
--
package body Component_A is

   function Average (Data : Float_Array) return Float is
      Total : Float := 0.0;
   begin
      for Value of Data loop
         Total := Total + Value;
      end loop;
      return Total / Float (Data'Length);
   end Average;

end Component_A;
```

```
--
--  File: main_system.adb
--
with Ada.Text_IO; use Ada.Text_IO;

with Component_A; use Component_A;

procedure Main_System is
   Values        : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
   Average_Value : Float;
begin
   Average_Value := Average (Values);
   Put_Line ("Average = " & Float'Image (Average_Value));
end Main_System;
```

Note that, in the source-code example above, we're indicating the name of each file. We'll now see how to organize those files in a structure that is suitable for the GNAT build system (**`GPRbuild`**).

In order to discuss how to create dynamic libraries, we need to dig into some details about the build system. With GNAT, we can use project files for **`GPRbuild`** to easily design dynamic libraries. Let's say we use the following directory structure for the code above:

```
|- component_a
|    | component_a.gpr
|    |- src
|    |    | component_a.adb
|    |    | component_a.ads
|- main_system
|    | main_system.gpr
|    |- src
|    |    | main_system.adb
```

Here, we have two directories: *component_a* and *main_system*. Each directory contains a project file (with the *.gpr* file extension) and a source-code directory (*src*).

In the source-code example above, we've seen the content of files `component_a.ads`, `component_a.adb` and `main_system.adb`. Now, let's discuss how to write the project file for `Component_A` (`component_a.gpr`), which will build the dynamic library for this component:

```
library project Component_A is

   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Create_Missing_Dirs use "True";
   for Library_Name use "component_a";
   for Library_Kind use "dynamic";
   for Library_Dir use "lib";

end Component_A;
```

The project is defined as a *library project* instead of *project*. This tells **`GPRbuild`** to build a library instead of an executable binary. We then specify the library name using the *Library_Name* attribute, which is required, so it must appear in a library project. The next two library-related attributes are optional, but important for our use-case. We use:

- *Library_Kind* to specify that we want to create a dynamic library — by default, this attribute is set to *static*;
- *Library_Dir* to specify the directory where the library is stored.

In the project file of our main system (`main_system.gpr`), we just need to reference the project of `Component_A` using a *with* clause and indicating the correct path to that project file:

```
with "../component_a/component_a.gpr";

project Main_System is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
    for Create_Missing_Dirs use "True";
    for Main use ("main_system.adb");
end Main_System;
```

`GPRbuild` takes care of selecting the correct settings to link the dynamic library created for `Component_A` with the main application (`Main_System`) and build an executable.

We can use the same strategy to create a `Component_B` and dynamically link to it in the `Main_System`. We just need to create the separate structure for this component — with the appropriate Ada packages and project file — and include it in the project file of the main system using a *with* clause:

```
with "../component_a/component_a.gpr";
with "../component_b/component_b.gpr";

...
```

Again, **GPRbuild** takes care of selecting the correct settings to link both dynamic libraries together with the main application.

You can find more details and special setting for library projects in the GPRbuild documentation[22].

In the GNAT toolchain

The GNAT toolchain includes a more advanced example focusing on how to load dynamic libraries at runtime. You can find it in the `share/examples/gnat/plugins` directory of the GNAT toolchain installation. As described in the README file from that directory, this example "comprises a main program which probes regularly for the existence of shared libraries in a known location. If such libraries are present, it uses them to implement features initially not present in the main program."

[22] https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/gnat_project_manager.html#library-projects

CHAPTER

EIGHT

PERFORMANCE CONSIDERATIONS

8.1 Overall expectations

All in all, there should not be significant performance differences between code written in Ada and code written in C, provided that they are semantically equivalent. Taking the current GNAT implementation and its GCC C counterpart for example, most of the code generation and optimization phases are shared between C and Ada — so there's not one compiler more efficient than the other. Furthermore, the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. They should be equivalent on average.

When comparing the performance of C and Ada code, differences might be observed. This usually comes from the fact that, while the two piece *appear* semantically equivalent, they happen to be actually quite different; C code semantics do not implicitly apply the same run-time checks that Ada does. This section will present common ways for improving Ada code performance.

8.2 Switches and optimizations

Clever use of compilation switches might optimize the performance of an application significantly. In this section, we'll briefly look into some of the switches available in the GNAT toolchain.

8.2.1 Optimizations levels

Optimization levels can be found in many compilers for multiple languages. On the lowest level, the GNAT compiler doesn't optimize the code at all, while at the higher levels, the compiler analyses the code and optimizes it by removing unnecessary operations and making the most use of the target processor's capabilities.

By being part of GCC, GNAT offers the same `-O_` switches as GCC:

Switch	Description
`-O0`	No optimization: the generated code is completely unoptimized. This is the default optimization level.
`-O1`	Moderate optimization.
`-O2`	Full optimization.
`-O3`	Same optimization level as for `-O2`. In addition, further optimization strategies, such as aggressive automatic inlining and vectorization.

Note that the higher the level, the longer the compilation time. For fast compilation during development phase, unless you're working on benchmarking algorithms, using `-O0` is probably a good idea.

In addition to the levels presented above, GNAT also has the `-Os` switch, which allows for optimizing code and data usage.

8.2.2 Inlining

As we've seen in the previous section, automatic inlining depends on the optimization level. The highest optimization level (`-O3`), for example, performs aggressive automatic inlining. This could mean that this level inlines too much rather than not enough. As a result, the cache may become an issue and the overall performance may be worse than the one we would achieve by compiling the same code with optimization level 2 (`-O2`). Therefore, the general recommendation is to not *just* select `-O3` for the optimized version of an application, but instead compare it the optimized version built with `-O2`.

In some cases, it's better to reduce the optimization level and perform manual inlining instead of automatic inlining. We do that by using the `Inline` aspect. Let's reuse an example from a previous chapter and inline the `Average` function:

[Ada]

```
package Float_Arrays is

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

   function Average (Data : Float_Array) return Float
     with Inline;

end Float_Arrays;
```

```
package body Float_Arrays is

   function Average (Data : Float_Array) return Float is
      Total : Float := 0.0;
   begin
      for Value of Data loop
         Total := Total + Value;
      end loop;
      return Total / Float (Data'Length);
   end Average;

end Float_Arrays;
```

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

with Float_Arrays; use Float_Arrays;

procedure Compute_Average is
   Values        : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
   Average_Value : Float;
begin
   Average_Value := Average (Values);
   Put_Line ("Average = " & Float'Image (Average_Value));
end Compute_Average;
```

When compiling this example, GNAT will inline `Average` in the `Compute_Average` procedure.

In order to effectively use this aspect, however, we need to set the optimization level to at least `-O1` and use the `-gnatn` switch, which instructs the compiler to take the `Inline` aspect into account.

Note, however, that the `Inline` aspect is just a *recommendation* to the compiler. Sometimes, the compiler might not be able to follow this recommendation, so it won't inline the subprogram. In this case, we get a compilation warning from GNAT.

These are some examples of situations where the compiler might not be able to inline a subprogram:

- when the code is too large,
- when it's too complicated — for example, when it involves exception handling —, or
- when it contains tasks, etc.

In addition to the `Inline` aspect, we also have the `Inline_Always` aspect. In contrast to the former aspect, however, the `Inline_Always` aspect isn't primarily related to performance. Instead, it should be used when the functionality would be incorrect if inlining was not performed by the compiler. Examples of this are procedures that insert Assembly instructions that only make sense when the procedure is inlined, such as memory barriers.

Similar to the `Inline` aspect, there might be situations where a subprogram has the `Inline_Always` aspect, but the compiler is unable to inline it. In this case, we get a compilation error from GNAT.

8.3 Checks and assertions

8.3.1 Checks

Ada provides many runtime checks to ensure that the implementation is working as expected. For example, when accessing an array, we would like to make sure that we're not accessing a memory position that is not allocated for that array. This is achieved by an index check.

Another example of runtime check is the verification of valid ranges. For example, when adding two integer numbers, we would like to ensure that the result is still in the valid range — that the value is neither too large nor too small. This is achieved by an range check. Likewise, arithmetic operations shouldn't overflow or underflow. This is achieved by an overflow check.

Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots. For example, checking the index of an array in a sorting algorithm may significantly decrease its performance. In those cases, suppressing the check may be an option. We can achieve this suppression by using `pragma Suppress (Index_Check)`. For example:

[Ada]

```
procedure Sort (A : in out Integer_Array) is
   pragma Suppress (Index_Check);
begin
   --  (implementation removed...)
   null;
end Sort;
```

In case of overflow checks, we can use `pragma Suppress (Overflow_Check)` to suppress them:

```
function Some_Computation (A, B : Int32) return Int32 is
   pragma Suppress (Overflow_Check);
begin
   --  (implementation removed...)
   null;
end Sort;
```

We can also deactivate overflow checks for integer types using the `-gnato` switch when compiling a source-code file with GNAT. In this case, overflow checks in the whole file are deactivated.

