John Barnes

With contributions by Ben Brosgol

Safe and Secure Software

An invitation to

UPDATED FOR **SPARK2014**

Courtesy of

www.adacore.com

First printed 2008. Reprinted with revisions 2009, 2013. 2015

V.20150501

Foreword

The aim of this booklet is to show how the study of Ada in general, and the features introduced by Ada 2005 and Ada 2012 in particular, can help anyone designing safe and secure software regardless of the programming language in which the software is eventually written. After all, successful implementers of safe and secure software write in the spirit of Ada in any language!

Thank you John for showing this throughout your papers, Ada rationales, books, and this booklet.

AdaCore dedicates this booklet to the memory of Dr. Jean Ichbiah (1940-2007), the principal designer of the original Ada language, who established the safe and secure foundations on which succeeding versions of the language have built.

Franco Gasperoni
Chief Executive Officer, AdaCore
Paris, January 2013

Preface

This revised version of the "Safe and Secure Software" booklet updates the content to take into account the important new facilities introduced in Ada 2012 which include support for contract-based programming. Ada 2012 marks the most significant advance in Ada since 1995 and is especially relevant for software that needs to meet safety and/or security certification standards.

I am very grateful for the assistance of Ben Brosgol of AdaCore in the preparation of the new content in this version of the booklet. Not only did Ben draft the new sections but he also ironed out several vague, misleading or plain incorrect bits in the original, and moreover has added a comprehensive index which I am sure will be of great value to all readers.

John Barnes
Caversham, England
January 2013

Since the 2013 edition of this booklet was published there has been a significant development relating to safe and secure Ada: the emergence of SPARK 2014. This major language revision, a subset of Ada 2012, is faithful to the original SPARK goal of facilitating formal analysis of program properties such as absence of run-time errors. But it also supports a variety of verification techniques including formal methods used in conjunction with traditional testing-based approaches. Accordingly the "Certified Safe with SPARK" chapter has been thoroughly updated to reflect SPARK 2014. We hope that this new chapter will inspire readers to take a further look at this exciting new language and technology.

John Barnes
Caversham, England
April 2015

Ben Brosgol
Bedford, Massachusetts USA
April 2015

Contents

Introduction

The aim of this booklet is to show how Ada – up to and including the Ada 2012 version of the language – addresses the needs of designers and implementers of safe and secure software. The discussion will also show that those aspects of Ada that make it ideal for safety-critical and security-critical application areas will also simplify the development of robust and reliable software in many other areas.

The world is becoming more and more concerned about both safety and security. Moreover, software now pervades all aspects of the workings of society. Accordingly, it is important that software for systems in which safety or security are a major requirement should be safe and secure.

There has been a long tradition of concern for safety going back to the development of railroad signaling and more recently with aviation, nuclear reactor control, and other areas in which a software flaw could lead to loss of human life or major environmental damage. Software in such domains has to meet well established certification criteria, for example DO-178B [1] (revised in December 2011 to DO-178C [2]) for airborne systems.

There is also a growing concern with security, not just in domains such as banking and communications where this issue would naturally be anticipated but also in safety-critical systems (automotive, avionics, medical devices, etc.) where networking can introduce vulnerabilities that might not have been possible earlier. This has been heightened with concern for the activities of terrorists.

Safety and security are intertwined through communication. An interesting characterization of the difference is

- *safety* – the software must not harm the world,
- *security* – the world must not harm the software.

So a safety-critical system is one in which the program must be *correct*, otherwise it might wrongly change some external device such as an aircraft flap or a railroad signal, with serious real-world consequences.

And a security-critical system is one in which it must not be possible for some incorrect or malicious input from the outside to violate the integrity of the system, for example by corrupting a password checking mechanism and stealing social security information.

The key to guarding against both problems is that the software must be correct in the aspects affecting the system's integrity. And by correct we mean that it meets its specification. Of course if the specification is incomplete or

itself incorrect then the system will be vulnerable. Capturing requirements correctly is a hard problem and is the focus of much attention from the software development community (as exemplified by the growing use of modelling languages and tools).

One of the trends of the second half of the twentieth century was a universal concern with freedom. But there are two aspects of freedom. The ability of the individual to do whatever they want conflicts with the right to be protected from the actions of others. Maybe A would like the freedom to smoke in a pub whereas B wants freedom from smoke in a pub. Concern with health in this example is changing the balance between these freedoms. Maybe the twenty-first century will see further shifts from "freedom to" to "freedom from".

In terms of software, the languages Ada and C have very different attitudes to freedom. Ada introduces restrictions and checks, with the goal of providing freedom from errors. On the other hand C gives the programmer more freedom, making it easier to make errors.

One of the historical guidelines in C was "trust the programmer". This would be fine were it not for the fact that programmers, like all humans, are frail and fallible beings. Experience shows that whatever techniques are used it is hard to write "correct" software. It is good advice therefore to use tools that can help by finding bugs and preventing bugs. Ada was specifically designed for this purpose. There have been four versions of Ada – Ada 83, Ada 95, Ada 2005, and now Ada 2012.

The purpose of this booklet is to illustrate the ways in which Ada, with a focus on Ada 2005 and Ada 2012, can help in the construction of reliable, safe, and secure software, by illustrating some aspects of its features. It is hoped that it will be of interest to programmers and managers at all levels.

It must be stressed that the discussion is not complete. Each chapter selects a particular topic under the banner of *Safe X* where *Safe* is just a brief token to designate both safety and security. For the most critical software, use of the related SPARK language appears to be very beneficial, and this is outlined in Chapter 11.

Topics with which Ada has much synergy are lean and agile software development – there is not enough space in this booklet to expand on these concepts but the reader is encouraged to explore their good ideas elsewhere.

As the twenty-first century progresses we will see software becoming even more pervasive. It would be nice to think that software in automobiles for example was developed with the same care as that in airplanes. But that is not so. My wife recently had an experience where her car displayed two warning icons. One said "stop at once", the other said "drive immediately to your dealer". Another anecdotal motor story is that of a driver attempting to select

channel 5 on the radio, only to see the car change into 5th gear! Luckily he did not try Replay.

For references to books and papers on Ada 2005, Ada 2012, SPARK, lean and agile software development, and related topics, please consult the bibliography.

1 Safe Syntax

Syntax is often considered to be a rather boring mechanical detail. The argument being that it is what you say that matters but not so much how it is said. That of course is not true. Clarity and unambiguity are important aids to any communication in a civilized world.

Similarly, a computer program is a communication between the writer and the reader, whether the reader be that awkward thing known as the compiler, or another team member, a reviewer or some other human soul. Indeed, most communication regarding a program is between two people. Clear and unambiguous syntax is a great help in aiding communication and, as we shall see, avoids a number of common errors.

An important aspect of good syntax design is that it is a worthwhile goal to try to ensure that typical simple typing errors cause the program to become illegal and thus fail to compile, rather than having an unintended meaning. Of course it is hard to prevent the accidental typing of X rather than Y or + rather than * but many structural risks can be prevented. Note incidentally that it is best to avoid short identifiers for just this reason. If we have a financial program about rates and times then using identifiers R and T is risky since we could easily type the wrong identifier by mistake (the letters are next to each other on the keyboard). But if the identifiers are Rate and Time then inadvertently typing Tate or Rime will be caught by the compiler. This applies to any language of course.

Equality and assignment

It is obvious that assignment and equality are different things. If we do an assignment then we change the state of some variable. On the other hand, equality is simply an operation to test some state. Changing state and testing state are very different things and understanding the distinction is important.

Many programming languages have confused these fundamentally different logical operations.

In Fortran, since its earliest days, one wrote

```
X = X + 1
```

But this is really rather peculiar. In mathematics x never equals $x + 1$. What the Fortran statement means of course is “replace the current value of X by the old value plus one”. But why misuse the equals sign in this way when society has

been cheerfully using the equals sign to mean equals for hundreds of years? (The equals sign dates from around 1550 when it was introduced by the English mathematician Robert Recorde.) The designers of Algol 60 recognized the problem and used the combination of a colon followed by an equals sign to mean assignment, thus

```
X := X + 1;
```

and this has the helpful consequence that the equals sign can unambiguously be used to mean equality, as in

```
if X = 0 then ...
```

The C language (like Fortran) adopted = for assignment and as a consequence C uses a double equals (==) to mean equality. This can cause much confusion.

Here is a fragment of a C program controlling the crossing gates on a railroad

```
if (the_signal == clear)
{
  open_gates( ... );
  start_train( ... );
}
```

The same program in Ada might be

```
if The_Signal = Clear then
  Open_Gates( ... );
  Start_Train( ... );
end if;
```

Now consider what happens if a programmer gets confused and accidentally forgets one of the equals signs in C thus

```
if (the_signal = clear)
{
  open_gates( ... );
  start_train( ... );
}
```

This still compiles but instead of just testing the_signal it actually assigns the value clear to the_signal. Moreover C unifies expressions (which have values) with assignments (which change state). So the assignment also acts as an expression and the result of the assignment is then used in the test. If the encoding is such that clear is not zero then the result will be true and so the gates are always opened, the_signal set to clear and the train started on its perilous journey. Conversely, if clear is encoded as zero, the test fails, the gates remain closed, and the train is blocked. In either case, things go badly wrong.

The pitfalls associated with the use of "=" for assignment and "==" for equality, and allowing assignments as expressions, are well known in the C community and have given rise to coding guidelines such as MISRA-C [3] and analysis tools such as lint. However it is preferable for such pitfalls to be avoided in the first place, through appropriate language design and that is how Ada has approached this issue

If the Ada programmer were to accidentally use an assignment in the test

```
if The_Signal := Clear then                -- illegal
```

then the program will simply fail to compile and all will be well.

Statement groups

It is often necessary to group a sequence of statements together – for example following a test using a keyword such as **if**. There are two typical ways of doing this

- by bracketing the group of statements so that they act as one (as in C),
- by closing the sequence with something matching the **if** (as in Ada).

These are also illustrated by the railroad example. The statements to open the gates and to start the train both need to be obeyed if the condition is true.

In C we had

```
if (the_signal == clear)
{
  open_gates( ... );
  start_train( ... );
}
```

and now suppose we inadvertently add a semicolon at the end of the first line (easily done). The program becomes

```
if (the_signal == clear) ;
{
  open_gates( ... );
  start_train( ... );
}
```

We now find that the condition is governing the null statement which is implicitly present between the test and the newly inserted semicolon. We cannot see it because a null statement is just nothing. So no matter what the state of the signal, the gates are always opened and the train set going.

In Ada the corresponding error would result in

```
if The_Signal = Clear then ;                -- illegal
   Open_Gates( ... );
   Start_Train( ... );
end if;
```

This is syntactically incorrect and so the error is safely caught by the compiler and the train wreck cannot occur.

Named notation

Another feature of Ada which is of a syntactic nature and can detect many unfortunate errors is the use of named associations in various situations. Dates provide a good illustration, because the order of the components varies according to local culture. Thus 12 January 2008 is written in Europe as 12/01/08 but in the US it is usually written as 01/12/08 (but not on the latest customs forms) whereas the ISO standard gives the year first, so would be 08/01/12.

In C we might declare a structure for manipulating dates as follows:

```
struct date {
   int day, month, year;
   } ;
```

which corresponds to the following type declaration in Ada

```
type Date is
   record
      Day, Month, Year: Integer;
   end record;
```

In C we might write

```
struct date today = {1, 12, 8};
```

But without looking at the type declaration we do not know whether this means 1 December 2008, 12 January 2008 or even 8 December 2001.

In Ada we have the option of writing

```
Today: Date := (Day => 1, Month => 12, Year => 08);
```

which uses named associations. Now it will be crystal clear if we ever write the values in the wrong order. (Note incidentally that Ada permits leading zeroes.).

We can also write the declaration as

```
Today: Date := (Month => 12, Day => 1, Year => 08);
```

which has the correct meaning and reveals the advantage that we do not need to remember the order in which the fields are declared.

Named associations can be used in other contexts in Ada as well. We might make similar errors with a function that has several parameters of the same type. Suppose we have a function to compute the obesity index of a person. The two parameters are the height and the weight which could be given as floating point values in pounds and inches (or kilograms and centimeters if you are metric). So we might have in C:

```
float index(float height, float weight) {
  ...
  return ... ;
}
```

or in Ada

```
function Index(Height, Weight: Float) return Float is
  ...
  return ... ;
end;
```

Now in the case of the author, the appropriate call of the index function in C might be

```
my_index = index(68.0, 168.0);
```

But if by mistake the call were reversed

```
my_index = index(168.0, 68.0);
```

then we would have a very thin and very tall giant! (It's a curious coincidence that both values end in 68.0 as well.)

Such an unhealthy disaster can be avoided in Ada by using named parameter calls thus

```
My_Index := Index(Height => 68.0, Weight => 168.0);
```

Again we can give the parameters in whatever order we wish and no error will occur if we forget the order in the declaration of the function.

Named notation is a very valuable feature of Ada. Its use is optional but it is well worth using freely since not only does it help to prevent errors but it also makes the program easier to understand.

Integer literals

Integer literals should not occur frequently in programs, apart from common values such as 0 and 1. Integer literals should mainly appear in initializations for constants. But when a literal does occur, its value should be obvious to the human reader. Intuitive notations for expressing the base (decimal, octal, hexadecimal, etc.) and for indicating groupings of digits will prevent errors.

Ada addresses both of these needs. It provides a clear syntax for numeric bases between 2 and 16 inclusive (10 is of course the default); for example `16#2B#` is an integer literal in base 16 with the value forty-three. In order to make large-magnitude literals more readable, Ada allows the use of an underscore symbol as separator between groups of digits. Thus the value of the integer literal `16#FFFF_FFFF_FFFF_FFFF#` is directly understandable as $2^{64}–1$

In contrast, the same literal in C (and in C++, Java, and other languages that have stayed with C-based syntax) would look like `0xFFFFFFFFFFFFFFFF` and it is easy to get eyestrain trying to figure out how many `F`s are present. Adding insult to injury, C interprets a leading 0 to mean octal, so a literal such as `031` does not mean what every schoolchild thinks it means, but rather has the value twenty-five.

2 Safe Typing

Safe typing is not about preventing heavy-handed use of the keyboard, although it can detect errors made by typos!

Safe typing is about designing the type structure of the language in order to prevent many common semantic errors. It is often known as strong typing.