It is also possible to suppress all checks at once using `pragma Suppress (All_Checks)`. In addition, GNAT offers a compilation switch called `-gnatp`, which has the same effect on the whole file.

Note, however, that this kind of suppression is just a recommendation to the compiler. There's no guarantee that the compiler will actually suppress any of the checks because the compiler may not be able to do so — typically because the hardware happens to do it. For example, if the machine traps on any access via address zero, requesting the removal of null access value checks in the generated code won't prevent the checks from happening.

It is important to differentiate between required and redundant checks. Let's consider the following example in C:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 8, b = 0, res;

    res = a / b;

    // printing the result
    printf("res = %d\n", res);

    return 0;
}
```

Because C doesn't have language-defined checks, as soon as the application tries to divide a value by zero in `res = a / b`, it'll break — on Linux, for example, you may get the following error message by the operating system: `Floating point exception (core dumped)`. Therefore, we need to manually introduce a check for zero before this operation. For example:

[C]

```c
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int a = 8, b = 0, res;

    if (b != 0) {
        res = a / b;

        // printing the result
        printf("res = %d\n", res);
    }
    else
    {
        // printing error message
        printf("Error: cannot calculate value (division by zero)\n");
    }

    return 0;
}
```

This is the corresponding code in Ada:

[Ada]

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

procedure Show_Division_By_Zero is
   A   : Integer := 8;
   B   : Integer := 0;
   Res : Integer;
```

(continues on next page)

(continued from previous page)

```
begin
   Res := A / B;

   Put_Line ("Res = " & Integer'Image (Res));
end Show_Division_By_Zero;
```

Similar to the first version of the C code, we're not explicitly checking for a potential division by zero here. In Ada, however, this check is *automatically inserted* by the language itself. When running the application above, an exception is raised when the application tries to divide the value in A by zero. We could introduce exception handling in our example, so that we get the same message as we did in the second version of the C code:

[Ada]

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

procedure Show_Division_By_Zero is
   A   : Integer := 8;
   B   : Integer := 0;
   Res : Integer;
begin
   Res := A / B;

   Put_Line ("Res = " & Integer'Image (Res));
exception
   when Constraint_Error =>
      Put_Line ("Error: cannot calculate value (division by zero)");
   when others =>
      null;
end Show_Division_By_Zero;
```

This example demonstrates that the division check for `Res := A / B` is required and shouldn't be suppressed. In contrast, a check is redundant — and therefore not required — when we know that the condition that leads to a failure can never happen. In many cases, the compiler itself detects redundant checks and eliminates them (for higher optimization levels). Therefore, when improving the performance of your application, you should:

1. keep all checks active for most parts of the application;
2. identify the hot-spots of your application;
3. identify which checks haven't been eliminated by the optimizer on these hot-spots;
4. identify which of those checks are redundant;
5. only suppress those checks that are redundant, and keep the required ones.

8.3.2 Assertions

We've already discussed assertions in *this section of the SPARK chapter* (page 87). Assertions are user-defined checks that you can add to your code using the `pragma Assert`. For example:

[Ada]

```
function Some_Computation (A, B : Int32) return Int32 is
   Res : Int32;
begin
   --  (implementation removed...)

   pragma Assert (Res >= 0);
```

(continues on next page)

(continued from previous page)

```
   return Res;
end Sort;
```

Assertions that are specified with `pragma Assert` are not enabled by default. You can enable them by setting the assertion policy to *check* — using `pragma Assertion_Policy (Check)` — or by using the `-gnata` switch when compiling with GNAT.

Similar to the checks discussed previously, assertions can generate significant overhead when used at hot-spots. Restricting those assertions to development (e.g. debug version) and turning them off on the release version may be an option. In this case, formal proof — as discussed in the *SPARK chapter* (page 81) — can help you. By formally proving that assertions will never fail at run-time, you can safely deactivate them.

8.4 Dynamic vs. static structures

Ada generally speaking provides more ways than C or C++ to write simple dynamic structures, that is to say structures that have constraints computed after variables. For example, it's quite typical to have initial values in record types:

[Ada]

```
type R is record
   F : Some_Field := Call_To_Some_Function;
end record;
```

However, the consequences of the above is that any declaration of a instance of this type without an explicit value for V will issue a call to `Call_To_Some_Function`. More subtle issue may arise with elaboration. For example, it's possible to write:

```
package Some_Functions is

   function Some_Function_Call return Integer is (2);

   function Some_Other_Function_Call return Integer is (10);

end Some_Functions;
```

```
with Some_Functions; use Some_Functions;

package Values is
   A_Start : Integer := Some_Function_Call;
   A_End   : Integer := Some_Other_Function_Call;
end Values;
```

```
with Values; use Values;

package Arr_Def is
   type Arr is array (Integer range A_Start .. A_End) of Integer;
end Arr_Def;
```

It may indeed be appealing to be able to change the values of `A_Start` and `A_End` at startup so as to align a series of arrays dynamically. The consequence, however, is that these values will not be known statically, so any code that needs to access to boundaries of the array will need to read data from memory. While it's perfectly fine most of the time, there may be situations where performances are so critical that static values for array boundaries must be enforced.

Here's a last case which may also be surprising:

[Ada]

```
package Arr_Def is
   type Arr is array (Integer range <>) of Integer;

   type R (D1, D2 : Integer) is record
      F1 : Arr (1 .. D1);
      F2 : Arr (1 .. D2);
   end record;
end Arr_Def;
```

In the code above, R contains two arrays, F1 and F2, respectively constrained by the discriminant D1 and D2. The consequence is, however, that to access F2, the run-time needs to know how large F1 is, which is dynamically constrained when creating an instance. Therefore, accessing to F2 requires a computation involving D1 which is slower than, let's say, two pointers in an C array that would point to two different arrays.

Generally speaking, when values are used in data structures, it's useful to always consider where they're coming from, and if their value is static (computed by the compiler) or dynamic (only known at run-time). There's nothing fundamentally wrong with dynamically constrained types, unless they appear in performance-critical pieces of the application.

8.5 Pointers vs. data copies

In the section about *pointers* (page 103), we mentioned that the Ada compiler will automatically pass parameters by reference when needed. Let's look into what "when needed" means. The fundamental point to understand is that the parameter types determine how the parameters are passed in and/or out. The parameter modes do not control how parameters are passed.

Specifically, the language standards specifies that scalar types are always passed by value, and that some other types are always passed by reference. It would not make sense to make a copy of a task when passing it as a parameter, for example. So parameters that can be passed reasonably by value will be, and those that must be passed by reference will be. That's the safest approach.

But the language also specifies that when the parameter is an array type or a record type, and the record/array components are all by-value types, then the compiler decides: it can pass the parameter using either mechanism. The critical case is when such a parameter is large, e.g., a large matrix. We don't want the compiler to pass it by value because that would entail a large copy, and indeed the compiler will not do so. But if the array or record parameter is small, say the same size as an address, then it doesn't matter how it is passed and by copy is just as fast as by reference. That's why the language gives the choice to the compiler. Although the language does not mandate that large parameters be passed by reference, any reasonable compiler will do the right thing.

The modes do have an effect, but not in determining how the parameters are passed. Their effect, for parameters passed by value, is to determine how many times the value is copied. For `mode in` and `mode out` there is just one copy. For `mode in out` there will be two copies, one in each direction.

Therefore, unlike C, you don't have to use access types in Ada to get better performance when passing arrays or records to subprograms. The compiler will almost certainly do the right thing for you.

Let's look at this example:

[C]

```
#include <stdio.h>

struct Data {
    int prev, curr;
```

(continues on next page)

(continued from previous page)

```
};

void update(struct Data *d,
            int          v)
{
    d->prev = d->curr;
    d->curr = v;
}

void display(const struct Data *d)
{
    printf("Prev : %d\n", d->prev);
    printf("Curr : %d\n", d->curr);
}

int main(int argc, const char * argv[])
{
    struct Data D1 = { 0, 1 };

    update (&D1, 3);
    display (&D1);
}
```

In this C code example, we're using pointers to pass `D1` as a reference to `update` and `display`. In contrast, the equivalent code in Ada simply uses the parameter modes to specify the data flow directions. The mechanisms used to pass the values do not appear in the source code.

[Ada]

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

procedure Update_Record is

   type Data is record
      Prev : Integer;
      Curr : Integer;
   end record;

   procedure Update (D : in out Data;
                     V :        Integer) is
   begin
      D.Prev := D.Curr;
      D.Curr := V;
   end Update;

   procedure Display (D : Data) is
   begin
      Put_Line ("Prev: " & Integer'Image (D.Prev));
      Put_Line ("Curr: " & Integer'Image (D.Curr));
   end Display;

   D1 : Data := (0, 1);

begin
   Update (D1, 3);
   Display (D1);
end Update_Record;
```

In the calls to `Update` and `Display`, `D1` is always be passed by reference. Because no extra copy takes place, we get a performance that is equivalent to the C version. If we had used arrays in the example above, `D1` would have been passed by reference as well:

[Ada]

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

procedure Update_Array is

   type Data_State is (Prev, Curr);
   type Data is array (Data_State) of Integer;

   procedure Update (D : in out Data;
                     V :        Integer) is
   begin
      D (Prev) := D (Curr);
      D (Curr) := V;
   end Update;

   procedure Display (D : Data) is
   begin
      Put_Line ("Prev: " & Integer'Image (D (Prev)));
      Put_Line ("Curr: " & Integer'Image (D (Curr)));
   end Display;

   D1 : Data := (0, 1);

begin
   Update (D1, 3);
   Display (D1);
end Update_Array;
```

Again, no extra copy is performed in the calls to `Update` and `Display`, which gives us optimal performance when dealing with arrays and avoids the need to use access types to optimize the code.

8.5.1 Function returns

Previously, we've discussed the cost of passing complex records as arguments to subprograms. We've seen that we don't have to use explicit access type parameters to get better performance in Ada. In this section, we'll briefly discuss the cost of function returns.

In general, we can use either procedures or functions to initialize a data structure. Let's look at this example in C:

[C]

```
#include <stdio.h>

struct Data {
    int prev, curr;
};

void init_data(struct Data *d)
{
    d->prev = 0;
    d->curr = 1;
}

struct Data get_init_data()
{
    struct Data d  = { 0, 1 };

    return d;
```

(continues on next page)

(continued from previous page)

```
}

int main(int argc, const char * argv[])
{
    struct Data D1;

    D1 = get_init_data();

    init_data(&D1);
}
```

This code example contains two subprograms that initialize the `Data` structure:

- `init_data()`, which receives the data structure as a reference (using a pointer) and initializes it, and
- `get_init_data()`, which returns the initialized structure.

In C, we generally avoid implementing functions such as `get_init_data()` because of the extra copy that is needed for the function return.

This is the corresponding implementation in Ada:

[Ada]

```
procedure Init_Record is

   type Data is record
      Prev : Integer;
      Curr : Integer;
   end record;

   procedure Init (D : out Data) is
   begin
      D := (Prev => 0, Curr => 1);
   end Init;

   function Init return Data is
      D : constant Data := (Prev => 0, Curr => 1);
   begin
      return D;
   end Init;

   D1 : Data;

   pragma Unreferenced (D1);
begin
   D1 := Init;

   Init (D1);
end Init_Record;
```

In this example, we have two versions of `Init`: one using a procedural form, and the other one using a functional form. Note that, because of Ada's support for subprogram overloading, we can use the same name for both subprograms.

The issue is that assignment of a function result entails a copy, just as if we assigned one variable to another. For example, when assigning a function result to a constant, the function result is copied into the memory for the constant. That's what is happening in the above examples for the initialized variables.

Therefore, in terms of performance, the same recommendations apply: for large types we should avoid writing functions like the `Init` function above. Instead, we should use the procedural form of

`Init`. The reason is that the compiler necessarily generates a copy for the `Init` function, while the `Init` procedure uses a reference for the output parameter, so that the actual record initialization is performed in place in the caller's argument.

An exception to this is when we use functions returning values of limited types, which by definition do not allow assignment. Here, to avoid allowing something that would otherwise look suspiciously like an assignment, the compiler generates the function body so that it builds the result directly into the object being assigned. No copy takes place.

We could, for example, rewrite the example above using limited types:

[Ada]

```
procedure Init_Limited_Record is

   type Data is limited record
      Prev : Integer;
      Curr : Integer;
   end record;

   function Init return Data is
   begin
      return D : Data do
         D.Prev := 0;
         D.Curr := 1;
      end return;
   end Init;

   D1 : Data := Init;