Early languages such as Fortran and Algol treated all data as numeric types. Of course, at the end of the day, everything is indeed held in the computer as a numeric of some form, usually as an integer or floating point value and usually encoded using a binary representation. Later languages, starting with Pascal, began to recognize that there was merit in taking a more abstract view of the objects being manipulated. Even if they were ultimately integers, there was much benefit to be gained by treating colors as colors and not as integers by using enumeration types (just called scalar types in Pascal).

Ada takes this idea much further as we shall see, but other languages treat scalar types as just raw numeric types, and miss the critical idea of abstraction, which is to distinguish semantic intent from machine representation. The Ada approach provides more opportunities for detecting programming errors.

Using distinct types

Suppose we are monitoring some engineering production and checking for faulty items. We might count the number of good ones and bad ones. We want to stop production if the number of bad ones reaches some limit and perhaps also stop when the number of good ones reaches some other limit. In C or C++ we might have variables

```
int badcount, goodcount;
int b_limit, g_limit;
```

and then perhaps

```
badcount = badcount + 1;
...
if (badcount == b_limit) { ... };
```

and similarly for the good items. Since everything is really an integer, there is nothing to prevent us writing by mistake

```
if (goodcount == b_limit) { ... }
```

where we really should have written g_limit. Maybe it was a cut and paste error or a simple typo (g is next to b on a qwerty keyboard). Anyway, since they are integers the compiler will be happy even if we are not.

We could do the same in any language. But Ada gives us the opportunity to be more precise about what we are doing. We can write

```
type Goods is new Integer;
type Bads is new Integer;
```

These declarations introduce new types, which have all the properties of the predefined type Integer (such as operations + and –) and indeed are implemented in the same way, but are nevertheless distinct. We can now write

```
Good_Count, G_Limit: Goods;
Bad_Count, B_Limit: Bads;
```

and now we have quite distinct groups of entities for our manipulation; any accidental mixing will be detected by the compiler and prevent the incorrect program from running. So we can happily write

```
Bad_Count := Bad_Count + 1;

if Bad_Count = B_Limit then
```

but are prevented from writing

```
if Good_Count = B_Limit then          -- illegal
```

since this is a type mismatch.

If we did indeed want to mix the types, perhaps to compare the bad items and good items then we can do a type conversion (known as a cast in other languages) to make the types compatible. Thus we can write

```
if Good_Count = Goods(B_Limit) then
```

Another example might be when computing the difference between the counts of good and bad objects as an Integer:

```
Diff : Integer := Integer(Good_Count) – Integer(Bad_Count);
```

We can use the same technique to avoid accidental mixing of floating types. Thus when dealing with weights and heights in the chapter on Safe Syntax, rather than

```
My_Height, My_Weight: Float;
```

it would better to write

```
type Inches is new Float;
type Pounds is new Float;

My_Height: Inches := 68.0;
My_Weight: Pounds := 168.0;
```

and then confusion between the two would be detected by the compiler.

Enumerations and integers

In the chapter on Safe Syntax we discussed an example of a railroad crossing which included a test

```
if (the_signal == clear) { ... };
```

```
if The_Signal = Clear then ... end if;
```

in C and Ada respectively. In C the variable the_signal and associated constants such as clear might be declared thus

```
enum signal {
  danger,
  caution,
  clear
};

enum signal the_signal;
```

This convenient notation in fact is simply a shorthand for defining constants danger, caution and clear of type int. And the variable the_signal is also of type int.

As a consequence, nothing can prevent us from assigning a nonsensical value such as 4 to the_signal. In particular, such a nonsensical value might arise from the use of an uninitialized variable. Moreover, suppose other parts of the program are concerned with chemistry and use states anion and cation; nothing would prevent confusion between *cation* and *caution*. We might also be dealing with girls' names such as betty and clare or weapons such as dagger and spear. Nothing prevents confusion between *dagger* and *danger* or *clare* and *clear*.

In Ada we write

```
type Signal is (Danger, Caution, Clear);

The_Signal: Signal := Danger;
```

and no confusion can ever arise since an enumeration type in Ada truly is a different type and not a shorthand for an integer type. If we did also have

```
type Ions is (Anion, Cation);
type Names is (Anne, Betty, Clare, ... );
type Weapons is (Arrow, Bow, Dagger, Spear);
```

then the compiler would prevent the compilation of a program that mixed these things up. Moreover the compiler would prevent us from assigning to Clear or Danger since these are literals and this would be as nonsensical as trying to change the value of an integer literal such as 5 by writing

```
5 := 2 + 2;
```

At the machine level the various enumeration types are indeed encoded as integers and we can access the default encodings if we really need to, by using the attribute Pos thus

```
Danger_Code: Integer := Signal'Pos(Danger);
```

We can also specify our own encodings, as we shall see in the chapter on Safe Communication.

Incidentally, a very important built-in type in Ada is the type Boolean, which formally has the declaration

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

The result of a test such as The_Signal = Clear is of the type Boolean, and there are operations such as **and**, **or**, **not** which operate on Boolean values. It is never possible in Ada to treat an integer value as a Boolean or vice versa. In C it will be recalled, tests yield integer values and zero is treated as false, and nonzero as true. Again we see the danger in

```
if (the_signal == clear)
{
 ...
};
```

As mentioned earlier, omitting one equals turns the test into an assignment and because C permits an assignment to act as an expression the syntax is acceptable. The error is further compounded since the integer result is treated as a Boolean for the test. So altogether C has several pitfalls illustrated by the one example:

- using = for assignment,
- allowing assignments as expressions,
- treating integers as Booleans in conditional expressions.

Most of these flaws have been carried over into C++. None of these issues are present in Ada.

Constraints and subtypes

It is often the case that we know that the value of a certain variable is always going to be within some meaningful range. If so we should say so and thereby make explicit in the program some assumption about the external world. Thus My_Weight could never be negative and would hopefully never exceed 300 pounds. So we can declare

```
My_Weight: Float range 0.0 .. 300.0;
```

or if we had been methodical programmers and had previously declared a floating type Pounds then

```
My_Weight: Pounds range 0.0 .. 300.0;
```

If by mistake the program generates a value outside this range and then attempts to assign it to My_Weight thus

```
My_Weight := Compute_Weight( ... );
```

then the exception Constraint_Error will be raised (or thrown) at run time. We might handle (or catch) this exception in some other part of the program and take remedial action. If we do not, the program will stop and the run-time system will produce an error message indicating where the violation occurred. This all happens automatically – appropriate checks are inserted into the compiled code. (The careful reader who is familiar with the concurrency features in Ada will note that our statement "the program will stop" requires qualification: it applies to sequential programs. The situation with concurrent programs is somewhat different but the topic is outside the scope of this chapter.)

This idea of subranges was first introduced in Pascal and improved in Ada. It is not available in most other languages and we would have to program our own checks all over the place but more likely we wouldn't bother, and any error resulting from violating these bounds would be that much harder to detect.

If we knew that every weight to be dealt with by the program was in a restricted range, then rather than putting a constraint on every variable declaration we can impose it on the type Pounds in the first place.

```
type Pounds is new Float range 0.0 .. 300.0;
```

On the other hand if some weights in the program are unrestricted and it is only the weight of people that are known to lie in a restricted range then we can write

```
type Pounds is new Float;
subtype People_Pounds is Pounds range 0.0 .. 300.0;

My_Weight: People_Pounds;
```

We can also apply constraints and declare subtypes of integer types and enumeration types. Thus when counting good items we would assume that the number was never negative and perhaps that it would never exceed 1000. So we might have

```
type Goods is new Integer range 0 .. 1000;
```

If we just wanted to ensure that it was never negative but did not wish to impose an upper limit then we could write

```
type Goods is new Integer range 0 .. Integer'Last;
```

where Integer'Last gives the upper value of the type Integer. The restriction to positive or nonnegative values is so common that the Ada language provides the following built-in subtypes:

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

The type Goods could then be declared as

```
type Goods is new Natural;
```

and this would just impose the lower limit of zero as required.

As an example of a constraint with an enumeration type we might have

```
type Day is (Monday, Tuesday, Wednesday, Thursday, Friday,
                    Saturday, Sunday);
subtype Weekday is Day range Monday .. Friday;
```

and then a run-time check will prevent assigning Sunday to a variable of the subtype Weekday.

Inserting constraints as in the above examples may seem to be tiresome but makes the program clearer. Moreover, it enables the compiler and run-time system to verify that the assumptions being expressed by the constraints are indeed correct.

Subtype predicates

The subtype feature of Ada is very valuable and enables the early detection of errors that linger in many programs in other languages and cause disaster later. However, although valuable, the subtype mechanism is somewhat limited. We can only specify a contiguous range of values in the case of integer and enumeration types.

Accordingly, Ada 2012 has introduced *subtype predicates* that can be applied to type and subtype declarations. The requirements proved awkward to satisfy with a single feature so in fact there are two, depending on whether the predicate is static or dynamic. They both take a Boolean expression and the key difference is that the static predicate is restricted to certain types of expressions so that it can be used in more contexts.

Suppose we are concerned with seasons and that we have a type Month thus

```
type Month is (Jan, Feb, Mar, Apr, May, Jun,
               Jul, Aug, Sep, Oct, Nov, Dec);
```

Now suppose we wish to declare subtypes for the seasons. For northern hemispherians winter is December, January, February. (From the point of view of solstices and equinoxes, winter is from December 21 until March 21 or thereabouts, but March seems to me generally more like spring rather than winter and December feels more like winter than autumn.) So we would like to declare a subtype embracing Dec, Jan and Feb. We cannot do this with a constraint but we can use a static predicate by writing

```
subtype Winter is Month
  with Static_Predicate => Winter in Dec | Jan | Feb;
```

and then we are assured that objects of subtype Winter can only be Dec, Jan or Feb. Note the use of the subtype name (Winter) in the expression where it stands for the current instance of the subtype.

This usage of the *with* syntax was introduced in Ada 2012 and is known as an *aspect specification*. Aspects are used in Ada 2012 for specifying a variety of program properties, some of which can also be defined by pragmas or representation clauses.

The Static_Predicate aspect is checked whenever an object is default initialized, on assignments, on conversions, on parameter passing and so on. If a check fails then Assertion_Error is raised. (Whether subtype predicate checking is performed is controlled by the Assertion_Policy pragma; it is enabled by specifying the pragma's argument as Check.)

If we want the expression to be dynamic then we have to specify the Dynamic_Predicate aspect thus

```
type T is ... ;
function Is_Good(X: T) return Boolean;

subtype Good_T is T
  with Dynamic_Predicate => Is_Good(Good_T):
```

Note that a subtype with predicates cannot be used in some contexts such as index constraints. This is to avoid having arrays with holes and similar nasty things. However, static predicates are allowed in a **for** loop meaning to try every value. So we could write

```
for M in Winter loop...
```

The loop uses values for M in the order, Jan, Feb, Dec, which is the same as the order in the declaration of the type Month itself.

Arrays and constraints

An array is an indexable set of things. As a simple example, suppose we are playing with a pair of dice and wish to record how many throws of each value (from 2 to 12) have been obtained. Since there are 11 possible values, in C we might write

```
int counters[11];

int throw;
```

and this will in fact declare 11 variables referred to as counters[0] to counters[10] and a single integer variable throw.

If we wish to record the result of another throw then we might write:

```
throw = ... ;

counters[throw–2] = counters[throw–2] + 1;
```

Note the need to decrement the throw value by 2, since C arrays are always zero-indexed (that is, have a lower bound of zero). Now suppose the counting mechanism goes wrong (some joker produces a die with 7 spots perhaps or maybe we are generating the throws using a random number generator and we have not programmed it correctly) and a throw of 13 is generated. What happens? The C program does not detect the error but simply computes where counters[11] would be and adds one to that location. Most likely this will be the location of the variable throw itself since it is declared after the array and it will become 14! The program just goes hopelessly wrong.

This is an example of the infamous buffer overflow problem. It is at the heart of many serious and hard-to-detect programming problems. It is ultimately a

gaping loophole which permits viruses to attack systems such as Windows. This is discussed further in Chapter 7 on Safe Memory Management.

Now consider the same program in Ada, we can write

```
Counters: array (2 .. 12) of Integer;
Throw: Integer;
```

and then

```
Throw := ... ;
Counters(Throw) := Counters(Throw) + 1;
```

And now if Throw has a rogue value such as 13 then since Ada has run-time checks to ensure that we cannot read or write to a part of an array that does not exist, the exception Constraint_Error is raised and the program is prevented from running wild.

Note that Ada gives control over the lower bound of the array as well as the upper bound. Array indices in Ada do not all start at zero. Lower bounds in real programs are more often one than zero. Specifying the lower bound as 2 in the above example means that the variable throw can be used directly in the index, without the complication of deciding on and subtracting the appropriate offset as in the C version.

The problem with the dice program was not so much that the upper bound of the array was exceeded (that was the symptom) but rather that the value in Throw was out of bounds. We can catch the mistake earlier by declaring a constraint on Throw thus

```
Throw: Integer range 2 .. 12;
```

and now Constraint_Error is raised when we try to assign 13 to Throw. As a consequence the compiler is able to deduce that Throw always has a value appropriate to the range of the array, and no checks will actually be necessary for accessing the array using Throw as an index. Indeed, placing a constraint on variables used for indexing typically reduces the number of run-time checks overall. Incidentally, we can reduce the double appearance of the range 2 .. 12 by writing

```
subtype Dice_Range is Integer range 2 .. 12;
Throw: Dice_Range;
Counters: array (Dice_Range) of Integer;
```

The advantage of only writing the range once is that if we need to change the program (perhaps adding a third die so that the range becomes 3 .. 18) then this only has to be done in one place.

Range checks in Ada are of enormous practical benefit during testing and can be turned off for a production program. Ada compilers are not unique in applying run-time checks in programs. The Whetstone Algol 60 compiler dating from 1962 did it. Ada (like Java and C#) specifies the checks in the language definition itself.

Perhaps it should also be mentioned that we can give names to array types as well. If we had several sets of counter values then it would be better to write

```
type Counter_Array is array (Dice_Range) of Integer;
Counters: Counter_Array;
Old_Counters: Counter_Array;
```

and then if we wanted to copy all the elements of the array Counters into the corresponding elements of the array Old_Counters then we simply write

```
Old_Counters := Counters;
```

Giving names to array types is not possible in many languages. The advantage of naming types is that it introduces *explicit* abstractions, as when counting the good and bad items. By telling the compiler more about what we are doing, we provide it with more opportunities to check that our program makes sense.