   pragma Unreferenced (D1);
begin
   null;
end Init_Limited_Record;
```

In this example, `D1 : Data := Init;` has the same cost as the call to the procedural form — `Init (D1);` — that we've seen in the previous example. This is because the assignment is done in place.

Note that limited types require the use of the extended return statements (`return ... do ... end return`) in function implementations. Also note that, because the `Data` type is limited, we can only use the `Init` function in the declaration of `D1`; a statement in the code such as `D1 := Init;` is therefore forbidden.

CHAPTER

NINE

ARGUMENTATION AND BUSINESS PERSPECTIVES

The technical benefits of a migration from C to Ada are usually relatively straightforward to demonstrate. Hopefully, this course provides a good basis for it. However, when faced with an actual business decision to make, additional considerations need to be taken into account, such as return on investment, perennity of the solution, tool support, etc. This section will cover a number of usual questions and provide elements of answers.

9.1 What's the expected ROI of a C to Ada transition?

Switching from one technology to another is a cost, may that be in terms of training, transition of the existing environment or acquisition of new tools. This investment needs to be matched with an expected return on investment, or ROI, to be consistent. Of course, it's incredibly difficult to provide a firm answer to how much money can be saved by transitioning, as this is highly dependent on specific project objectives and constraints. We're going to provide qualitative and quantitative arguments here, from the perspective of a project that has to reach relatively high level of integrity, that is to say a system where the occurrence of a software failure is a relatively costly event.

From a qualitative standpoint, there are various times in the software development life cycle where defects can be found:

1. on the developer's desk
2. during component testing
3. during integration testing
4. after deployment
5. during maintenance

Numbers from studies vary greatly on the relative costs of defects found at each of these phases, but there's a clear ordering between them. For example, a defect found while developing is orders of magnitude less expensive to fix than a defect found e.g. at integration time, which may involve costly debugging sessions and slow down the entire system acceptance. The whole purpose of Ada and SPARK is to push defect detection to the developer's desk as much as possible; at least for all of these defects that can be identified at that level. While the strict act of writing software may be taking more effort because of all of the additional safeguards, this should have a significant and positive impact down the line and help to control costs overall. The exact value this may translate into is highly business dependent.

From a quantitative standpoint, two studies have been done almost 25 years apart and provide similar insights:

- Rational Software in 1995 found that the cost of developing software in Ada was overall half as much as the cost of developing software in C.
- VDC ran a study in 2018, finding that the cost savings of developing with Ada over C ranged from 6% to 38% in savings.

From a qualitative standpoint, in particular with regards to Ada and C from a formal proof perspective, an interesting presentation was made in 2017 by two researchers. They tried to apply formal proof on the same piece of code, developed in Ada/SPARK on one end and C/Frama-C on the other. Their results indicate that the Ada/SPARK technology is indeed more conducive to formal proof methodologies.

Although all of these studies have their own biases, they provide a good idea of what to expect in terms of savings once the initial investment in switching to Ada is made. This is assuming everything else is equal, in particular that the level of integrity is the same. In many situations, the migration to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will raise (it's more expensive to develop better software) and Ada is viewed as a means to mitigate this rise in development costs.

That being said, the point of this argument is not to say that it's not possible to write very safe and secure software with languages different than Ada. With the right expertise, the right processes and the right tools, it's done every day. The point is that Ada overall reduces the level of processes, expertise and tools necessary and will allow to reach the same target at a lower cost.

9.2 Who is using Ada today?

Ada was initially born as a DoD project, and thus got its initial customer base in aerospace and defence (A&D). At the time these lines are written and from the perspective of AdaCore, A&D is still the largest consumer of Ada today and covers about 70% of the market. This creates a consistent and long lasting set of established users as these project last often for decades, using the same codebase migrating from platform to platform.

More recently however, there has been an emerging interest for Ada in new communities of users such as automotive, medical device, industrial automation and overall cyber-security. This can probably be explained by a rise of safety, reliability and cyber-security requirements. The market is moving relatively rapidly today and we're anticipating an increase of the Ada footprint in these domains, while still remaining a technology of choice for the development of mission critical software.

9.3 What is the future of the Ada technology?

The first piece of the answer lies in the user base of the Ada language, as seen in the previous question. Projects using Ada in the aerospace and defence domain maintain source code over decades, providing healthy funding foundation for Ada-based technologies.

AdaCore being the author of this course, it's difficult for us to be fair in our description of other Ada compilation technologies. We will leave to the readers the responsibility of forging their own opinion. If they present a credible alternative to the GNAT compiler, then this whole section can be considered as void.

Assuming GNAT is the only option available, and acknowledging that this is an argument that we're hearing from a number of Ada adopters, let's discuss the "sole source" issue.

First of all, it's worth noting that industries are using a lot of software that is provided by only one source, so while non-ideal, these situations are also quite common.

In the case of the GNAT compiler however, while AdaCore is the main maintainer, this maintenance is done as part of an open-source community. This means that nothing prevents a third party to start selling a competing set of products based on the same compiler, provided that it too adopts the open-source approach. Our job is to be more cost-effective than the alternative, and indeed for the vast part this has prevented a competing offering to emerge. However, should AdaCore disappear or switch focus, Ada users would not be prevented from carrying on using its software (there is no lock) and a third party could take over maintenance. This is not a theoretical case, this

has been done in the past either by companies looking at supporting their own version of GNAT, vendors occupying a specific niche that was left uncovered , or hobbyists developing their own builds.

With that in mind, it's clear that the "sole source" provider issue is a circumstantial — nothing is preventing other vendors from emerging if the conditions are met.

9.4 Is the Ada toolset complete?

A language by itself is of little use for the development of safety-critical software. Instead, a complete toolset is needed to accompany the development process, in particular tools for edition, testing, static analysis, etc.

AdaCore provides a number of these tools either in through its core or add-on package. These include (as of 2019):

- An IDE (GNAT Studio)
- An Eclipse plug-in (GNATbench)
- A debugger (GDB)
- A testing tool (GNATtest)
- A structural code coverage tool (GNATcoverage)
- A metric computation tool (GNATmetric)
- A coding standard checker (GNATcheck)
- Static analysis tools (CodePeer, SPARK Pro)
- A Simulink code generator (QGen)
- An Ada parser to develop custom tools (libadalang)

Ada is, however, an internationally standardized language, and many companies are providing third party solution to complete the toolset. Overall, the language can be and is used with tools on par with their equivalent C counterparts.

9.5 Where can I find Ada or SPARK developers?

A common question from teams on the verge of selecting Ada and SPARK is how to manage the developer team growth and turnover. While Ada and SPARK are taught by a growing number of universities worldwide, it may still be challenging to hire new staff with prior Ada experience.

Fortunately, Ada's base semantics are very close to those of C/C++, so that a good embedded software developer should be able to learn it relatively easily. This course is definitely a resource available to get started. Online training material is also available, together with on-site in person training.

In general, getting an engineer operational in Ada and SPARK shouldn't take more than a few weeks worth of time.

9.6 How to introduce Ada and SPARK in an existing code base?

The most common scenario when introducing Ada and SPARK to a project or a team is to do it within an pre-existing C codebase, which can already spread over hundreds of thousands if not

millions lines of code. Re-writing this software to Ada or SPARK is of course not practical and counterproductive.

Most team select either a small piece of existing code which deserves particular attention, or new modules to develop, and concentrate on this. Developing this module or part of the application will also help in developing the coding patterns to be used for the particular project and company. This typically concentrates an effort of a few people on a few thousands lines of code. The resulting code can be linked to the rest of the C application. From there, the newly established practices and their benefit can slowly spread through the rest of the environment.

Establishing this initial core in Ada and SPARK is critical, and while learning the language isn't a particularly difficult task, applying it to its full capacity may require some expertise. One possibility to accelerate this initial process is to use AdaCore mentorship services.

CHAPTER

TEN

CONCLUSION

Although Ada's syntax might seem peculiar to C developers at first glance, it was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner — as it is often the case in C.

Especially in the embedded domain, C developers are used to working at a very low level, which includes mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations because it was designed to replace Assembly language for faster, more efficient programming.

Ada can be used to describe high level semantics and architectures. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded Assembly code or bit-level data management. However, although Ada supports bitwise operations such as masks and shifts, they should be relatively rarely needed. When translating C code to Ada, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other. For example, we can interpret an arbitrary data type as a bit-field and perform low-level operations on it.

Because Ada is a strongly typed language, it doesn't define any implicit type conversions like C. If we try to compile Ada code that contains type mismatches, we'll get a compilation error. Because the compiler prevents mixing variables of different types without explicit type conversion, we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect. In this sense, Ada's type system encourages programmers to think about data at a high level of abstraction. Ada supports overlays and unchecked conversions as a way of converting between unrelated data type, which are typically used for interfacing with low-level elements such as registers.

In Ada, arrays aren't interchangeable with operations on pointers like in C. Also, array types are considered first-class citizens and have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and we can specify both the starting and ending bounds. In addition, Ada offers high-level operations for copying, slicing, and assigning values to arrays.

Although Ada supports pointers, most situations that would require a pointer in C do not in Ada. In the vast majority of the cases, indirect memory management can be hidden from the developer and thus prevent many potential errors. In C, pointers are typically used to pass references to subprograms, for example. In contrast, Ada parameter modes indicate the flow of information to the reader, leaving the means of passing that information to the compiler.

When translating pointers from C code to Ada, we need to assess whether they are needed in the first place. Ada pointers (access types) should only be used with complex structures that cannot be allocated at run-time. There are many situations that would require a pointer in C, but do not in Ada. For example, arrays — even when dynamically allocated —, results of functions, passing of large structures as parameters, access to registers, etc.

Because of the absence of namespaces, global names in C tend to be very long. Also, because of the absence of overloading, they can even encode type names in their name. In Ada, a package is

a namespace. Also, we can use the private part of a package to declare private types and private subprograms. In fact, private types are useful for preventing the users of those types from depending on the implementation details. Another use-case is the prevention of package users from accessing the package state/data arbitrarily.

Ada has a dedicated set of features for interfacing with other languages, so we can easily interface with our existing C code before translating it to Ada. Also, GNAT includes automatic binding generators. Therefore, instead of re-writing the entire C code upfront, which isn't practical or cost-effective, we can selectively translate modules from C to Ada.

When it comes to implementing concurrency and real time, Ada offers several options. Ada provides high level constructs such as tasks and protected objects to express concurrency and synchronization, which can be used when running on top of an operating system such as Linux. On more constrained systems, such as bare metal or some real-time operating systems, a subset of the Ada tasking capabilities — known as the Ravenscar and Jorvik profiles — is available. Though restricted, this subset also has nice properties, in particular the absence of deadlock,the absence of priority inversion, schedulability and very small footprint. On bare metal systems, this also essentially means that Ada comes with its own real-time kernel. The advantage of using the full Ada tasking model or the restricted profiles is to enhance portability.

Ada includes many features typically used for embedded programming:

- Built-in support for handling interrupts, so we can process interrupts by attaching a handler — as a protected procedure — to it.
- Built-in support for handling both volatile and atomic data.
- Support for register overlays, which we can use to create a structure that facilitates manipulating bits from registers.
- Support for creating data streams for serialization of arbitrary information and transmission over a communication channel, such as a serial port.
- Built-in support for fixed-point arithmetic, which is an option when our target device doesn't have a floating-point unit or the result of calculations needs to be bit-exact.

Also, Ada compilers such as GNAT have built-in support for directly mixing Ada and Assembly code.

Ada also supports contracts, which can be associated with types and variables to refine values and define valid and invalid values. The most common kind of contract is a *range constraint* — using the `range` reserved word. Ada also supports contract-based programming in the form of preconditions and postconditions. One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations.

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. In C, variability is usually achieved through macros and function pointers, the former being tied to static variability and the latter to dynamic variability. Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. Examples of static variability in Ada are: genericity, simple derivation, configuration pragma files, and configuration packages. Examples of dynamic variability in Ada are: records with discriminants, variant records — which may include the use of unions —, object orientation, pointers to subprograms, and design by components using dynamic libraries.

There shouldn't be significant performance differences between code written in Ada and code written in C — provided that they are semantically equivalent. One reason is that the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. Therefore, they should be equivalent on average. However, when a piece of code in Ada is significantly slower than its counterpart in C, this usually comes from the fact that, while the two pieces of code appear to be semantically equivalent, they happen to be actually quite different. Fortunately, there are strategies that we can use to improve the performance and make it equivalent to the C version. These are some examples:

- Clever use of compilation switches, which might optimize the performance of an application significantly.

- Suppression of checks at specific parts of the implementation.
  - Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots.
- Restriction of assertions to development code.
  - For example, we may use assertions in the debug version of the code and turn them off in the release version.
  - Also, we may use formal proof to decide which assertions we turn off in the release version. By formally proving that assertions will never fail at run-time, we can safely deactivate them.

Formal proof — a form of static analysis — can give strong guarantees about checks, for all possible conditions and all possible inputs. It verifies conditions prior to execution, even prior to compilation, so we can remove bugs earlier in the development phase. This is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after deployment. Preventing bug introduction into the deployed system is the least expensive approach of all.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the **`gnatprove`** verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof.

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering *exceptions* that interrupt the normal flow of execution. However, we may be able to prove that the language-defined checks won't raise exceptions at run-time. This is known as proving *Absence of Run-Time Errors*. Successful proof of these checks is highly significant in itself. One of the major resulting benefits is that we can deploy the final executable with checks disabled.

In many situations, the migration of C code to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will raise. However, Ada is a more expressive, powerful language, designed to reduce errors earlier in the life-cycle, thus reducing costs. Therefore, Ada makes it possible to write very safe and secure software at a lower cost than languages such as C.

CHAPTER

ELEVEN

APPENDIX A: HANDS-ON OBJECT-ORIENTED PROGRAMMING

The goal of this appendix is to present a hands-on view on how to translate a system from C to Ada and improve it with object-oriented programming.

11.1 System Overview

Let's start with an overview of a simple system that we'll implement and use below. The main system is called AB and it combines two systems A and B. System AB is not supposed to do anything useful. However, it can serve as a good model for the hands-on we're about to start.

This is a list of requirements for the individual systems A and B, and the combined system AB:

- System A:
  - The system can be activated and deactivated.
    - During activation, the system's values are reset.
  - Its current value (in floating-point) can be retrieved.
    - This value is the average of the two internal floating-point values.
  - Its current state (activated or deactivated) can be retrieved.
- System B:
  - The system can be activated and deactivated.
    - During activation, the system's value is reset.
  - Its current value (in floating-point) can be retrieved.
  - Its current state (activated or deactivated) can be retrieved.
- System AB
  - The system contains an instance of system A and an instance of system B.
  - The system can be activated and deactivated.
    - System AB activates both systems A and B during its own activation.
    - System AB deactivates both systems A and B during its own deactivation.
  - Its current value (in floating-point) can be retrieved.
    - This value is the average of the current values of systems A and B.
  - Its current state (activated or deactivated) can be retrieved.
    - AB is only considered activated when both systems A and B are activated.
  - The system's health can be checked.

* This check consists in calculating the absolute difference D between the current values of systems A and B and checking whether D is below a threshold of 0.1.

The source-code in the following section contains an implementation of these requirements.

11.2 Non Object-Oriented Approach

In this section, we look into implementations (in both C and Ada) of system AB that don't make use of object-oriented programming.

11.2.1 Starting point in C

Let's start with an implementation in C for the system described above:

[C]