All objects of the type Counter_Array have the same number of components as give by the subtype Dice_Range. Accordingly, the type is called a *constrained array type*. Sometimes it is convenient to introduce a more flexible type which embraces objects with the same index and component type but with different numbers of components. Consider

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

The type Float_Array is known as an *unconstrained array type*. When an object of this type is declared, the upper and lower bounds have to be supplied either as a constraint or from the initial value. Thus we can write

```
An_Array: Float_Array(1 .. N);
```

The inquisitive reader may wonder what happens when the upper bound is less than the lower bound; for example, suppose N has the value 0 This is permitted in Ada and is referred to as a *null array*. Interestingly, the upper bound is thus allowed to be less than the lower bound of the index subtype.

Unconstrained array types are very useful as parameters since they enable us to write subprograms that manipulate arrays of any size. We will see examples of this later.

Default initialization

The assurance given by subtype predicates (and by type invariants as we will see later) can depend upon the object having a sensible initial value. The original Ada design provided a partial solution to this issue. Values of access types ("pointers") are guaranteed a default initialization to a special value **null**, and the programmer can define default initializations for record components as in

```
type Font is (Arial, Bookman, Times_New_Roman);
type Size is range 1..100;

type Formatted_Character is
  record
    C: Character;
    F: Font := Times_New_Roman;
    S: Size := 12;
  end record;

FC: Formatted_Character;
  -- Now FC.F = Times_New_Roman, FC.S = 12,
  -- FC.C is uninitialized
```

Default initialization is somewhat controversial. There is a school of thought that giving default initial values (such as zero) is bad since it can obscure flow errors. A counterargument is that it can also ensure that objects have consistent initial state, which can help prevent certain kinds of vulnerabilities.

In any event, it is strange that early versions of Ada did allow default initial values to be given for components of records but not for scalar types or array types. This is rectified in Ada 2012 by aspects Default_Value and Default_Component_Value. Here's an alternative version of the scalar types shown above:

```
type Font is (Arial, Bookman, Times_New_Roman)
   with Default_Value => Times_New_Roman;

type Size is range 1..100
   with Default_Value => 12;
```

With these declarations we can define Formatted_Character without needing to provide default values for the components F and S

```
type Formatted_Character is
  record
    C: Character;
```

```
    F: Font; -- Times_New_Roman by default
    S: Size; -- 12 by default
  end record;
```

We can also set a default value for an array component as in

```
type Text is new String
  with Default_Component_Value =>
         Ada.Characters.Latin_1.Space;
```

Note that, unlike default initial values for record components, these have to be static.

Real errors

The title of this section is an example of those nasty puns so hated by the software pioneer Christopher Strachey as mentioned in the Conclusion. This is about accuracy in arithmetic and in particular with real as opposed to integer types.

In floating point arithmetic (using types such as real in Pascal, float in C and Float in Ada) the computation is done with the underlying floating point hardware. Floating point numbers have a relative accuracy. A 32-bit word might allocate 23 bits for the mantissa, one bit for the sign and 8 bits for the exponent. This gives an accuracy of 23 binary digits or about 7 decimal digits.

So a large value such as 123456.7 is accurate to one decimal place, whereas a very small value such as 0.01234567 is accurate to eight decimal places, but in all cases the number of significant digits is always 7. So the accuracy is relative to the magnitude of the number.

Relative accuracy works well most of the time but not always. Consider the representation of an angle giving the bearing of a ship or rocket. Perhaps we would like to hold the accuracy to a second of arc. Remember that there are 60 seconds in a minute, 60 minutes in a degree and 360 degrees in a whole circle.

If we hold the angle as a floating point number

```
float bearing;
```

then the accuracy at 360 degrees will be about 8 seconds which is not good enough, whereas the accuracy at 1 degree will be about 1/45 second which is unnecessary.

We could of course hold the value as an integral number of seconds by using an integer type

```
int bearingsecs;
```

This works but it means we have to remember to do our own scaling for input and display purposes.

But the real trouble with floating point is that the accuracy of operations such as addition and subtraction is affected by rounding errors. If we subtract two nearly equal values then we get cancellation errors. And of course certain numbers will not be held exactly. If we have a stepping motor which works in 1/10 degree steps then because 0.1 cannot be held exactly in binary the result of adding 10 steps will not be exactly one degree at all. So even if the accuracy required is quite coarse so that the notional accuracy is more than adequate the cumulative effect of tiny computational errors can be unbounded.

Scaling everything to use integers is acceptable for simple applications but when we have several types held as scaled integers and we have to operate on several together we often get into problems and have to do our own scaling (perhaps even by using raw machine operations such as shifting). This is all prone to errors and difficult to maintain.

Ada is one of the few languages to provide fixed point arithmetic. This does the scaling automatically for us. Thus for the stepping motor we might declare

```
type Angle is delta 0.1 range –360.0 .. 360.0;
for Angle'Small use 0.1;
```

and this will hold the values internally as scaled integers that represent multiples of 0.1 but we can think about them as the abstract values they represent, that is degrees and tenths of degrees. And our arithmetic operations will not suffer from rounding errors.

In summary, Ada has two forms of real arithmetic

- floating point, which provides relative accuracy,
- fixed point, which provides absolute accuracy.

Ada also supplies a specialized form of fixed point for decimal arithmetic, which is the standard model for financial calculations.

The topic of this section is rather specialized but it does illustrate the breadth of facilities in Ada and the care taken to encourage safety in numerical calculations.

3 Safe Pointers

Primitive man made a huge leap forward with the discovery of fire. Not only did this allow him to keep warm and cook and thereby expand into more challenging environments but it also enabled the creation of metal tools and thus the bootstrap to an industrial society. But fire is dangerous when misused and can cause tremendous havoc; observe that society has special standing organizations just to deal with fires that are out of control.

Software similarly made a big leap forward in its capabilities when the notion of pointers or references was introduced. But playing with pointers is like playing with fire. Pointers can bring enormous benefits but if misused can bring immediate disaster such as a blue screen, or allow a rampaging program to destroy data, or create the loophole through which a virus can invade.

High integrity software typically limits drastically the use of pointers. The access types of Ada have the semantics of pointers but in addition carry numerous safeguards on their use, which makes them suitable for all but the most demanding safety-critical programs.

References, pointers and addresses

Pointers introduce several opportunities for programming errors such as

- *Type safety violations* – creating an object of one type and then accessing it (through a pointer) as though it were of some other type. Or, more generally, using a pointer to access an object in a manner that is inconsistent with some of the object's semantic properties (for example, assigning to a constant or violating a range constraint).
- *Dangling references* – accessing an object through a pointer after the object has been freed; either a local variable that has gone out of scope, or a dynamically allocated object that has been explicitly freed through some other pointer.
- *Storage exhaustion* – failing to allocate an object, because of the unavailability of sufficient space. This may be caused by a number of factors:
    - Allocating objects that later become inaccessible ("garbage") but which are never freed;
    - Heap fragmentation, where there may be sufficient total space for a given allocation but not enough contiguous space;

- Heap size underestimation;
- Storage leakage (allocating accessible objects *ad infinitum*, for example continually adding elements onto a linked list)

Although the details are different, type safety violations and dangling references may similarly arise if the language allows pointers to subprograms.

Historically, languages have taken different approaches to these problems. Early languages such as Fortran, COBOL and Algol 60 did not have a notion of pointers at the level of the user program. Programs in all languages use addresses for basic operations such as calling a subprogram, but addresses in these languages cannot be directly manipulated by the user.

C (and C++) permit pointers to both heap-allocated and declared (stack-allocated) objects, and also to functions. Although these languages offer some checks, it is basically the programmer's responsibility to use pointers correctly. For example, since C treats an array as a pointer to its initial element, and allows pointer arithmetic as the equivalent of array indexing, all the necessary low-level ingredients are provided that can get programmers into trouble.

Java and other "pure" object-oriented languages do not expose pointers to the application but rely on pointers and dynamic allocation as the basis of the language semantics. Type checking is preserved, dangling references are prevented (there is no explicit "free"), but to prevent inaccessible objects from cluttering up the heap the implementation has to provide automatic storage reclamation (garbage collection). This is a reasonable approach for certain kinds of programs. It is still a questionable technology for real-time applications, especially ones with safety-critical or security-critical requirements.

Note also that garbage collection does not by itself prevent storage leaks: a program that adds objects onto a linked list in an infinite loop will eventually exhaust the heap despite the most heroic efforts of a garbage collector. (Infinite loops are not necessarily program bugs; process control and similar applications are often written as non-terminating programs, requiring an external action such as an operator pressing a reset button in order to halt the process.)

The history of Ada with respect to pointers is interesting. The original version of the language, Ada 83, provided pointers only for dynamic allocation (thus no pointers to declared objects, no pointers to subprograms) and also supplied an explicit free operation known as Unchecked_Deallocation. This preserved type safety, and avoided dangling references caused by pointers to out-of-scope local variables, but introduced the possibility of dangling references through incorrect uses of Unchecked_Deallocation.

The decision to include Unchecked_Deallocation was unavoidable, since the only alternative – requiring implementations to supply garbage collection – was not an appropriate option given Ada's intended domain of real-time and high-integrity systems. However, the Ada philosophy is that if a feature defeats

checks that are normally performed, then its use must be explicit. And indeed, if we are using Unchecked_Deallocation we need to "with" and then instantiate a generic procedure. (The concepts of a *with clause* and *generic instantiation* are explained in the next chapter.) This somewhat heavyweight syntax both prevents accidental usage and makes our intent clear to whoever needs to read or maintain our code.

Ada 95 extended the Ada 83 mechanism, allowing pointers to declared objects and also to subprograms. Ada 2005 has taken things a bit further – for example, making it easier to pass (pointers to) subprograms as run-time parameters. How these were accomplished without sacrificing safety will be the subject of this chapter.

A final note before going into further detail. Perhaps because pointers and references have a hardware-level connotation, Ada uses the term *access types*. This enforces the view that values of an access type give access to other objects of some designated type (are like dynamic names for these objects) and should not be thought of as simply machine addresses. Indeed, at the implementation level, the representation of an access value might be different from a physical pointer.

Access types and strong typing

Using a feature introduced by Ada 2005, we can declare a variable Ref whose values give access to objects of type T:

```
Ref: access T;
```

If we do not give an initial value then a special value **null** is assumed. Ref can refer to a normal declared object of type T (which must be marked **aliased**) by

```
Obj: aliased T;
...
Ref := Obj'Access;
```

The analogous C version is:

```
t* ref;
t obj;
ref = &obj;
```

T might be a record type such as

```
type Date is
  record
    Day: Integer range 1 .. 31;
    Month: Integer range 1 .. 12;
    Year: Integer;
end record;
```

so we might have

```
Birthday: aliased Date := (Day => 10, Month => 12, Year => 1815);
AD: access Date := Birthday'Access;
```

and then to retrieve the individual components of the date referred to indirectly by AD we can write for example

```
The_Day: Integer := AD.Day;
```

A variable such as AD can also refer to an object dynamically allocated on the heap (called a *storage pool* in Ada). We can write

```
AD := new Date'(Day => 27, Month => 11, Year => 1852);
```

(The two dates are those of the birth and death of Ada, Countess of Lovelace after whom the language is named.)

A common application of access types is to create linked lists – we might declare

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
```

and then we can create chains of objects of the type Cell linked together.

It is often convenient to give a name to an access type

```
type Date_Ptr is access all Date;
```

The "**all**" in the syntax indicates that this named type can refer to both objects on the heap and also to those declared locally on the stack that are marked as **aliased**.

Having to mark objects as **aliased** is a useful safeguard. It alerts the programmer to the fact that the object might be referred to indirectly (good for walkthrough reviews) and it also warns the compiler that any optimizations need to take heed of the possibility of multiple and indirect accesses.

But the key point is that an access type always identifies the type of the object that its values refer to and strong typing is enforced on assignments, parameter passing, and all other uses. Moreover, an access value always has a legitimate value (which could be **null**). At run time, whenever we attempt to access an object referred to by an object of the type Date_Ptr, there is a check to ensure that the value is not null – the exception Constraint_Error is raised if this check fails.

We can explicitly state that an access value cannot be null by declaring it as follows (this syntax was introduced in Ada 2005):

```
WD: not null access Date := Wedding_Day'Access;
```

and then of course it must be given an initial value which is not null. The advantage of a so-called *null exclusion* is that we are guaranteed that an exception cannot occur when accessing the indirect object.

Finally, note that an access value can denote a component of a composite structure, provided the component type is marked as aliased. For example

```
A: array (1 .. 10) of aliased Integer := (1,2,3,4,5,6,7,8,9,10);
P: access Integer := A(4)'Access;
```

But we cannot perform any incremental operations on P such as P++ or P+1 to make it refer to A(5) as can be done in C. (Indeed, the ++ operator is not even part of the Ada language.) This sort of thing in C is prone to errors since nothing prevents us from pointing beyond either end of the array.

Access types and accessibility

We have just seen that the strong typing of Ada ensures that an access value can never refer to an object of the wrong type. The other requirement is to ensure that the object referred to cannot cease to exist while access objects still refer to it. This is achieved for declared objects through the notion of accessibility. Consider

```
package Data is
  type Int_Ref is access all Integer;
  Ref1: Int_Ref;
end Data;

with Data; use Data;
```

```
procedure P is
   K: aliased Integer;
   Ref2: Int_Ref;
begin
   Ref2 := K'Access;                -- illegal

   Ref1 := Ref2;
   ...
end P;
```

This is clearly a very artificial example but illustrates the key points in a small space. The package Data has an access type Int_Ref and an object of that type called Ref1. The procedure P declares a local variable K and a local access variable Ref2 also of the type Int_Ref and attempts to assign an access to K to the variable Ref2. This is forbidden. The problem is not with the assignment to Ref2 – both Ref2 and K will cease to exist when we return from a call of the procedure P. The danger is that we might assign the value in Ref2 to a global variable, as we do here with Ref1, which would then contain a reference to K that would be usable after K had ceased to exist.

The basic rule is that the lifetime of the accessed object (such as K) must be at least as long as the lifetime of the specified access type (in this case Int_Ref). Here it is not and so the attempt to obtain a pointer to K is illegal.

The rules are phrased in terms of accessibility levels (how deeply nested the declaration of something is) and are mostly static, that is to say checked by the compiler; they incur no cost at run time. But the rules concerning parameters of subprograms that are of anonymous access types are dynamic (that is, require run-time checks). This gives more programming flexibility than would otherwise be possible.

In this short introduction to Ada it is not feasible to go into further details. Suffice it to say that the accessibility rules of Ada prevent dangling references to declared objects, which can be a source of many subtle and hard-to-diagnose errors in lax languages.

References to subprograms

Ada permits references to procedures and functions to be manipulated in a similar way to references to objects. Both strong typing and accessibility rules apply. For example, using a feature introduced in Ada 2005, we can write

```
A_Func: access function (X: Float) return Float;
```

and A_Func is then an object that can only refer to functions that take a parameter of the type Float and return a value of type Float (such as the predefined function Sqrt).

So we can write

```
A_Func := Sqrt'Access;
```

and then

```
X: Float := A_Func(4.0);                    -- indirect call
```

and this will call Sqrt with argument 4.0 and hopefully produce 2.0.

Ada thoroughly checks that the parameters and result always match properly and so we cannot call a function indirectly that has the wrong number or types of parameters. The parameter list and result type constitute what is technically called the *profile* of the function.

Thus consider the predefined function Arctan (the inverse tangent). It takes two parameters

```
function Arctan(Y: Float; X: Float) return Float;
```

and returns the angle θ (in radians) such that tan $\theta = Y/X$. If we attempt to write

```
A_Func := Arctan'Access;                    -- illegal
Z := A_Func(A);                             -- indirect call prevented
```

then the compiler rejects the code because the profile of Arctan does not match that of A_Func. This is just as well because otherwise the function Arctan would read two items from the run-time stack whereas the indirect call via A_Func placed only one parameter on the stack. This would result in the computation becoming meaningless.

Corresponding checks in Ada occur also across compilation unit boundaries (compilation units are units that can be compiled separately, as explained in the chapter on Safe Architecture). Equivalent mismatches are not prevented in C and this is a common cause of serious errors.

More complex situations arise because a subprogram can have another subprogram as a parameter. Thus we might have a function whose purpose is to solve an equation $Fn(x) = 0$ where the function Fn is itself passed as a parameter. Thus

```
function Solve(Trial: Float; Accuracy: Float;
               Fn: access function (X: Float) return Float)
    return Float;
```

The parameter Trial is the initial guess, the parameter Accuracy is the accuracy required and the third parameter Fn identifies the equation to be solved.

As an example suppose we invest 1000 dollars today and 500 dollars in a year's time: what would the interest rate have to be for the final value two years from now to be exactly 2000 dollars? If the interest rate is x% then the Net Final Value (Nfv) will be given by

$$Nfv(x) = 1000 \times (1 + x/100)^2 + 500 \times (1 + x/100)$$

We can answer the question by declaring the following function, which returns 0.0 when X is such that the net final value is precisely 2000.0.

```
function Nfv_2000 (X: Float) return Float is
  Factor: constant Float := 1.0 + X/100.0;
begin
  return 1000.0 * Factor**2 + 500.0 * Factor – 2000.0;
end Nfv_2000;
```

We can then write:

```
Answer: Float :=
     Solve (Trial => 5.0, Accuracy => 0.01, Fn => Nfv_2000'Access);
```

We are guessing that the answer might be around 5%, we want the answer with 2 decimal figures of accuracy and of course Nfv'Access identifies the problem. The reader is invited to estimate the interest rate – the answer is at the end of this chapter. (Note that terms such as Net Final Value and Net Present Worth are standard terms used by financial professionals.)

The point of this discussion is to emphasize that Ada checks the matching of the parameters of the function parameter as well. Indeed, the nesting of profiles can continue to any degree and Ada matches all levels thoroughly. Many languages give up after one level.

Note that the parameter Fn was actually of an anonymous type. Access to subprogram types can be named or anonymous just like access to object types. They can also have a null exclusion. Thus (using features introduced in Ada 2005) we should really have written

```
A_Func: not null access function (X: Float) return Float := Sqrt'Access;
```

The advantage of using a null exclusion is that we are guaranteed that the value of A_Func is not null when the function is called indirectly.

If it seems that having to initialize it, perhaps arbitrarily, to Sqrt'Access is distasteful then we could always declare

```
function Default(X: Float) return Float is
begin
  Put("Value not set");  return 0.0;
end Default;
...
A_Func: not null access function (X: Float) return Float := Default'Access;
```

Similarly we should really add **not null** to the profile in Solve thus

```
function Solve(Trial: Float; Accuracy: Float;
     Fn: not null access function (X: Float) return Float) return Float;
```

This ensures that the actual function corresponding to Fn cannot be null.

Nested subprograms as parameters

We mentioned that accessibility rules also apply to access-to-subprogram values. Suppose we had declared Solve so that the parameter Fn was of a named type and that it and Solve are in some package

```
package Algorithms is
  type A_Function is not null access function (X: Float) return Float;

  function Solve(Trial: Float; Accuracy: Float; Fn: A_Function)
                                                          return Float;
  ...
end Algorithms;
```

Suppose we now decide to express the interest example with the target value passed as a parameter. We might try

```
with Algorithms;  use Algorithms;
function Compute_Interest(Target: Float) return Float is

  function Nfv_T (X: Float) return Float is
    Factor: constant Float := 1.0 + X/100.0;
  begin
    return 1000.0 * Factor**2 + 500.0 * Factor – Target;
  end Nfv_T;

begin
  return Solve(Trial => 5.0, Accuracy => 0.01, Fn => Nfv_T'Access);
                                                          -- illegal
end Compute_Interest;
```

However, Nfv_T'Access is not allowed as the Fn parameter because it violates the accessibility rules. The trouble is that the function Nfv_T is at an inner level with respect to the type A_Function. (It has to be in order to get hold of the parameter Target.) If Nfv_T'Access had been allowed then we could have assigned this value to a global variable of the type A_Function so that when Compute_Interest had returned we would have still had a reference to Nfv_T even after it had ceased to be accessible. For example

```
Dodgy_Fn: A_Function := Default'Access;          -- a global variable

function Compute_Interest(Target: Float) return Float is

  function Nfv_T(X: Float) return Float is
    ...
  end Nfv_T;

begin
  Dodgy_Fn := Nfv_T'Access;    -- illegal
  ...
end Compute_Interest;
```

and now suppose that after a call of Compute_Interest we execute:

```
Answer := Dodgy_Fn(99.9);       -- would have unpredictable results
```

The call of Dodgy_Fn would attempt to call Nfv_T but that is no longer possible since it is local to Compute_Interest and would attempt to access the parameter Target which no longer exists. The consequences would be unpredictable (a meaningless result, or perhaps an exception would be raised) if Ada did not prevent it. Note that using an anonymous type for the parameter as in the previous section allows passing the nested function as a parameter, but the accessibility checks prevent the assignment to Dogdy_Fn. A run-time check would detect that Nfv_T is more deeply nested than the target access type A_Function, and a Program_Error exception would be raised. So the solution is just to change the package Algorithms thus

```
package Algorithms is
  function Solve(Trial: Float; Accuracy: Float;
                 Fn: not null access function (X: Float) return Float)
                                                          return Float;
end Algorithms;
```

and the original function Compute_Interest is now exactly as before (except that the comment -- *illegal* needs to be removed).

Those of a mischievous mind might suggest that the problem lies with nesting Nfv_T inside Compute_Interest. It would indeed be possible to declare Nfv_T at the outermost level so that no accessibility problem arises, but then the

value Target would have to be passed globally through some package – in the style of Fortran Common blocks. We cannot add it as an additional parameter to Nfv_T because the parameters of Nfv_T must match those of Fn. But passing data globally in this way is in fact bad practice. It violates principles of information hiding and abstraction and does not work at all in a multitasking program. Note that the practice of nesting a function within another, where the inner function uses non-local variables (such as Target) is often called a "downward closure".

Downward closures, that is to say passing a pointer to a nested subprogram as a run-time parameter, are used in several parts of the Ada predefined library, for applications such as iterating over a data structure.

The nesting of subprograms is a natural requirement for these applications because of the need to pass non-local information. This is harder to do in flat languages such as C, C++ and Java. Although type extensions can be used in some languages to model subprogram nesting, this mechanism is less clear and can be a problem for program maintenance.

Finally, some applications need to combine (invoke) algorithms in a nested manner. Thus we might have other useful stuff in the package Algorithms

```
package Algorithms is

  function Solve(Trial: Float; Accuracy: Float;
                 Fn: not null access function (X: Float) return Float)
                                                          return Float;
  function Integrate (Lo, Hi: Float; Accuracy: Float;
                 Fn: not null access function (X: Float) return Float)
                                                          return Float;
  type Vector is array (Positive range <>) of Float;

  procedure Minimize(V: in out Vector; Accuracy: Float;
        Fn: not null access function (V: Vector) return Float);

end Algorithms;
```

The function Integrate is similar to Solve. It computes the definite integral of the function parameter, between the given limits. The procedure Minimize is a little different. It finds those values of the elements of the array V which make the value of the function parameter a minimum. We might have a situation where a cost function is to be minimized and is itself the result of doing an integration and that the values of V are used in the integration (this might seem rather unlikely but the author spent the first few years of his programming life doing just this sort of thing in the chemical industry).

The structure could be

```
with Algorithms;  use Algorithms;
procedure Do_It is

  function Cost(V: Vector) return Float is

    function F(X: Float) return Float is
      Result: Float;
    begin
      ...           -- compute Result using V as well as X
      return Result;
    end F;

  begin
    return Integrate(0.0, 1.0, 0.01, F'Access);
  end Cost;

  A: Vector(1 .. 10);
begin

  ...               -- perhaps read in or set trial values for the vector A

  Minimize(A, 0.01, Cost'Access);

  ...               -- output final values of the vector A.
end Do_It;
```

This all works like a dream in Ada 2005 (and of course also in Ada 2012) – just as it did in Algol 60. In other programming languages this is either difficult or requires the use of unsafe constructs with potentially dangling references.

Further examples of the use of access to subprogram types will be found in the chapter on Safe Communication.

Finally, the interest rate that turns the investment of 1000 dollars and 500 dollars into 2000 dollars in two years is about 18.6%. Nice rate if you can get it.

4 Safe Architecture

When speaking of buildings, a good architecture is one whose design gives the required strength in a natural and unobtrusive manner and thereby provides a safe environment for the people within. An elegant example is the Pantheon in Rome whose spherical shape has enormous strength and provides an uncluttered space. Many ancient cathedrals are not so successful, and need buttresses tacked on the outside to prop up the walls. In 1624, Sir Henry Wooton summed the matter up in his book, *The Elements of Architecture*, by saying "Well building hath three conditions – commoditie, firmenes & delight". In modern terms, it should work, be strong and be beautiful as well.

A good architecture in a program should similarly provide unobtrusive safety for the detailed workings of the inner parts within a clean framework. It should permit interaction where appropriate and prevent unrelated activities from accidentally interfering with each other. And a good language should enable the writing of aesthetically pleasing programs with a good architecture.

There is perhaps an analogy with the architecture of office spaces. An arrangement where everyone has an individual office can inhibit communication and the flow of ideas. On the other hand, an open plan office often causes problems because noise and other distractions interfere with productivity.

The structure of an Ada program is based primarily around the concept of a *package*, which groups related entities together and provides a natural framework for hiding implementation details from its clients.

Package specifications and bodies

Early languages such as Fortran had a flat structure with everything essentially at the same level. As a consequence all data (other than that local to a subroutine) is visible everywhere. This can be considered as rather like an open plan office. The same flat structure appears in C, although C does provide a degree of encapsulation by allowing programmer control over the external visibility of functions and file-scope variables.

Other languages such as Algol and Pascal have a simple block structure, rather like nested Russian dolls. This is a bit better but really is no more than having an open plan office subdivided into more such offices. There are still big problems of communication.

Consider the simple problem of a stack of numbers. The desired protocol is that an item can be added to the stack by calling a procedure Push and that the

top item can be removed from the stack by calling a function Pop – and perhaps also a procedure Clear to set the stack to an empty state. We do not want any other means of manipulating the stack since we want this protocol to be independent of the way we implement it.

Now consider the following implementation of a stack written in Pascal. The stack is represented by an array of reals and there are three operations, Push and Pop to add items and remove items respectively, and Clear to set it empty. We also declare a constant max and give it a suitable value such as 100. This avoids writing 100 in several places, which would be bad if we changed our minds later on about the required size of the stack.

```
const   max = 100;

var     top : 0 .. max;
        a : array[1..max] of real;

procedure Clear;
begin
        top := 0
end;

procedure Push(x : real);
begin
        top := top + 1;
        a[top] := x
end;

function Pop : real;
begin
        top := top – 1;
        Pop:= a[top + 1]
end
```

The main trouble with this is that max, top and a have to be declared outside Push, Pop and Clear so that they can all be accessed. And from any part of the program from which we can call Push, Pop and Clear we can also change a and top directly and so bypass the protocol and create an inconsistent stack.

This is a source of danger. If we want to monitor how many times the stack is changed then adding monitoring statements to count the calls of Push, Pop and Clear to do this is not adequate. Similarly, if we are reviewing a large program and are looking for all places where the stack is changed then we have to track all references to top and a as well as the calls of Push, Pop and Clear.

This problem applies to C as well as to Fortran and Pascal. These languages to some extent overcome the problem by adding some form of separate compilation facility. Those entities which are to be visible to other separately compiled units can then be marked by special statements such as **extern** or by

using a header file. However, type checking in these languages is weaker across compilation units than within a single file.

The technique in Ada is to use a package to encapsulate and hide the data shared by Push, Pop and Clear so that only those subprograms can access it. A package comes in two parts – its specification which describes its interface to other units, and its body which describes how it is implemented. We can paraphrase this by saying that the specification says *what* it does and the body says *how* it does it. The specification would simply be

```
package Stack is
  procedure Clear;
  procedure Push(X: Float);
  function Pop return Float;
end Stack;
```

This just describes the interface to the outside world. So outside the package all that is available are the three subprograms. The specification gives just enough information for the external client to write calls to the subprograms and for the compiler to compile the calls. The body could then be written as

```
package body Stack is

  Max: constant := 100;
  Top: Integer range 0 .. Max := 0;
  A: array (1 .. Max) of Float;

  procedure Clear is
  begin
    Top := 0;
  end Clear;

  procedure Push(X: Float) is
  begin
    Top := Top + 1;
    A(Top) := X;
  end Push;

  function Pop return Float is
  begin
    Top := Top – 1;
    return A(Top + 1);
  end Pop;

end Stack;
```

The body gives the full details of the subprograms and also declares the hidden objects Max, Top and A. Note the initial value of zero for Top.

In order to make use of the entities declared in a package, the client code must mention the package by means of a *with clause* thus