```
typedef struct {
    float val[2];
    int   active;
} A;

void A_activate (A *a);

int A_is_active (A *a);

float A_value (A *a);

void A_deactivate (A *a);
```

```
#include "system_a.h"

void A_activate (A *a)
{
    int i;

    for (i = 0; i < 2; i++)
    {
        a->val[i] = 0.0;
    }
    a->active = 1;
}

int A_is_active (A *a)
{
    return a->active == 1;
}

float A_value (A *a)
{
    return (a->val[0] + a->val[1]) / 2.0;
}

void A_deactivate (A *a)
{
    a->active = 0;
}
```

```c
typedef struct {
    float val;
    int   active;
} B;

void B_activate (B *b);

int B_is_active (B *b);

float B_value (B *b);

void B_deactivate (B *b);
```

```c
#include "system_b.h"

void B_activate (B *b)
{
    b->val    = 0.0;
    b->active = 1;
}

int B_is_active (B *b)
{
    return b->active == 1;
}

float B_value (B *b)
{
    return b->val;
}

void B_deactivate (B *b)
{
    b->active = 0;
}
```

```c
#include "system_a.h"
#include "system_b.h"

typedef struct {
    A a;
    B b;
} AB;

void AB_activate (AB *ab);

int AB_is_active (AB *ab);

float AB_value (AB *ab);

int AB_check (AB *ab);

void AB_deactivate (AB *ab);
```

```c
#include <math.h>
#include "system_ab.h"

void AB_activate (AB *ab)
{
    A_activate (&ab->a);
    B_activate (&ab->b);
```

(continues on next page)

(continued from previous page)

```
}

int AB_is_active (AB *ab)
{
    return A_is_active(&ab->a) && B_is_active(&ab->b);
}

float AB_value (AB *ab)
{
    return (A_value (&ab->a) + B_value (&ab->b)) / 2;
}

int AB_check (AB *ab)
{
    const float threshold = 0.1;

    return fabs (A_value (&ab->a) - B_value (&ab->b)) < threshold;
}

void AB_deactivate (AB *ab)
{
    A_deactivate (&ab->a);
    B_deactivate (&ab->b);
}
```

```
#include <stdio.h>
#include "system_ab.h"

void display_active (AB *ab)
{
    if (AB_is_active (ab))
        printf ("System AB is active.\n");
    else
        printf ("System AB is not active.\n");
}

void display_check (AB *ab)
{
    if (AB_check (ab))
        printf ("System AB check: PASSED.\n");
    else
        printf ("System AB check: FAILED.\n");
}

int main()
{
    AB s;

    printf ("Activating system AB...\n");
    AB_activate (&s);

    display_active (&s);
    display_check (&s);

    printf ("Deactivating system AB...\n");
    AB_deactivate (&s);

    display_active (&s);
}
```

Here, each system is implemented in a separate set of header and source-code files. For example,

the API of system AB is in `system_ab.h` and its implementation in `system_ab.c`.

In the main application, we instantiate system AB and activate it. Then, we proceed to display the activation state and the result of the system's health check. Finally, we deactivate the system and display the activation state again.

11.2.2 Initial translation to Ada

The direct implementation in Ada is:

[Ada]

```
package System_A is

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

   type A is record
      Val    : Val_Array (1 .. 2);
      Active : Boolean;
   end record;

   procedure A_Activate (E : in out A);

   function A_Is_Active (E : A) return Boolean;

   function A_Value (E : A) return Float;

   procedure A_Deactivate (E : in out A);

end System_A;
```

```
package body System_A is

   procedure A_Activate (E : in out A) is
   begin
      E.Val    := (others => 0.0);
      E.Active := True;
   end A_Activate;

   function A_Is_Active (E : A) return Boolean is
   begin
      return E.Active;
   end A_Is_Active;

   function A_Value (E : A) return Float is
   begin
      return (E.Val (1) + E.Val (2)) / 2.0;
   end A_Value;

   procedure A_Deactivate (E : in out A) is
   begin
      E.Active := False;
   end A_Deactivate;

end System_A;
```

```
package System_B is

   type B is record
      Val    : Float;
```

(continues on next page)

(continued from previous page)

```
      Active : Boolean;
   end record;

   procedure B_Activate (E : in out B);

   function B_Is_Active (E : B) return Boolean;

   function B_Value (E : B) return Float;

   procedure B_Deactivate (E : in out B);

end System_B;
```

```
package body System_B is

   procedure B_Activate (E : in out B) is
   begin
      E.Val    := 0.0;
      E.Active := True;
   end B_Activate;

   function B_Is_Active (E : B) return Boolean is
   begin
      return E.Active;
   end B_Is_Active;

   function B_Value (E : B) return Float is
   begin
      return E.Val;
   end B_Value;

   procedure B_Deactivate (E : in out B) is
   begin
      E.Active := False;
   end B_Deactivate;

end System_B;
```

```
with System_A; use System_A;
with System_B; use System_B;

package System_AB is

   type AB is record
      SA : A;
      SB : B;
   end record;

   procedure AB_Activate (E : in out AB);

   function AB_Is_Active (E : AB) return Boolean;

   function AB_Value (E : AB) return Float;

   function AB_Check (E : AB) return Boolean;

   procedure AB_Deactivate (E : in out AB);

end System_AB;
```

```
package body System_AB is

   procedure AB_Activate (E : in out AB) is
   begin
      A_Activate (E.SA);
      B_Activate (E.SB);
   end AB_Activate;

   function AB_Is_Active (E : AB) return Boolean is
   begin
      return A_Is_Active (E.SA) and B_Is_Active (E.SB);
   end AB_Is_Active;

   function AB_Value (E : AB) return Float is
   begin
      return (A_Value (E.SA) + B_Value (E.SB)) / 2.0;
   end AB_Value;

   function AB_Check (E : AB) return Boolean is
      Threshold : constant := 0.1;
   begin
      return abs (A_Value (E.SA) - B_Value (E.SB)) < Threshold;
   end AB_Check;

   procedure AB_Deactivate (E : in out AB) is
   begin
      A_Deactivate (E.SA);
      B_Deactivate (E.SB);
   end AB_Deactivate;

end System_AB;
```

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

with System_AB;   use System_AB;

procedure Main is

   procedure Display_Active (E : AB) is
   begin
      if AB_Is_Active (E) then
         Put_Line ("System AB is active");
      else
         Put_Line ("System AB is not active");
      end if;
   end Display_Active;

   procedure Display_Check (E : AB) is
   begin
      if AB_Check (E) then
         Put_Line ("System AB check: PASSED");
      else
         Put_Line ("System AB check: FAILED");
      end if;
   end Display_Check;

   S : AB;
begin
   Put_Line ("Activating system AB...");
   AB_Activate (S);
```

(continues on next page)

(continued from previous page)

```
   Display_Active (S);
   Display_Check (S);

   Put_Line ("Deactivating system AB...");
   AB_Deactivate (S);

   Display_Active (S);
end Main;
```

As you can see, this is a direct translation that doesn't change much of the structure of the original C code. Here, the goal was to simply translate the system from one language to another and make sure that the behavior remains the same.

11.2.3 Improved Ada implementation

By analyzing this direct implementation, we may notice the following points:

- Packages `System_A`, `System_B` and `System_AB` are used to describe aspects of the same system. Instead of having three distinct packages, we could group them as child packages of a common parent package — let's call it `Simple`, since this system is supposed to be simple. This approach has the advantage of allowing us to later use the parent package to implement functionality that is common for all parts of the system.
- Since we have subprograms that operate on types `A`, `B` and `AB`, we should avoid exposing the record components by moving the type declarations to the private part of the corresponding packages.
- Since Ada supports subprogram overloading — as discussed in *this section from chapter 2* (page 45) —, we don't need to have different names for subprograms with similar functionality. For example, instead of having `A_Is_Active` and `B_Is_Active`, we can simply name these functions `Is_Active` for both types `A` and `B`.
- Some of the functions — such as `A_Is_Active` and `A_Value` — are very simple, so we could simplify them with expression functions.

This is an update to the implementation that addresses all the points above:

[Ada]

```
package Simple
  with Pure
is
end Simple;
```

```
package Simple.System_A is

   type A is private;

   procedure Activate (E : in out A);

   function Is_Active (E : A) return Boolean;

   function Value (E : A) return Float;

   procedure Finalize (E : in out A);

private

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

(continues on next page)

(continued from previous page)

```
   type A is record
      Val    : Val_Array (1 .. 2);
      Active : Boolean;
   end record;

end Simple.System_A;
```

```
package body Simple.System_A is

   procedure Activate (E : in out A) is
   begin
      E.Val    := (others => 0.0);
      E.Active := True;
   end Activate;

   function Is_Active (E : A) return Boolean is
      (E.Active);

   function Value (E : A) return Float is
   begin
      return (E.Val (1) + E.Val (2)) / 2.0;
   end Value;

   procedure Finalize (E : in out A) is
   begin
      E.Active := False;
   end Finalize;

end Simple.System_A;
```

```
package Simple.System_B is

   type B is private;

   procedure Activate (E : in out B);

   function Is_Active (E : B) return Boolean;

   function Value (E : B) return Float;

   procedure Finalize (E : in out B);

private

   type B is record
      Val    : Float;
      Active : Boolean;
   end record;

end Simple.System_B;
```

```
package body Simple.System_B is

   procedure Activate (E : in out B) is
   begin
      E.Val    := 0.0;
      E.Active := True;
   end Activate;

   function Is_Active (E : B) return Boolean is
```

(continues on next page)

(continued from previous page)

```
   begin
      return E.Active;
   end Is_Active;

   function Value (E : B) return Float is
     (E.Val);

   procedure Finalize (E : in out B) is
   begin
      E.Active := False;
   end Finalize;

end Simple.System_B;
```

```
with Simple.System_A; use Simple.System_A;
with Simple.System_B; use Simple.System_B;

package Simple.System_AB is

   type AB is private;

   procedure Activate (E : in out AB);

   function Is_Active (E : AB) return Boolean;

   function Value (E : AB) return Float;

   function Check (E : AB) return Boolean;

   procedure Finalize (E : in out AB);

private

   type AB is record
      SA : A;
      SB : B;
   end record;

end Simple.System_AB;
```

```
package body Simple.System_AB is

   procedure Activate (E : in out AB) is
   begin
      Activate (E.SA);
      Activate (E.SB);
   end Activate;

   function Is_Active (E : AB) return Boolean is
      (Is_Active (E.SA) and Is_Active (E.SB));

   function Value (E : AB) return Float is
      ((Value (E.SA) + Value (E.SB)) / 2.0);

   function Check (E : AB) return Boolean is
      Threshold : constant := 0.1;
   begin
      return abs (Value (E.SA) - Value (E.SB)) < Threshold;
   end Check;
```

(continues on next page)

(continued from previous page)

```
   procedure Finalize (E : in out AB) is
   begin
      Finalize (E.SA);
      Finalize (E.SB);
   end Finalize;

end Simple.System_AB;
```

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

with Simple.System_AB; use Simple.System_AB;

procedure Main is

   procedure Display_Active (E : AB) is
   begin
      if Is_Active (E) then
         Put_Line ("System AB is active");
      else
         Put_Line ("System AB is not active");
      end if;
   end Display_Active;

   procedure Display_Check (E : AB) is
   begin
      if Check (E) then
         Put_Line ("System AB check: PASSED");
      else
         Put_Line ("System AB check: FAILED");
      end if;
   end Display_Check;

   S : AB;
begin
   Put_Line ("Activating system AB...");
   Activate (S);

   Display_Active (S);
   Display_Check (S);

   Put_Line ("Deactivating system AB...");
   Finalize (S);

   Display_Active (S);
end Main;
```

11.3 First Object-Oriented Approach

Until now, we haven't used any of the object-oriented programming features of the Ada language. So we can start by analyzing the API of systems A and B and deciding how to best abstract some of its elements using object-oriented programming.

11.3.1 Interfaces

The first thing we may notice is that we actually have two distinct sets of APIs there:

- one API for activating and deactivating the system.

- one API for retrieving the value of the system.

We can use this distinction to declare two interface types:

- `Activation_IF` for the `Activate` and `Deactivate` procedures and the `Is_Active` function;
- `Value_Retrieval_IF` for the `Value` function.

This is how the declaration could look like:

```
type Activation_IF is interface;

procedure Activate (E : in out Activation_IF) is abstract;
function Is_Active (E : Activation_IF) return Boolean is abstract;
procedure Deactivate (E : in out Activation_IF) is abstract;

type Value_Retrieval_IF is interface;

function Value (E : Value_Retrieval_IF) return Float is abstract;
```

Note that, because we are declaring interface types, all operations on those types must be abstract or, in the case of procedures, they can also be declared `null`. For example, we could change the declaration of the procedures above to this:

```
procedure Activate (E : in out Activation_IF) is null;
procedure Deactivate (E : in out Activation_IF) is null;
```

When an operation is declared abstract, we must override it for the type that derives from the interface. When a procedure is declared `null`, it acts as a do-nothing default. In this case, overriding the operation is optional for the type that derives from this interface.

11.3.2 Base type

Since the original system needs both interfaces we've just described, we have to declare another type that combines those interfaces. We can do this by declaring the interface type `Sys_Base`, which serves as the base type for systems A and B. This is the declaration:

```
type Sys_Base is interface and Activation_IF and Value_Retrieval_IF;
```

Since the system activation functionality is common for both systems A and B, we could implement it as part of `Sys_Base`. That would require changing the declaration from a simple interface to an abstract record:

```
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
  with null record;
```

Now, we can add the Boolean component to the record (as a private component) and override the subprograms of the `Activation_IF` interface. This is the adapted declaration:

```
   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with private;

   overriding procedure Activate (E : in out Sys_Base);
   overriding function Is_Active (E : Sys_Base) return Boolean;
   overriding procedure Deactivate (E : in out Sys_Base);

private

   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with record
      Active : Boolean;
   end record;
```

11.3.3 Derived types

In the declaration of the `Sys_Base` type we've just seen, we're not overriding the `Value` function — from the `Value_Retrieval_IF` interface — for the `Sys_Base` type, so it remains an abstract function for `Sys_Base`. Therefore, the `Sys_Base` type itself remains abstract and needs be explicitly declared as such.

We use this strategy to ensure that all types derived from `Sys_Base` need to implement their own version of the `Value` function. For example:

```
type A is new Sys_Base with private;

overriding function Value (E : A) return Float;
```

Here, the A type is derived from the `Sys_Base` and it includes its own version of the `Value` function by overriding it. Therefore, A is not an abstract type anymore and can be used to declare objects:

```
procedure Main is
   Obj : A;
   V   : Float;
begin
   Obj.Activate;
   V := Obj.Value;
end Main;
```

Important

Note that the use of the `overriding` keyword in the subprogram declaration is not strictly necessary. In fact, we could leave this keyword out, and the code would still compile. However, if provided, the compiler will check whether the information is correct.

Using the `overriding` keyword can help to avoid bad surprises — when you *may think* that you're overriding a subprogram, but you're actually not. Similarly, you can also write `not overriding` to be explicit about subprograms that are new primitives of a derived type. For example:

```
not overriding function Check (E : AB) return Boolean;
```

We also need to declare the values that are used internally in systems A and B. For system A, this is the declaration:

```
   type A is new Sys_Base with private;

   overriding function Value (E : A) return Float;

private

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

   type A is new Sys_Base with record
      Val : Val_Array (1 .. 2);
   end record;
```

11.3.4 Subprograms from parent

In the previous implementation, we've seen that the `A_Activate` and `B_Activate` procedures perform the following steps:

- initialize internal values;

- indicate that the system is active (by setting the `Active` flag to `True`).