```
with Stack;
procedure Some_Client is
  F: Float;
begin
  Stack.Clear;
  Stack.Push(37.4);
  …
  F := Stack.Pop;
...
  Stack.Top := 5;          -- illegal!
end Some_Client;
```

So now we know that the required protocol is enforced. The client cannot accidentally or purposely interfere with the inner workings of the stack. Note in particular that the direct assignment to Stack.Top is prevented since Top is not visible to the client (it is not mentioned in the specification of the stack).

Observe carefully that there are three entities to consider: the specification of the package, its body, and of course the client.

There are important rules concerning their compilation. The client cannot be compiled without the specification being available and the body also cannot be compiled without the specification being available. But there are no similar constraints relating to the client and the body. If we decide to change the details of the implementation and this does not require the specification to be changed then the client does not have to be recompiled.

Packages and subprograms at the top level (that is, not nested inside other packages or subprograms) can always be and usually are compiled separately. They are often known as library units and said to be at the library level.

Note that the package Stack is mentioned each time an entity in it is used. This ensures that the client code is very clear as to what it is doing. Sometimes repeating the package name is tedious and so we can add a *use clause* thus

```
with Stack;  use Stack;
procedure Client is
begin
  Clear;
  Push(37.4);
  ...
end Client;
```

Of course if there were two packages Stack1 and Stack2, both declaring a procedure called Clear, and we try to “with” and “use” both of them then the

code would be ambiguous and the compiler would reject it. In such a case the solution is to supply the desired package name explicitly, for example Stack2.Clear.

In conclusion, the specification defines a *contract* between the client and the package. The body promises to implement the specification and the client promises to use the package as described by the specification. Finally the compiler ensures that both sides stick to the contract. We will come back to these thoughts later in this chapter and also in the last chapter when we look into Ada 2012's contract-based programming and the ideas behind the SPARK toolset, respectively.

The careful reader will note that we have been ignoring issues of stack overflow (calling Push when Top=Max) and underflow (calling Pop when Top=0). Indeed, if either of these unpleasantries arise then a range check on Top will fail, raising Constraint_Error. It would be nice if the specifications for Push and Pop in the Stack package specification could explicitly include the preconditions that they are assuming, with corresponding checking enforced. Then the programmer intending to make use of the package would know what is expected of the actual parameter that is passed. Such a facility has been added in Ada 2012; it is part of the contract-based programming support that will be discussed below.

A vital point about Ada is that the strong type matching is enforced across compilation unit boundaries. Exactly the same checking applies, whether the program is just one compilation unit or consists of several units distributed across various files.

Private types

Another feature of a package is that part of the specification can be hidden from the client. This is done using a so-called *private part*. The above package Stack only implements a single stack. It might be more useful to declare a package that enabled us to declare many stacks – to do this we need to introduce the concept of a stack type.

We might write

```
package Stacks is                                    -- visible part
  type Stack is private;                             -- private type
  procedure Clear(S: out Stack);
  procedure Push(S: in out Stack; X: in Float);
  procedure Pop(S: in out Stack; X: out Float);

private                                              -- private part
  Max: constant := 100;
  type Vector is array (1 .. Max) of Float;
  type Stack is                                      -- full type
    record
      A: Vector;
      Top: Integer range 0 .. Max := 0;
    end record;
end Stacks;
```

This is a straightforward generalization of the single-stack version, but we note that Ada 2012 offers a choice of declaring Pop as either a function returning a Float result, or a procedure taking a Float as an **out** parameter. Pop is permitted as a function since Ada 2012 allows functions to have **out** or **in out** parameters; relaxing a restriction that has been present since the original Ada design. Nonetheless, although the declaration of Pop as a function is permitted, we have followed the more traditional Ada approach and expressed Pop as a procedure. This style is consistent with the invocation of Push, and also makes it clearer that there is a side effect.

The package body would then be

```
package body Stacks is

  procedure Clear(S: out Stack) is
  begin
    S.Top := 0;
  end Clear;

  procedure Push(S: in out Stack; X: in Float) is
  begin
    S.Top := S.Top + 1;
    S.A(Top) := X;
  end Push;

  -- procedure Pop similarly

end Stacks;
```

The user can now declare lots of stacks and act on them individually thus

```
with Stacks; use Stacks;
procedure Main is
  This_One: Stack;
  That_One: Stack;
begin
  Clear(This_One);  Clear(That_One);
  Push(This_One, 37.4);
  ...
end Main;
```

The detailed information about the type Stack is given in the private part of the package and, although visible to the human reader, is not directly accessible to the code written by the client. So the specification is logically split into two parts, the visible part (everything before the keyword **private**) and the private part.

If the private part alone is changed then the text of the client will not need changing but the client code will need recompiling because the object code might change even though the source code does not.

Any necessary recompilation is ensured by the compilation system and can be performed automatically if desired. Note carefully that this is required by the Ada language and is not simply a property of a particular implementation. It is never left to the user to decide when recompilation is necessary and so there is no risk of attempting to link together a set of inconsistent units – a big hazard in languages that do not specify precisely the interaction between compiling, binding and linking.

Finally, note the modes **in**, **out** and **in out** on the parameters. These refer to the flow of information and are explained in Chapter 6 on Safe Object Construction.

Generic contract model

Templates are an important feature of languages such as C++ (and more recently Java and C#). These correspond to generics in Ada and in fact C++ based its templates partly on Ada generics. Ada generics are type-safe because of the so-called *contract model*.

We can extend the stack example to enable us to declare stacks of any type and any size (we can do the latter in other ways as well). Consider

```
generic
   Max: Integer;                          -- formal generic parameters
   type Item is private;
package Generic_Stacks is
   type Stack is private;
   procedure Clear(S: out Stack);
   procedure Push(S: in out Stack; X: in Item);
   procedure Pop(S: in out Stack; X: out Item);

private                                                -- private part
   type Vector is array (1 .. Max) of Item;
   type Stack is
      record
         A: Vector;
         Top: Integer range 0 .. Max := 0;
      end record;
end Generic_Stacks;
```

with an appropriate body obtained simply by replacing Float by Item.

The generic package is just a template and in order to be used in a program it has to be instantiated with appropriate actual parameters corresponding to the two generic formal parameters Max and Item. The result of instantiating a generic package is the declaration of an actual package. For example if we want stacks of integers with maximum size 50, we write

```
package Integer_Stacks is
                  new Generic_Stacks(Max => 50, Item => Integer);
```

This declares a package called Integer_Stacks which we can then use in the normal way. The essence of the contract model is that if we provide parameters that correctly match the generic specification then the package obtained from the instantiation will compile and execute correctly.

Other languages do not have this desirable property. In C++, for instance, some mismatches are caught by the linker rather than the compiler and others are even left until execution and throw an exception.

There are extensive forms of generic parameters in Ada. The generic formal parameter

```
type Item is private;
```

permits the actual type to be almost any type at all. The generic formal parameter

```
type Item is (<>);
```

permits the actual type to be any integer type (such as Integer or Long_Integer) or any enumeration type (such as Signal). Within the generic we can then use all the properties common to all integer and enumeration types with the certainty that the actual type will indeed provide these properties.

The generic contract model is very important. It enables the development of flexible but safe general-purpose libraries. An important goal is that the Ada user should not ever need to pore over the code of the generic body in order to puzzle out what went wrong.

Child units

The overall architecture of an Ada system can have a hierarchical (tree-like) structure of units, which provides both flexible information hiding and ease of modification. Child units can be public or private. Given a package called Parent we can declare a public child thus

```
package Parent.Child is ...
```

and a private child thus

```
private package Parent.Slave ...
```

Both have bodies and can have private parts as usual. The key difference is that a public child essentially extends the specification of the parent (and is thus visible to clients) whereas a private child extends the private part and body of the parent (and thus is not visible to clients). The structure permits grandchildren etc to any depth.

There are various rules concerning visibility. A child unit does not need an explicit “with” clause for its parent (visibility is automatic). However, the parent body can have a “with” clause for a child if it needs to use the functionality defined in the child. But since the specification of the parent must be available before the children are compiled (since the children share the name of the parent), the parent specification cannot have a normal “with” clause for a child. More of this later.

Another rule is that the visible part of a private child has visibility of the private part of its parent (just as the body of the parent does). This extra visibility does not compromise the parent’s encapsulation, since the only units that can “with” a private child are ones that would in any event have visibility into the parent’s private part. But for a public child only its private part and its body (and not its visible part) have such visibility of the parent. This restriction prevents breaking the parent unit’s encapsulation.

A special form of with clause (the **private with** clause introduced in Ada 2005) is permitted on a package specification; it only allows the private part to have visibility of the unit concerned. This is useful, for example, where the private part of a public child needs information provided by a private child. Thus we might have an application package App and two children App.User_View and App.Secret_Details thus

```
private package App.Secret_Details is
  type Inner is ...
  ...      -- various operations on Inner etc
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is

  type Outer is private;
  ...      -- various operations on Outer visible to the user

           -- type Inner is not visible here
private
           -- type Inner is visible here

  type Outer is
    record
      X: App.Secret_Details.Inner;
      ...
    end record;
  ...
end App.User_View;
```

A normal "with" clause for Secret_Details is not permitted on User_View because this would allow the client to see information in the package Secret_Details via the visible part of User_View. Ada carefully blocks all attempts to bypass the strict visibility control.

Unit testing

One of the problems that confronts the testing of code is to ensure that the testing does not upset the software being tested. There is an echo here of Quantum Mechanics whereby when we make an observation of a particle such as an electron, the very observation itself disturbs the state of the particle.

One problem with good software design is that we strive to hide detailed information in order to produce good abstractions – by the use of private types for example. But then when we test the system we often want to observe the detailed behavior of this hidden material.

To take a trivial example we might want to know the value of Top for a particular stack declared using the package Stacks (the one where Stack is a private type). We have not provided a means of doing this. We could add a function Size to the package Stacks but this would disturb the package and require its recompilation and that of all the client code. And possibly we might introduce errors into the package we were testing or (worse) might make errors when we later removed the testing code.

Child units provide a convenient way of overcoming this difficulty. We can write

```
package Stacks.Monitor is
  function Size(S: Stack) return Integer;
end Stacks.Monitor;

package body Stacks.Monitor is
  function Size(S: Stack) return Integer is
  begin
    return S.Top;
  end Size;
end Stacks.Monitor;
```

This works because the body of a child has visibility of the private part of its parent. So we can now call the function Size at will for test purposes and when we are satisfied that the software is correct we can delete the child package. The parent package Stacks did not have to be disturbed at all.

Mutually dependent types

Many languages have the equivalent of private types especially in connection with object-oriented programming. Basically, the intrinsic operations (methods) belonging to a type are those declared in a package (or a class) along with the type. Thus the intrinsic operations of the type Stack are Clear, Push and Pop. The same structure in C++ would be written as

```
class Stack {
...                 /*  details of stack structure  */
public:
  void   Clear();
  void   Push(float);
  float  Pop();
};
```

The C++ approach is convenient in that it only has one level of naming Stack whereas in Ada we have both package name and type name, thus Stacks.Stack.

However, in practice the Ada style is not a burden especially if we apply "use" clauses. (Moreover, Ada users have the option of using a different style by giving the type some neutral name such as Object or Data so that they can then write Stacks.Object or Stacks.Data.)

On the other hand if we have two types that wish to share private information, it is very easy to write this in Ada. We can write

```
package Twins is
  type Dum is private;
  type Dee is private;
  ...
private
  ...                -- shared private part
end Twins;
```

and the private part defines both Dum and Dee and so they have mutual access to anything in the private part.

This is not so easy in other languages and involves constructs such as the controversial "friend" mechanism in C++. In Ada there is no possibility of getting it wrong or of breaking privacy in unexpected ways, and the mechanism is symmetric.

Other examples exhibit mutual recursion. Suppose we wish to study patterns of points and lines where each point has three lines through it and each line has three points on it. (This is not an arbitrary example. Two of the most fundamental theorems of projective geometry, those of the geometers Pappus of Alexandria and Girard Desargues concern such structures.) We use access types. A simple approach is a single package

```
package Points_and_Lines is
  type Point is private;
  type Line is private;
  ...
private
  type Point is
    record
      L, M, N: access Line;
    end record;
  type Line is
    record
      P, Q, R: access Point;
    end record;
end Points_and_Lines;
```

If we decided that each type deserved its own package then we could still define their mutually recursive structure using a *limited with clause*, a mechanism introduced in Ada 2005. (Two packages cannot have normal "with" clauses referring to each other because that creates a circularity that makes their initialization impossible.) We can write

```
limited with Lines;
package Points is
  type Point is private;
  ...
private
  type Point is
    record
      L, N, N: access Lines.Line;
    end record;
end Points;
```

and similarly for the package Lines. A limited with clause gives a so-called incomplete view of the types in the package concerned, which means roughly that they can only be used to form access types.

Contract-based programming

The Stacks example shown earlier in the chapter is not completely satisfactory. Although it nicely illustrates encapsulation through a private type, nothing in the package specification reflects the type's essential stackhood. We have chosen suggestive names for the various operations – Push, Pop, and Clear – but an inattentive or malicious developer could implement Push to remove an element or Pop to do an insertion. It would be useful, especially in the interest of reliability, safety and security, to have a mechanism that captures the author's intent concerning the semantics of the type and subprograms declared in the package specification.

Such a facility has been introduced in Ada 2012. Known as *contract-based programming* and analogous to the Design by Contract™ approach in Eiffel, it permits the programmer to specify subprogram preconditions, subprogram postconditions, and private type invariants. These take the form of Boolean expressions that can be checked at run time under the control of a pragma, and that are associated with the relevant entity (a subprogram for a precondition or a postcondition, and a private type for an invariant) via a new piece of syntax referred to as an *aspect specification*. In brief:

- A precondition is checked at the point of call, and reflects the obligations of the caller;

- A postcondition is checked at any point of normal return, and reflects the obligations of the called subprogram; and
- An invariant is equivalent to a postcondition of each subprogram that is visible outside the package and thus is checked on return from each such subprogram. An invariant reflects the guaranteed "global" state of the program after any of these subprograms returns.

Below is a version of the package Stacks that illustrates all three concepts. In order to have a non-trivial type invariant we have added the requirement that duplicate elements are not allowed.

```
package Stacks is

  type Stack is private
    with
      Type_Invariant => Is_Unduplicated(Stack);

  function Is_Empty(S: Stack) return Boolean;
  function Is_Full(S: Stack) return Boolean;
  function Is_Unduplicated(S: Stack) return Boolean;

  function Contains(S : Stack; X : Float) return Boolean;
  -- Note: Contains(S, X) implies not Is_Empty(S)

  procedure Push(S: in out Stack; X: in Float)
    with
      Pre => not Contains(S, X) and not Is_Full(S),
      Post => Contains(S, X);

  procedure Pop(S: in out Stack; X : out Float)
    with
      Pre => not Is_Empty(S),
      Post => not Contains(S, X) and not Is_Full(S);

  procedure Clear(S : in out Stack)
    with
      Post => Is_Empty(S);

private
  ...
end Stacks;
```

The notation should be self-explanatory. The absence of a precondition (as in the case of the procedure Clear) is equivalent to a True precondition.

Contracts (that is, preconditions, postconditions, and invariants) have several uses. At a minimum, they specify the programmer's intent unambiguously and can serve as formal documentation. But they can also be used to generate run-

time checks to ensure that the associated Boolean conditions are true. The Assertion_Policy pragma controls whether checks are generated. The general form is

pragma Assertion_Policy(*policy_identifier*);

If *policy_identifier* is Check then checks are generated; if *policy_identifier* is Ignore, then assertions are ignored at run time. (The default, in the absence of the pragma, is implementation defined.)

A third application of contracts is to aid in the use of formal proofs of program properties, for example to show that the code of a subprogram is consistent with its pre- and postconditions. This approach is illustrated by the SPARK language as will be discussed in a later chapter.

Ada 2012 includes a variety of features in connection with contract-based programming that were not illustrated in this simple example. The language allows quantification expressions ("**for all**" and "**for some**") and permits certain function bodies in package specifications since often the specific logic of a pre- or postcondition is in effect part of the package's interface. Ada 2012 also supplies the attribute 'Old allowing the original value of a formal parameter to be referenced in the postcondition, and the attribute 'Result analogously allowing the function result to be referenced in the postcondition The interested reader can refer to the Ada 2012 reference manual or rationale (see bibliography) for further information on such details.

Note that there is considerable flexibility in how specific the contracts should be. In our example, the only requirement on Push (its postcondition) is that the element needs to be inserted in the stack. However, the last-in–first-out semantics of stacks has a more demanding requirement: the element needs to be inserted such that it will be removed the next time Pop is called. Likewise, as specified above, Pop only has the obligation to remove some element. More precisely, it should remove the element that was most recently Pushed. These more specific contracts can be expressed using Ada 2012 features; this is left as an exercise for the interested reader.

5 Safe Object-Oriented Programming

Object-Oriented Programming (OOP) first appeared in the late 1960s in Simula, spread to the research and academic community with languages such as Smalltalk, and then in the late 1980s and early 1990s took hold in the mainstream with C++ and later Java. Its supreme merit is said to be its flexibility. But flexibility is somewhat like freedom discussed in the Introduction – the wrong kind of flexibility can be an opportunity that permits dangerous errors to intrude.

The key idea of OOP is that the objects dominate the programming, and subprograms (methods) that manipulate objects are properties of objects. The other, older, view sometimes called Function-Oriented (or structured) programming, is that programming is primarily about functional decomposition and that it is the subprograms that dominate program organization, and that objects are merely passive things being manipulated by them.

Both views have their place and fanatical devotion to just a strict object view is often inappropriate.

Ada strikes an excellent balance – we can refer to it as "methodology agnostic" as compared with, say, Java, which is "pure" OO – and enables either approach to be taken according to the needs of the application. Indeed, Ada has incorporated the idea of objects right from its inception in 1980 through the concept of packages which encapsulate types and the operations upon them, and tasks that encapsulate independent activities. Ada 95 introduced the main features that one associates with OOP: inheritance, polymorphism, and dynamic binding, together with the notion of a "class" as a collection of types related through inheritance.

Object-Orientation versus Function-Orientation

We will look at two examples which can be used to illustrate various points. They are chosen for their familiarity which avoids the need to explain particular application areas. The examples concern geometrical objects (of which there are lots of kinds) and people (of which there are only two kinds, male and female).

Consider the geometrical objects first. For simplicity we will consider just flat objects in a plane. Every object has a position. In Ada we can declare a root object which has properties common to all objects thus

```
type Object is tagged
  record
    X_Coord: Float;
    Y_Coord: Float;
  end record;
```

The word **tagged** distinguishes this type from a plain record type (such as Date in Chapter 3) and indicates that it can be extended. Moreover, objects of this type carry a tag with them at execution time and this tag identifies the type of the object. We are going to declare various specific object types such as Circle, Triangle, Square and so on in a moment and these will all have distinct values for the tag. The components X_Coord and Y_Coord are of course the coordinates of the centre of the object.

We can declare various properties of geometrical objects such as area and moment of inertia about the centre. Every object has such properties but they vary according to shape. These properties can be defined by functions and they are declared in the same package as the corresponding type. We can start with

```
package Geometry is
  type Object is abstract tagged
    record
      X_Coord, Y_Coord: Float;
    end record;

  function Area(Obj: Object) return Float is abstract;
  function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

We have declared the type and the operations as abstract. We don't actually want any objects of type Object and making it abstract prevents us from inadvertently creating any. We want real objects such as a Circle, which have properties such as Area. If we did want to discuss a plain point without any area then we should declare a specific type Point for this. The functions Area and Moment have been declared as abstract also. This ensures that when we declare a genuine type such as Circle then we are forced to declare concrete functions Area and Moment with appropriate code.

We can now declare the type Circle. It is best to use a child package for this

```
package Geometry.Circles is
  type Circle is new Object with
    record
      Radius: Float;
    end record;

  function Area(C: Circle) return Float;
  function Moment(C: Circle) return Float;
end Geometry.Circles;

with Ada.Numerics;  use Ada.Numerics;          -- to give access to π
package body Geometry.Circles is
  function Area(C: Circle) return Float is
  begin
    return π * C.Radius**2;                    -- uses Greek letter π
  end Area;

  function Moment(C: Circle) return Float is
  begin
    return 0.5 * C.Area * C.Radius**2;
  end Moment;
end Geometry.Circles;
```

The key feature here is the use of *type derivation* in the declaration of Circle. By declaring Circle as **new** Object we will implicitly inherit the visible operations on Object from package Geometry, unless we override them. Since these operations are abstract and Circle is not abstract, we are actually forced to override them and thus we declare Area and Moment explicitly. The type derivation includes an extension part (the Radius component), and this is added to the components present in Object (X_Coord and Y_Coord).

Note that the code defining the Area and Moment functions is in the package body. We recall from the chapter on Safe Architecture that this means that the code can be changed and recompiled as necessary without forcing recompilation of the description of the type itself and consequently all those programs that use it.

We could then declare other types such as Square (which has an extra component giving the length of the side), Triangle (three components giving the three sides) and so on without disturbing the existing abstract type Object and the type Circle in any way.

The various types form an inheritance hierarchy rooted at Object and this set of types (a *class* in Ada terminology) is denoted by Object'Class. Ada carefully distinguishes between a specific type such as Circle and a class of types such as Object'Class. This distinction avoids confusion that can occur in other languages. If we subsequently define other types as extensions of the type Circle then we can usefully talk about the class Circle'Class.

The function Moment declared above illustrates the use of the prefixed notation. We can write either of

```
C.Area                  -- prefixed notation
Area(C)                 -- functional notation
```

The prefixed notation was introduced in Ada 2005 and emphasizes the object model, indicating that we consider the object C to be the predominant entity rather than the function Area.

Suppose now that we have declared various objects, perhaps

```
A_Circle: Circle := (1.0, 2.0, Radius => 4.5);
My_Square: Square := (0.0, 0.0, Side => 3.7);
The_Triangle: Triangle := (1.0, 0.5, A => 3.0, B => 4.0, C => 5.0);
```

By way of illustration, we have used named notation for components other than the *x* and *y* coordinates which are common to all the types.

We might have a procedure to output the properties of a general object. We might write

```
procedure Print(Obj: Object'Class) is     -- Obj is polymorphic
begin
  Put("Area is ");  Put(Obj.Area);        -- dispatching call of Area
  ...                                     -- and so on
end Print;
```

and then

```
Print(A_Circle);
Print(My_Square);
```

The formal parameter Obj is polymorphic, meaning that it can reference objects of different types (from any type in the hierarchy rooted at Object) at different times, in particular at different calls of Print.

The procedure Print can take any item in the class Object'Class. Within the procedure, the call to Area is dynamically bound and calls the function Area appropriate to the specific type of the parameter Obj. This always works safely since the language rules are such that every possible object in the class Object'Class is of a specific type derived ultimately from Object and will have a function Area. Note that the type Object itself was abstract and so no geometrical object of that type can be created – accordingly it does not matter that the function Area for the type Object is abstract and has no code – it could never be called anyway.

In a similar way we might have types concerning persons. Consider

```
package People is
  type Person is abstract tagged
    record
      Birthday: Date;
      Height: Inches;
      Weight: Pounds;
    end record;

  type Man is new Person with
    record
      Bearded: Boolean;          -- whether he has a beard
    end record;

  type Woman is new Person with
    record
      Births: Integer;           -- how many children she has borne
    end record;

  ...      -- various operations
end People;
```

Since there is no possibility of any additional types of persons we could alternatively describe them by using a variant record, which is more in the line of function-oriented programming. Thus

```
type Gender is (Male, Female);

type Person (Sex: Gender) is
  record
    Birthday: Date;
    Height: Inches;
    Weight: Pounds;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Births: Integer;
    end case;
  end record;
```

and we might then declare various operations on this version of the type Person. Each operation would have to have a **case** statement to take account of the two sexes.

This might be considered rather old fashioned and inelegant. However, it has its own considerable advantages.

If we need to add another **operation** in the Object-Oriented formulation then the whole structure will need to be recompiled – each type will need to be revisited in order to implement the new operation. If we need to add another **type** (such as a Pentagon) then the existing structure can be left unchanged.

In the case of the Function-Oriented formulation, the situation is completely reversed (basically we simply interchange the words type and operation).

If we need to add another **type** in the Function-Oriented formulation then the whole structure will need to be recompiled – each operation will need to be revisited to implement the new type (by adding another branch to its case statement). If we need to add another **operation** then the existing structure can be left unchanged.

The Object-Oriented approach has often been lauded as so much safer than Function-Oriented programming because there are no **case** statements to maintain. This certainly is true but sometimes the maintenance is harder if new operations are added because they have to be added individually for every type.

Ada offers both approaches and both approaches are safe in Ada.

Overriding indicators

One of the dangers of Object-Oriented programming occurs with overriding inherited operations. When we add a new type to a class we can add new versions of all the appropriate operations. If we do not add a new operation then that of the parent is inherited.

The danger is that we might attempt to add a new version but spell it incorrectly

```
function Aera(C: Circle) return Float;
```

or get a parameter or result wrong

```
function Area(C: Circle) return Integer;
```

In both cases the existing function Area is not overridden but a totally new operation added. And then when a class-wide operation dispatches to Area it will call the inherited version rather than the one that failed to override it. Such bugs can be very difficult to find – the program compiles quietly and seems to run but just produces curious answers.

(Actually, Ada has already provided a safeguard here because we declared Area for Object as abstract and this is a further defensive measure. But if we had a second generation or had not had the wisdom to make Area abstract then we would be in trouble.)

In order to guard against such mistakes we can take advantage of syntax introduced in Ada 2005 and write for example

```
overriding
function Area(C: Circle) return Float;
```

Then if we make an error we will not get a new operation but instead the program will fail to compile. On the other hand, if we did truly want to add a new operation then we could assert that also by

```
not overriding
function Aera(C: Circle) return Float;
```

Such overriding indicators are always optional, largely for compatibility with earlier versions of Ada.

Admittedly the **not overriding** syntax is somewhat heavy, especially since this will be the common case. The ideal rule would be to require specifying the **overriding** keyword for all overriding declarations, and only for overriding declarations; that is, to use the **overriding** keyword but not the **not overriding** form. That rule would catch both kinds of errors:

- Misspellings and other accidental non-overridings, since a non-overriding declaration would have the **overriding** keyword; and
- Accidental overriding (for example when a subprogram with the same profile is added to a parent type), since an overriding declaration would not have an **overriding** keyword.

Compatibility with Ada 95 prevented these semantics from being adopted in Ada 2005, but their effect can be obtained in AdaCore's GNAT compiler through the following switches:

- -gnatyO, which warns about overriding declarations that are not marked with the overriding keyword, and
- -gnatwe, which treats warnings as errors.

Languages such as C++ and Java provide less assistance in this area and consequently subtle errors can remain undetected for some time.

Dispatchless programming

In safety-critical programming, the dynamic selection of code is sometimes forbidden. Safety is enhanced if we can prove that the flow of control follows a strict pattern with, for example, no dead code. Traditionally this means that we

have to use a more function-oriented approach, with visible **if** statements and **case** statements to select the appropriate flow path.

Although dynamic dispatching is at the heart of much of the power of Object-Oriented programming, other object-oriented features (chiefly code reuse through inheritance) are valuable. Thus we might value the ability to extend types and thereby share much coding but declare specific named operations where no dynamic behavior is required. We might also wish to use the prefixed notation which has a number of advantages.

Ada has a facility known as pragma Restrictions which enables a programmer to ensure that specific features of Ada are not used in a particular program. In this case we write

```
pragma Restrictions(No_Dispatch);
```

and this ensures (through compile-time checks) that no use is made of the construction X'Class which in turn means that no dispatching calls are possible.

Note that this exactly matches the requirements of SPARK which, as we mentioned in the Introduction, is often used for critical software. SPARK permits type extension but does not permit class-wide types and operations.

If we do specify the restriction No_Dispatch then the implementation is able to reduce the code overheads typically associated with OOP. There is of course no need to generate a dispatch table for each type. (A dispatch table is a look-up table that contains the addresses of the various specific operations for the type.) Moreover, there is also no need to store a tag in every record structure.

There are other less obvious benefits as well. In full OOP some of the predefined operations such as equality are dispatching and so the code overheads associated with them are also avoided. The net result is that the use of the pragma minimizes the need for the justification of deactivated code (code that is present in the executable and that can be traced back to specific requirements, but which will never be executed) for DO-178B or DO-178C certification.

Interfaces and multiple inheritance

Some have looked upon multiple inheritance as a Holy Grail – an objective against which languages should be judged. This is not the place to digress on the history of various techniques that have been used. Rather we will summarize the key problems.