In the implementation of the `Activate` procedure for the `Sys_Base` type, however, we're only dealing with the second step. Therefore, we need to override the `Activate` procedure and make sure that we initialize internal values as well. First, we need to declare this procedure for type `A`:

```
type A is new Sys_Base with private;

overriding procedure Activate (E : in out A);
```

In the implementation of `Activate`, we should call the `Activate` procedure from the parent (`Sys_Base`) to ensure that whatever was performed for the parent will be performed in the derived type as well. For example:

```
overriding procedure Activate (E : in out A) is
begin
   E.Val := (others => 0.0);
   Sys_Base (E).Activate;    --  Calling Activate for Sys_Base type:
                             --  this call initializes the Active flag.
end;
```

Here, by writing `Sys_Base (E)`, we're performing a view conversion. Basically, we're telling the compiler to view `E` not as an object of type `A`, but of type `Sys_Base`. When we do this, any operation performed on this object will be done as if it was an object of `Sys_Base` type, which includes calling the `Activate` procedure of the `Sys_Base` type.

Important

If we write `T (Obj).Proc`, we're telling the compiler to call the `Proc` procedure of type `T` and apply it on `Obj`.

If we write `T'Class (Obj).Proc`, however, we're telling the compiler to dispatch the call. For example, if `Obj` is of derived type `T2` and there's an overridden `Proc` procedure for type `T2`, then this procedure will be called instead of the `Proc` procedure for type `T`.

11.3.5 Type AB

While the implementation of systems A and B is almost straightforward, it gets more interesting in the case of system AB. Here, we have a similar API, but we don't need the activation mechanism implemented in the abstract type `Sys_Base`. Therefore, deriving from `Sys_Base` is not the best option. Instead, when declaring the `AB` type, we can simply use the same interfaces as we did for `Sys_Base`, but keep it independent from `Sys_Base`. For example:

```
   type AB is new Activation_IF and Value_Retrieval_IF with private;

private

   type AB is new Activation_IF and Value_Retrieval_IF with record
      SA : A;
      SB : B;
   end record;
```

Naturally, we still need to override all the subprograms that are part of the `Activation_IF` and `Value_Retrieval_IF` interfaces. Also, we need to implement the additional `Check` function that was originally only available on system AB. Therefore, we declare these subprograms:

```
overriding procedure Activate (E : in out AB);
overriding function Is_Active (E : AB) return Boolean;
```

(continues on next page)

(continued from previous page)

```
overriding procedure Deactivate (E : in out AB);

overriding function Value (E : AB) return Float;

not overriding function Check (E : AB) return Boolean;
```

11.3.6 Updated source-code

Finally, this is the complete source-code example:

[Ada]

```
package Simple is

   type Activation_IF is interface;

   procedure Activate (E : in out Activation_IF) is abstract;
   function Is_Active (E : Activation_IF) return Boolean is abstract;
   procedure Deactivate (E : in out Activation_IF) is abstract;

   type Value_Retrieval_IF is interface;

   function Value (E : Value_Retrieval_IF) return Float is abstract;

   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
     with private;

   overriding procedure Activate (E : in out Sys_Base);
   overriding function Is_Active (E : Sys_Base) return Boolean;
   overriding procedure Deactivate (E : in out Sys_Base);

private

   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
     with record
      Active : Boolean;
   end record;

end Simple;
```

```
package body Simple is

   overriding procedure Activate (E : in out Sys_Base) is
   begin
      E.Active := True;
   end Activate;

   overriding function Is_Active (E : Sys_Base) return Boolean is
      (E.Active);

   overriding procedure Deactivate (E : in out Sys_Base) is
   begin
      E.Active := False;
   end Deactivate;

end Simple;
```

```
package Simple.System_A is
```

(continues on next page)

(continued from previous page)

```
   type A is new Sys_Base with private;

   overriding procedure Activate (E : in out A);

   overriding function Value (E : A) return Float;

private

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

   type A is new Sys_Base with record
      Val : Val_Array (1 .. 2);
   end record;

end Simple.System_A;
```

```
package body Simple.System_A is

   procedure Activate (E : in out A) is
   begin
      E.Val := (others => 0.0);
      Sys_Base (E).Activate;
   end Activate;

   function Value (E : A) return Float is
      pragma Assert (E.Val'Length = 2);
   begin
      return (E.Val (1) + E.Val (2)) / 2.0;
   end Value;

end Simple.System_A;
```

```
package Simple.System_B is

   type B is new Sys_Base with private;

   overriding procedure Activate (E : in out B);

   overriding function Value (E : B) return Float;

private

   type B is new Sys_Base with record
      Val : Float;
   end record;

end Simple.System_B;
```

```
package body Simple.System_B is

   procedure Activate (E : in out B) is
   begin
      E.Val := 0.0;
      Sys_Base (E).Activate;
   end Activate;

   function Value (E : B) return Float is
     (E.Val);

end Simple.System_B;
```

```
with Simple.System_A; use Simple.System_A;
with Simple.System_B; use Simple.System_B;

package Simple.System_AB is

   type AB is new Activation_IF and Value_Retrieval_IF with private;

   overriding procedure Activate (E : in out AB);
   overriding function Is_Active (E : AB) return Boolean;
   overriding procedure Deactivate (E : in out AB);

   overriding function Value (E : AB) return Float;

   not overriding function Check (E : AB) return Boolean;

private

   type AB is new Activation_IF and Value_Retrieval_IF with record
      SA : A;
      SB : B;
   end record;

end Simple.System_AB;
```

```
package body Simple.System_AB is

   procedure Activate (E : in out AB) is
   begin
      E.SA.Activate;
      E.SB.Activate;
   end Activate;

   function Is_Active (E : AB) return Boolean is
     (E.SA.Is_Active and E.SB.Is_Active);

   procedure Deactivate (E : in out AB) is
   begin
      E.SA.Deactivate;
      E.SB.Deactivate;
   end Deactivate;

   function Value (E : AB) return Float is
     ((E.SA.Value + E.SB.Value) / 2.0);

   function Check (E : AB) return Boolean is
      Threshold : constant := 0.1;
   begin
      return abs (E.SA.Value - E.SB.Value) < Threshold;
   end Check;

end Simple.System_AB;
```

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

with Simple.System_AB; use Simple.System_AB;

procedure Main is

   procedure Display_Active (E : AB) is
   begin
      if Is_Active (E) then
```

(continues on next page)

(continued from previous page)

```
         Put_Line ("System AB is active");
      else
         Put_Line ("System AB is not active");
      end if;
   end Display_Active;

   procedure Display_Check (E : AB) is
   begin
      if Check (E) then
         Put_Line ("System AB check: PASSED");
      else
         Put_Line ("System AB check: FAILED");
      end if;
   end Display_Check;

   S : AB;
begin
   Put_Line ("Activating system AB...");
   Activate (S);

   Display_Active (S);
   Display_Check (S);

   Put_Line ("Deactivating system AB...");
   Deactivate (S);

   Display_Active (S);
end Main;
```

11.4 Further Improvements

When analyzing the complete source-code, we see that there are at least two areas that we could still improve.

11.4.1 Dispatching calls

The first issue concerns the implementation of the `Activate` procedure for types derived from `Sys_Base`. For those derived types, we're expecting that the `Activate` procedure of the parent must be called in the implementation of the overriding `Activate` procedure. For example:

```
package body Simple.System_A is

   procedure Activate (E : in out A) is
   begin
      E.Val := (others => 0.0);
      Activate (Sys_Base (E));
   end;
```

If a developer forgets to call that specific `Activate` procedure, however, the system won't work as expected. A better strategy could be the following:

- Declare a new `Activation_Reset` procedure for `Sys_Base` type.
- Make a dispatching call to the `Activation_Reset` procedure in the body of the `Activate` procedure (of the `Sys_Base` type).
- Let the derived types implement their own version of the `Activation_Reset` procedure.

This is a simplified view of the implementation using the points described above:

```
package Simple is

   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
     private;

   not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;

end Simple;

package body Simple is

   procedure Activate (E : in out Sys_Base) is
   begin
      --  NOTE: calling "E.Activation_Reset" does NOT dispatch!
      --        We need to use the 'Class attribute here --- not using this
      --        attribute is an error that will be caught by the compiler.
      Sys_Base'Class (E).Activation_Reset;

      E.Active := True;
   end Activate;

end Simple;

package Simple.System_A is

   type A is new Sys_Base with private;

private

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

   type A is new Sys_Base with record
      Val : Val_Array (1 .. 2);
   end record;

   overriding procedure Activation_Reset (E : in out A);

end Simple.System_A;

package body Simple.System_A is

   procedure Activation_Reset (E : in out A) is
   begin
      E.Val    := (others => 0.0);
   end Activation_Reset;

end Simple.System_A;
```

An important detail is that, in the implementation of `Activate`, we use `Sys_Base'Class` to ensure that the call to `Activation_Reset` will dispatch. If we had just written `E.Activation_Reset` instead, then we would be calling the `Activation_Reset` procedure of `Sys_Base` itself, which is not what we actually want here. The compiler will catch the error if you don't do the conversion to the class-wide type, because it would otherwise be a statically-bound call to an abstract procedure, which is illegal at compile-time.

11.4.2 Dynamic allocation

The next area that we could improve is in the declaration of the system AB. In the previous implementation, we were explicitly describing the two components of that system, namely a component

of type A and a component of type B:

```
type AB is new Activation_IF and Value_Retrieval_IF with record
   SA : A;
   SB : B;
end record;
```

Of course, this declaration matches the system requirements that we presented in the beginning. However, we could use strategies that make it easier to incorporate requirement changes later on. For example, we could hide this information about systems A and B by simply declaring an array of components of type `access Sys_Base'Class` and allocate them dynamically in the body of the package. Naturally, this approach might not be suitable for certain platforms. However, the advantage would be that, if we wanted to replace the component of type B by a new component of type C, for example, we wouldn't need to change the interface. This is how the updated declaration could look like:

```
type Sys_Base_Class_Access is access Sys_Base'Class;
type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;

type AB is limited new Activation_IF and Value_Retrieval_IF with record
   S_Array : Sys_Base_Array (1 .. 2);
end record;
```

Important

Note that we're now using the `limited` keyword in the declaration of type AB. That is necessary because we want to prevent objects of type AB being copied by assignment, which would lead to two objects having the same (dynamically allocated) subsystems A and B internally. This change requires that both `Activation_IF` and `Value_Retrieval_IF` are declared limited as well.

The body of `Activate` could then allocate those components:

```
procedure Activate (E : in out AB) is
begin
   E.S_Array := (new A, new B);
   for S of E.S_Array loop
      S.Activate;
   end loop;
end Activate;
```

And the body of `Deactivate` could deallocate them:

```
procedure Deactivate (E : in out AB) is
   procedure Free is
     new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
begin
   for S of E.S_Array loop
      S.Deactivate;
      Free (S);
   end loop;
end Deactivate;
```

11.4.3 Limited controlled types

Another approach that we could use to implement the dynamic allocation of systems A and B is to declare AB as a limited controlled type — based on the `Limited_Controlled` type of the `Ada.Finalization` package.

The `Limited_Controlled` type includes the following operations:

- `Initialize`, which is called when objects of a type derived from the `Limited_Controlled` type are being created — by declaring an object of the derived type, for example —, and
- `Finalize`, which is called when objects are being destroyed — for example, when an object gets out of scope at the end of a subprogram where it was created.

In this case, we must override those procedures, so we can use them for dynamic memory allocation. This is a simplified view of the update implementation:

```
package Simple.System_AB is

   type AB is limited new Ada.Finalization.Limited_Controlled and
     Activation_IF and Value_Retrieval_IF with private;

   overriding procedure Initialize (E : in out AB);
   overriding procedure Finalize   (E : in out AB);

end Simple.System_AB;

package body Simple.System_AB is

   overriding procedure Initialize (E : in out AB) is
   begin
      E.S_Array := (new A, new B);
   end Initialize;

   overriding procedure Finalize   (E : in out AB) is
      procedure Free is
        new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
   begin
      for S of E.S_Array loop
         Free (S);
      end loop;
   end Finalize;

end Simple.System_AB;
```

11.4.4 Updated source-code

Finally, this is the complete updated source-code example:

[Ada]

```
package Simple is

   type Activation_IF is limited interface;

   procedure Activate (E : in out Activation_IF) is abstract;
   function Is_Active (E : Activation_IF) return Boolean is abstract;
   procedure Deactivate (E : in out Activation_IF) is abstract;

   type Value_Retrieval_IF is limited interface;

   function Value (E : Value_Retrieval_IF) return Float is abstract;

   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
     private;

   overriding procedure Activate (E : in out Sys_Base);
   overriding function Is_Active (E : Sys_Base) return Boolean;
   overriding procedure Deactivate (E : in out Sys_Base);
```

(continues on next page)

(continued from previous page)

```
   not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;

private

   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
      record
         Active : Boolean;
      end record;

end Simple;
```

```
package body Simple is

   procedure Activate (E : in out Sys_Base) is
   begin
      --  NOTE: calling "E.Activation_Reset" does NOT dispatch!
      --        We need to use the 'Class attribute:
      Sys_Base'Class (E).Activation_Reset;

      E.Active := True;
   end Activate;

   function Is_Active (E : Sys_Base) return Boolean is
      (E.Active);

   procedure Deactivate (E : in out Sys_Base) is
   begin
      E.Active := False;
   end Deactivate;

end Simple;
```

```
package Simple.System_A is

   type A is new Sys_Base with private;

   overriding function Value (E : A) return Float;

private

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

   type A is new Sys_Base with record
      Val : Val_Array (1 .. 2);
   end record;

   overriding procedure Activation_Reset (E : in out A);

end Simple.System_A;
```

```
package body Simple.System_A is

   procedure Activation_Reset (E : in out A) is
   begin
      E.Val := (others => 0.0);
   end Activation_Reset;

   function Value (E : A) return Float is
      pragma Assert (E.Val'Length = 2);
```

(continues on next page)

(continued from previous page)

```
   begin
      return (E.Val (1) + E.Val (2)) / 2.0;
   end Value;

end Simple.System_A;
```

```
package Simple.System_B is

   type B is new Sys_Base with private;

   overriding function Value (E : B) return Float;

private

   type B is new Sys_Base with record
      Val : Float;
   end record;

   overriding procedure Activation_Reset (E : in out B);

end Simple.System_B;
```

```
package body Simple.System_B is

   procedure Activation_Reset (E : in out B) is
   begin
      E.Val := 0.0;
   end Activation_Reset;

   function Value (E : B) return Float is
     (E.Val);

end Simple.System_B;
```

```
with Ada.Finalization;

package Simple.System_AB is

   type AB is limited new Ada.Finalization.Limited_Controlled and
     Activation_IF and Value_Retrieval_IF with private;

   overriding procedure Activate (E : in out AB);
   overriding function Is_Active (E : AB) return Boolean;
   overriding procedure Deactivate (E : in out AB);

   overriding function Value (E : AB) return Float;

   not overriding function Check (E : AB) return Boolean;

private

   type Sys_Base_Class_Access is access Sys_Base'Class;
   type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;

   type AB is limited new Ada.Finalization.Limited_Controlled and
     Activation_IF and Value_Retrieval_IF with record
      S_Array : Sys_Base_Array (1 .. 2);
   end record;

   overriding procedure Initialize (E : in out AB);
```

(continues on next page)

(continued from previous page)

```
   overriding procedure Finalize   (E : in out AB);

end Simple.System_AB;
```

```
with Ada.Unchecked_Deallocation;

with Simple.System_A; use Simple.System_A;
with Simple.System_B; use Simple.System_B;

package body Simple.System_AB is

   overriding procedure Initialize (E : in out AB) is
   begin
      E.S_Array := (new A, new B);
   end Initialize;

   overriding procedure Finalize   (E : in out AB) is
      procedure Free is
        new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
   begin
      for S of E.S_Array loop
         Free (S);
      end loop;
   end Finalize;

   procedure Activate (E : in out AB) is
   begin
      for S of E.S_Array loop
         S.Activate;
      end loop;
   end Activate;

   function Is_Active (E : AB) return Boolean is
     (for all S of E.S_Array => S.Is_Active);

   procedure Deactivate (E : in out AB) is
   begin
      for S of E.S_Array loop
         S.Deactivate;
      end loop;
   end Deactivate;

   function Value (E : AB) return Float is
     ((E.S_Array (1).Value + E.S_Array (2).Value) / 2.0);

   function Check (E : AB) return Boolean is
      Threshold : constant := 0.1;
   begin
      return abs (E.S_Array (1).Value - E.S_Array (2).Value) < Threshold;
   end Check;

end Simple.System_AB;
```

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

with Simple.System_AB; use Simple.System_AB;

procedure Main is

   procedure Display_Active (E : AB) is
```

(continues on next page)

(continued from previous page)

```
   begin
      if Is_Active (E) then
         Put_Line ("System AB is active");
      else
         Put_Line ("System AB is not active");
      end if;
   end Display_Active;

   procedure Display_Check (E : AB) is
   begin
      if Check (E) then
         Put_Line ("System AB check: PASSED");
      else
         Put_Line ("System AB check: FAILED");
      end if;
   end Display_Check;

   S : AB;
begin
   Put_Line ("Activating system AB...");
   Activate (S);

   Display_Active (S);
   Display_Check (S);

   Put_Line ("Deactivating system AB...");
   Deactivate (S);

   Display_Active (S);
end Main;
```

Naturally, this is by no means the best possible implementation of system AB. By applying other software design strategies that we haven't covered here, we could most probably think of different ways to use object-oriented programming to improve this implementation. Also, in comparison to the *original implementation* (page 183), we recognize that the amount of source-code has grown. On the other hand, we now have a system that is factored nicely, and also more extensible.

BIBLIOGRAPHY

[Jorvik] A New Ravenscar-Based Profile by P. Rogers, J. Ruiz, T. Gingold and P. Bernardi, in Reliable Software Technologies — Ada Europe 2017, Springer-Verlag Lecture Notes in Computer Science, Number 10300.