Suppose that we were able to inherit arbitrarily from two parent types. Recall that fabulous book *Flatland* written by Edwin Abbott (the second edition was

published in 1884). It is a satire on class structure (in the sociological, not the programming sense) and concerns a world in which people are flat geometrical objects. The working classes are triangles, the middle classes are other polygons. The aristocracy are circles. Curiously, all females are two-sided and thus simply a line segment.

So using the two classes Objects and Persons introduced above, we could conceive of representing the inhabitants of Flatland by a type derived from both such as

```
type Flatlander is new Geometry.Object and People.Person; -- illegal
```

The question now arises as to what are the properties inherited from the two parent types? We might expect a Flatlander to have components X_Coord and Y_Coord inherited from Object and also a Birthday inherited from Person, although Height and Weight might be dubious for a two-dimensional person. And certainly we would expect an operation such as Area to be inherited because clearly a Flatlander has an area and indeed a moment of inertia.

But we see potential problems in the general case. Suppose both parent types have an operation with the same identifier. This would typically arise with operations of a rather general nature such as Print, Make, Copy and so on. Which one is inherited? Suppose both parents have components with the same identifier. Which one do we get? These problems particularly arise if both parents themselves have a common ancestor.

Some languages have provided multiple inheritance and devised somewhat lengthy rules to overcome these difficulties (C++ and Eiffel for example). Possibilities include using renaming, mentioning the parent name for ambiguous entities, and giving precedence to the first parent type in the list. Sometimes the solutions have the flavor of unification for its own sake – one person's unification is often another person's confusion. The rules in C++ give plenty of opportunities for the programmer to make mistakes.

The difficulties are basically twofold: inheriting components and inheriting the *implementation* of the operations from more than one parent. But there is generally no problem with inheriting the *specification* (that is, the *interface*) of the operations. This solution was adopted by Java and has proved successful and is also the approach used by Ada.

So the Ada rule, introduced in Ada 2005, is that we can inherit from more than one type thus

```
type T is new A and B and C with
  record
    ...             -- additional components
  end record;
```

but only the first type in the list (A) can have components and concrete operations. The other types must be what are known as *interfaces* – the reuse of the Java term was intentional – which are essentially abstract types without components and all of whose operations are abstract or null procedures. (The first type could be an interface as well.)

We can reformulate the type Object as an interface as follows

```
package Geometry is
   type Object is interface;

   procedure Move(Obj: in out Object;
                  New_X, New_Y: in Float) is abstract;
   function X_Coord(Obj: Object) return Float is abstract;
   function Y_Coord(Obj: Object) return Float is abstract;
   function Area(Obj: Object) return Float is abstract;
   function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

Observe that the components have been deleted and replaced by further operations. The procedure Move enables an object to be moved – that is it sets both the *x* and *y* coordinates and the functions X_Coord and Y_Coord return its current position.

Note that the prefixed notation means that we can still access the coordinates by for example A_Circle.X_Coord and The_Triangle.Y_Coord just as when they were visible components.

So now when we declare a concrete type Circle we have to provide implementations of all these operations. Perhaps

```
package Geometry.Circles is
   type Circle is new Object with private;                -- partial view

   procedure Move(C: in out Circle; New_X, New_Y: in Float);
   function X_Coord(C: Circle) return Float;
   function Y_Coord(C: Circle) return Float;
   function Area(C: Circle) return Float;
   function Moment(C: Circle) return Float;

   function Radius(C: Circle) return Float;
   function Make_Circle(X, Y, R: Float) return Circle;
```

```
private
  type Circle is new Object with                    -- full view
    record
      X_Coord, Y_Coord: Float;
      Radius: Float;
    end record;
end Geometry.Circles;

package body Geometry.Circles is
  procedure Move(C: in out Circle; New_X, New_Y: in Float) is
  begin
    C.X_Coord := New_X;
    C.Y_Coord := New_Y;
  end Move;

  function X_Coord(C: Circle) return Float is
  begin
    return C.X_Coord;
  end X_Coord;

  -- and similarly Y_Coord and Area and Moment as before
  -- also functions Radius and Make_Circle
end Geometry.Circles;
```

We have made the type Circle private so that all the components are hidden. Nevertheless the partial view reveals that it is derived from the type Object and so must have all the properties of the type Object. Note how we also add functions to create a circle and to access the radius component.

So the essence of programming with interfaces is that we have to implement the properties promised. It is not so much multiple inheritance of existing properties but multiple inheritance of contracts to be satisfied. In short, Ada allows multiple interface inheritance but single implementation inheritance.

Returning now to Flatland, we can declare

```
package Flatland is
  type Flatlander is abstract new Person and Object with private;

  procedure Move(F: in out Flatlander; New_X, New_Y: in Float);
  function X_Coord(F: Flatlander) return Float;
  function Y_Coord(F: Flatlander) return Float;
```

```
private
  type Flatlander is abstract new Person and Object with
    record
      X_Coord, Y_Coord: Float := 0.0;              -- at origin by default
      ... -- any new components we wish
    end record;
end Flatland;
```

and the type Flatlander will inherit the components Birthday etc of the type Person, any operations of the type Person (we didn't show any above) and the abstract operations of the type Object. However, it is convenient to declare the coordinates as components since we need to do that eventually and we can then override the inherited abstract operations Move, X_Coord and Y_Coord with concrete ones. Note also that we have given the coordinates the default value of zero so that any flatlander is by default at the origin.

The package body is

```
package body Flatland is
  procedure Move(F: in out Flatlander; New_X, New_Y: Float) is
  begin
    F.X_Coord := New_X;
    F.Y_Coord := New_Y;
  end Move;

  function X_Coord(F: Flatlander) return Float is
  begin
    return F.X_Coord;
  end X_Coord;

  -- and similarly Y_Coord
end Flatland;
```

Making Flatlander abstract means that we do not have to implement all the operations such as Area just yet. And finally we could declare a type Square suitable for Flatland (when originally written the book was published anonymously and the author designated as A Square) as follows

```
package Flatland.Squares is
  type Square is new Flatlander with
    record
      Side: Float;
    end record;

  function Area(S: Square) return Float;
  function Moment(S: Square) return Float;
end Flatland.Squares;
```

```
package body Flatland.Squares is

  function Area(S: Square) is
  begin
    return S.Side**2;
  end Area;

  function Moment(S: Square) is
  begin
    return S.Area * S.Side**2 / 6.0;
  end Moment;

end Flatland.Squares.
```

and all the operations are thereby implemented. By way of illustration we have made the extra component Side of the type Square directly visible but we could have used a private type. So we can now declare Dr Abbott as

```
A_Square: Square := (Flatlander with Side => 3.00);
```

and he will have all the properties of a square and a person. Note the extension aggregate which takes the default values for the private components and gives the additional visible component explicitly.

There are other important properties of interfaces that can only be touched upon in this overview. An interface can have a null procedure as an operation. A null procedure behaves as if it has a null body – that is, it can be called but does nothing. If two ancestors have the same operation then a null procedure overrides an abstract operation with the same parameters and results. If two ancestors have the same abstract operation with equivalent parameters and results then these merge into a single operation to be implemented. If the parameters and results are different then this results in overloading and both operations have to be implemented. In summary the rules are designed to minimize surprises and maximize the benefits of multiple inheritance.

Substitutability

This section treats a specialized topic that may be of interest to the advanced reader.

Inheritance can be regarded from two perspectives. As a language feature, it means the ability to derive from a parent type (superclass) and inherit state (components) and operations, with the ability to add new components, to override inherited operations, and to add new operations. But from a data modelling or type theoretic perspective, inheritance means a specialization ("is a") relationship between a subclass and its superclass: if class S is a

subclass of T, then any instance of S is also an instance of T. This property is the basis of polymorphism; in Ada terms, for any tagged type T, a variable of type T'Class can refer to an object of type T or of any type derived directly or indirectly from T. This means that any operation that works for T should also work (either inherited or overridden) when applied to an instance of any of T's subclasses.

A more formal way of stating this last requirement is known as the *Liskov Substitution Principle* (LSP) [4] where it is expressed in type theoretic terms:

> *Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects of type S where S is a subtype of T*

(Here "subtype" means "subclass".)

Designing a class hierarchy to adhere to LSP is good practice. If LSP is violated then it is possible for an inappropriate operation to be applied to a subclass instance through dynamic binding, resulting in a run-time error. This can occur if inheritance is used as the relationship between two classes when a less strongly coupled relationship should have been used instead.

Although LSP may seem intuitively obvious, its interaction with contract-based programming is not. Recall that, with contract-based programming, you can augment the specification of a subprogram with a precondition, a postcondition, or both. The question then arises: if you override an operation, does LSP place any requirements on the pre- and postconditions for the overriding version? The answer is "yes": the precondition cannot be strengthened (for example you cannot form a precondition for the subclass's version by "and"ing a condition onto the superclass's precondition), and symmetrically the postcondition cannot be weakened.

At first glance this may seem opposite to what you would expect. A subclass generally constrains the value set of its superclass, so it would seem to make sense to impose a stronger precondition on the subclass's version of an operation. Upon closer inspection, however, this would violate LSP. From the caller's perspective, invoking an operation X.Op(…) for a polymorphic variable X of type T means ensuring that Op's precondition for type T is met. The author of this code has no way of knowing all the subclasses of T. If X happens to refer to an object of type T1, and the precondition of Op for T1 is stronger than that for T, then invoking X.Op(…) will fail with a precondition check error. Analogous reasoning shows that an operation's postcondition should not be weakened by a subclass. The caller will expect the invocation to satisfy the superclass operation's postcondition; if this is not fulfilled by the subclass, then again the program will fail.

DO-178C's Object-Oriented Technology and Related Techniques supplement (DO-332) [5] has addressed this issue. It does not mandate compliance with LSP but instead supplies guidance on the verification of "Local Type Consistency".

"Type consistency" means adherence to LSP: subclass operations are not allowed to strengthen preconditions or weaken postconditions. "Local" means that the analysis need only consider contexts that actually occur in the program. For example if an operation's precondition is strengthened by a mischievous operation that is never invoked – wherever there is a dynamically bound call, it can be demonstrated that the target object cannot be from the subclass that violates LSP – then the violation does no harm.

DO-332 offers three possible approaches to demonstrating Local Type Consistency; one based on formal methods and the other two on requirements-based testing:

- Formally verify substitutability.
- Ensure that each type passes all the tests of all its parent types which it can replace.
- For each dispatch point, test every method that can be invoked (pessimistic testing)

The first approach offers direct verification of LSP. Ada 2012's explicit support for preconditions and postconditions helps make programs amenable to analysis by automated formal methods-based tools. This approach is recommended when appropriate formal methods can be used, as for example with the SPARK Pro toolset, since they provide the highest level of confidence that the Type Consistency objective is met.

The second approach is well adapted to verification based on unit testing. In such a context, each operation in a class has a corresponding set of tests to verify that the operation's requirements have been met. An overriding operation usually has extended requirements compared to the one it overrides, so it will be associated with an extended set of tests. Each class can be tested separately by calling the tests of its methods. In order to verify substitutability of a given tagged type by testing, the idea is to run all the tests for all parent types with objects of the given tagged type. If all the parent tests pass, this provides a high degree of confidence that objects of the new tagged type can properly substitute for parent type objects. Aunit, the unit testing tool included in GNAT Pro, offers the necessary support for automating such verification activities and provide specific examples showing the detection of LSP violations.

The third approach may be the simplest verification method when dispatching calls are rare and the class hierarchy is shallow and/or narrow. In the context of GNAT Pro, the GNATstack tool can locate all the dispatching points of the application and identify the potential destination subprograms for each.

Further discussion of this issue may be found in the *High-Integrity Object-Oriented Programming in Ada* report [6] produced by AdaCore.

6 Safe Object Construction

This chapter covers a number of aspects of the control of objects. By objects here we mean both small objects in the sense of simple constants and variables of an elementary type such as Integer and big objects in the sense of Object-Oriented Programming.

Ada provides good control and flexibility in this area. This control is in many cases optional but the good programmer will use the features wherever possible and the good manager will insist upon them being used wherever possible.

Variables and constants

As we have seen we can declare a variable or a constant by writing

```
Top: Integer;                          -- a variable
Max: constant Integer := 100;          -- a constant
```

respectively. Top is a variable and we can assign new values to it whereas Max is a constant and its value cannot be changed. Note that when we declare a constant we have to give it a value since we cannot assign to it afterwards. A variable can optionally be given an initial value as well.

The advantage of using a constant is that it cannot be changed accidentally. It is not only a useful safeguard but it helps any person later reading the program and informs them of its status. An important point is that the value of a constant does not have to be static – that is, computed at compile time. An example was in the program for interest rates where we declared a constant called Factor

```
function Nfv_2000(X: Float) return Float is
  Factor: constant Float := 1.0 + X/100.0;
begin
  return 1000.0 * Factor**2 + 500.0 * Factor – 2000.0;
end Nfv_2000;
```

Each call of the function Nfv_2000 has a different value for X and so a different value for Factor. But Factor is constant throughout each individual call. Although this is a trivial example and it is clear that Factor is not changed during execution of an individual call nevertheless we should get into the habit of writing **constant** whenever possible.

Parameters of subprograms are another example of variables and constants.

Parameters may have three modes: **in**, **in out**, and **out**. If no mode is shown then it is taken to be **in** by default. In versions of the language up to and including Ada 2005, the only parameter mode permitted for functions was **in**. This restriction was motivated by methodological considerations, in order to discourage programmers from writing functions with side effects. However, in practice the restriction did not really work. It was still possible to achieve side effects in functions, for example, by the use of access parameters or by assigning to a non-local variable. Moreover, if a side effect were needed, the programmer did not have the ability to use the technique that most clearly showed the intent, that is through a formal parameter of the relevant mode. As a result of these considerations, Ada 2012 has finally done the right thing and functions may take parameters of mode **in out** or **out** as well as **in**.

A parameter of mode **in** is a constant whose value is given by the actual parameter. Thus the parameter X of Nfv_2000 has mode **in** and so is a constant – this means that we cannot assign to it and so are assured that its value will not change. The actual parameter can be any expression of the type concerned.

Parameters of modes **in out** and **out** are variables. The actual parameter must also be a variable. The difference concerns their initial value. A parameter of mode **in out** is a variable whose initial value is given by that of the actual parameter whereas a parameter of mode **out** has no initial value (unless the type has a default value such as **null** in the case of an access type).

Examples of all three modes occur in the procedures Push and Pop in the chapter on Safe Architecture

```
procedure Push(S: in out Stack; X: in Float);
procedure Pop(S: in out Stack; X: out Float);
```

The rules regarding actual parameters ensure that constancy is never violated. Thus we could not pass a constant such as Factor to Pop since the relevant parameter of Pop has mode **out** and this would enable Pop to change Factor.

The distinction between variables and constants also applies to access types and objects. Thus if we have

```
type Int_Ptr is access all Integer;
K: aliased Integer;
KP: Int_Ptr := K'Access;
CKP: constant Int_Ptr := K'Access;
```

then the value of KP can be changed but the value of CKP cannot. This means that CKP will always refer to K. However, although we cannot make CKP refer to any other object we can use CKP to change the value in K by

```
CKP.all := 47;                    -- change value of K to 47
```

On the other hand we might have

```
type Const_Int_Ptr is access constant Integer;
J: aliased Integer;
JP: Const_Int_Ptr := J'Access;
CJP: constant Const_Int_Ptr := J'Access;
```

where the access type itself has **constant**. This means that we cannot change the value of the object J referred to indirectly whether we use JP or CJP. Note that JP can refer to different objects from time to time but CJP cannot. Of course, the value of the object J can always be changed by a direct assignment to J.

Constant and variable views

Sometimes it is convenient to enable a client to read a variable but not to assign to it (that is, write to it). In other words to give the client a constant view of a variable. This can be done with a so-called *deferred constant* and the access types just described.

A deferred constant is one declared in the visible part of a package and for which we do not give an initial value. The initial value must then be given in the private part. Consider the following

```
package P is
  type Const_Int_Ptr is access constant Integer;
  The_Ptr: constant Const_Int_Ptr;              -- deferred constant
private
  The_Variable: aliased Integer;
  The_Ptr: constant Const_Int_Ptr := The_Variable'Access;
  ...
end P;
```

The client can read the value of The_Variable indirectly through the object The_Ptr of type Const_Int_Ptr by writing

```
K := The_Ptr.all;          -- indirect read of The_Variable
```

But since the access type Const_Int_Ptr is declared as **access constant** the value of the object referred to by The_Ptr cannot be changed by writing

```
The_Ptr.all := K;          -- illegal, cannot change The_Variable indirectly
```

However, any subprogram declared in the package P can access The_Variable directly and so write to it. This technique is particularly useful with tables where

the table is computed dynamically but we do not want the client to be able to change it.

Using a feature introduced in Ada 2005 the named access type is not really necessary since we can equally write

```
package P is
   The_Ptr: constant access constant Integer;       -- deferred constant
private
   The_Variable: aliased Integer;
   The_Ptr: constant access constant Integer := The_Variable'Access;
   ...
end P;
```

Note the double use of **constant** in the declaration of The_Ptr. The first says that The_Ptr is itself a constant. The second says that it cannot be used to change the value of the object that it refers to.

Constructor functions

Languages such as C++, Java and C# have a special syntax for functions that are used to construct new data objects, with the type (class) name used as the name of the function. Such a mechanism was considered for Ada but was rejected in the interest of simplicity. Constructors in other languages have special semantics (for example concerning the need to invoke parent type constructors, whether/when a default constructor is provided, when the constructor is invoked relative to default initialization, etc.) Ada provides several features that can be used to model constructors, for example the use of discriminants for parameterized initialization, the use of controlled types with a user-provided Initialize procedure as will be explained below, or simply the use of an ordinary function returning a value of the target type, so there was no need for a special language feature.

Limited types

The types we have met so far (Integer, Float, Date, Circle and so on) have various operations. Some are predefined, such as the equality operation to compare two values (with =) and some also have user-defined operations, such as Area in the case of the type Circle. The operation of assignment is also available for all the types mentioned so far.

Sometimes assignment is undesirable. There are two main reasons why this might be the case

- the type might represent some resource such as an access right and copying could imply a violation of security,
- the type might be implemented as a linked data structure and copying would simply copy the head of the structure and not all of it.

We can prevent assignment by declaring the type as **limited**. A good illustration of the second problem occurs if we implement the stack using a linked list. We might have

```
package Linked_Stacks is
  type Stack is limited private;
  procedure Clear(S: out Stack);
  procedure Push(S: in out Stack; X: in Float);
  procedure Pop(S: in out Stack; X: out Float);

private
  type Cell is
    record
      Next: access Cell;
      Value: Float;
    end record;

  type Stack is access all Cell;
end Linked_Stacks;
```

The body might be

```
package body Linked_Stacks is

  procedure Clear(S: out Stack) is
  begin
    S := null;
  end Clear;

  procedure Push(S: in out Stack; X: in Float) is
  begin
    S := new Cell'(S, X);
  end Push;

  procedure Pop(S: in out Stack; X: out Float) is
  begin
    X := S.Value;
    S := Stack(S.Next);
  end Pop;

end Linked_Stacks;
```

This uses the normal linked list style of implementation. Note that the type Stack is declared as limited private so that assignment of a stack as in

```
This_One, That_One: Stack;
...
This_One := That_One;          -- illegal, type Stack is limited
```

is prohibited. If assignment had been permitted then all that would have happened is that This_One would end up pointing to the start of the list defining the value of That_One. Calling Pop on This_One would simply move it down the chain representing That_One. This sort of problem is known as aliasing – we would have two ways of referring to the same entity and that is often very unwise.

In this example there is no problem with declaring a stack, it is automatically initialized to be null which represents an empty stack. However, sometimes we need to create an object with a specific initial value (necessary if it is a constant). We cannot do this by assigning in a general way as in

```
type T is limited ...
...
X: constant T := Y;       -- illegal, cannot copy value in variable Y
```

because this involves copying which is forbidden since the type is limited.

Two techniques are possible. One involves aggregates and the other uses functions. We will consider aggregates first. Suppose the type represents some sort of key with components giving the date of issue and the internal code number such as

```
type Key is limited
  record
    Issued: Date;
    Code: Integer;
  end record;
```

The type is limited so that keys cannot be copied. (They are a bit visible but we will come to that in a moment.) But we can write

```
K: Key := (Today, 27);
```

since, in the case of a limited type, this does not copy the value defined by the aggregate as a whole but rather the individual components are given the values Today and 27. In other words the value for K is built *in situ*.

It would be more realistic to make the type private and then of course we could not use an aggregate because the components would not be individually visible. Instead we can use a constructor function. Consider

```
package Key_Stuff is
  type Key is limited private;
  function Make_Key( ... ) return Key;
  ...
private
  type Key is limited
    record
      Issued: Date;
      Code: Integer;
    end record;
end Key_Stuff;

package body Key_Stuff is

  function Make_Key( ... ) return Key is
  begin
    return New_Key: Key do
      New_Key.Issued := Today;
      New_Key.Code := ... ;
    end return;
  end Make_Key;
  ...
end Key_Stuff;
```

The external client (for whom the type is private) can now write

```
My_Key: Key := Make_Key( ... );          -- no copying involved
```

where we assume that the parameters of Make_Key are used to compute the internal secret code.

It is worth carefully examining the function Make_Key. It has an extended return statement which starts by declaring the return object New_Key. When the result type is limited (as here) the return object is actually built in the final destination of the result of the call (such as the object My_Key). This is similar to the way in which the components of the aggregate were actually built *in situ* in the earlier example. So again no copying is involved.

The net outcome is that Ada provides a way of creating initial values for objects declared by clients and yet prevents the client from making copies. The limited type mechanism gives the provider of resources such as the keys considerable control over their use.

Controlled types

Ada provides a further mechanism for the safe management of objects through the use of controlled types. This enables us to write special code to be executed when

1) an object is created and,
2) when it ceases to exist and,
3) when it is copied if it is of a nonlimited type.

The mechanism is based on types called Controlled and Limited_Controlled declared in a predefined package thus

```
package Ada.Finalization is
   type Controlled is abstract tagged private;
   procedure Initialize(Object: in out Controlled) is null;
   procedure Adjust(Object: in out Controlled) is null;
   procedure Finalize(Object: in out Controlled) is null;

   type Limited_Controlled is abstract tagged limited private;
   procedure Initialize(Object: in out Limited_Controlled) is null;
   procedure Finalize(Object: in out Limited_Controlled) is null;
private
   ...
end Ada.Finalization;
```

The **is null** syntax, introduced in Ada 2005, makes the default behavior clear.

The central idea (for a nonlimited type) is that the user declares a type which is derived from Controlled and then provides overriding declarations of the three procedures Initialize, Adjust and Finalize. These procedures are called when an object is created, when it is copied, and when it ceases to exist, respectively. Note carefully that these calls are inserted automatically by the system and the programmer does not have to write explicit calls. The same mechanism applies to a limited type which has to be derived from Limited_Controlled but there is no procedure Adjust since copying is not permitted. These operations are typically used to provide complex initializations, deep copying of linked structures, storage reclamation at the end of the lifetime of an object, and other housekeeping activities that are specific to the type.

As an example, suppose we reconsider the stack and decide that we want to use the linked mechanism (so there is effectively no upper bound to the capacity of the stack) but wish to allow copying one stack to another. We can write

```
package Linked_Stacks is
  type Stack is private;
  procedure Clear(S: out Stack);
  procedure Push(S: in out Stack; X: in Float);
  procedure Pop(S: in out Stack; X: out Float);

private
  type Cell is
    record
      Next: access Cell;
      Value: Float;
    end record;

  type Stack is new Controlled with
    record
      Header: access Cell;
    end record;

  overriding
  procedure Adjust(S: in out Stack);
end Linked_Stacks;
```

The type Stack is now just private. The full type shows that it is actually a tagged type derived from the type Controlled and has a component Header which effectively is the stack in the previous formulation. In other words we have introduced a wrapper. Note that a client unit cannot see that the type is controlled and tagged. Since we want to make assignment work properly we have to override the procedure Adjust. Note also that we have supplied the overriding indicator so that the compiler can double check that Adjust does indeed have the correct parameters.

The package body might be

```
package body Linked_Stacks is

  procedure Clear(S: out Stack) is
  begin
    S := (Controlled with Header => null);
  end Clear;

  procedure Push(S: in out Stack; X: in Float) is
  begin
    S.Header := new Cell'(S.Header, X);
  end Push;
```

```
   procedure Pop(S: in out Stack; X: out Float) is
   begin
     X := S.Header.Value;
     S.Header := S.Header.Next;
   end Pop;

   function Clone(L: access Cell) return access Cell is
   begin
     if L = null then
       return null;
     else
       return new Cell'(Clone(L.Next), L.Value);
     end if;
   end Clone;

   procedure Adjust(S: in out Stack) is
   begin
     S.Header := Clone(S.Header);
   end Adjust;

end Linked_Stacks;
```

Assignment will now work properly. Suppose we write

```
This_One, That_One: Stack;
...
This_One := That_One;          -- calls Adjust automatically
```

The raw assignment of That_One to This_One copies just the record containing the component Header. The procedure Adjust is then called automatically with This_One as parameter. Adjust calls the recursive function Clone which actually makes the copy. This process is often called a deep copy. The result is that This_One and That_One now contain the same elements but are otherwise disjoint structures.

Another notable point is that the procedure Clear sets the parameter S to a record whose header component is null; the structure is known as an *extension aggregate*. The first part of the extension aggregate just gives the name of the parent type (or the value of an object of that type) and the part after **with** gives the values of the additional components, if any. The procedures Pop and Push are straightforward.

The reader might wonder about reclamation of unused storage when Pop removes an item and also when Clear sets a stack to empty. This will be discussed in the next chapter when we consider memory management in general.

Note that Initialize and Finalize are not overridden and thus inherit the null procedure of the type Controlled. So nothing special happens when a stack is

declared – this is correct since we just get a record whose Header is null by default and nothing else is required. Also nothing happens when an object of type Stack ceases to exist on exit from a procedure and so on – this again raises the issue of the reclamation of storage and will be addressed in the next chapter.

7 Safe Memory Management

The memory of the computer provides a vital part of the framework in which the program resides. The integrity of the memory contents is necessary for the health of the program. There is perhaps an analogy with human memory. If the memory is unreliable then the proper functioning of the person is seriously impaired.

There are two main problems with managing computer memory. One is that information can be lost by being improperly overwritten by other information. The other is that the memory itself can become filled and irrecoverable, so that no new information can be stored. This is the problem of memory leaks.

Memory leak is an insidious fault since it often does not show up for a long time. There was an example of a chemical control program that seemed to run flawlessly for several years. It was restarted every three months because of some external constraints (a crane had to be moved which necessitated stopping the plant). But the schedule for the crane changed and the program was then allowed to run for longer – it crashed after four months. There was a memory leak which slowly gnawed away at the free storage.

Buffer overflow

Buffer overflow is almost a generic term used to denote the violation of the security of information. Buffer overflow enables information to be overwritten or read mistakenly or maliciously.

This is a common fault with C and C++ programs and is typically caused by the absence of checks in those languages regarding writing or reading outside the bounds of an array. We illustrated this problem in the chapter on Safe Typing when discussing the example of throwing a pair of dice.

This problem cannot normally arise in Ada because there are checks that an array index does not lie outside the range of allowed values. These checks can be suppressed if we are absolutely sure that the program is perfect, but this is perhaps an unwise thing to do unless the program has been proved to be correct by analysis tools such as the SPARK Examiner mentioned in Chapter 11.

Although the absence of index checks is the ultimate cause of buffer overflow problems in C, it is exacerbated by other language features such as the choice of indicating the end of a string with a zero byte. This means that programmers have to test for this value (directly or indirectly) in many string manipulation routines. It is easy to make mistakes in performing such tests and

in any event the zero value might be accidentally overwritten itself. These secondary problems are often the key to loopholes which enable viruses to enter a system.

Another common way in which data can be accidentally destroyed is through the use of incorrect pointers. Pointers in C are treated as addresses and arithmetic can be performed on them. It is therefore easy for a pointer to have a miscomputed value and so to point to the wrong thing. Writing through the pointer then destroys some other data.

In the chapter on Safe Pointers we saw that Ada guards against this by applying strong typing to all pointers, and through the accessibility rules which ensure that declared objects do not vanish while being referenced by other objects.

Therefore, basic features of Ada guard against the accidental loss of data through overwriting memory. The remainder of this chapter addresses the issue of losing memory itself.

Heap control

Programming languages are typically implemented using three sorts of data storage

- global data that exists throughout the life of the program and can thus be allocated permanently (and often statically),
- data stored on a stack which grows and contracts as the flow of control passes through various subprograms,
- data allocated in a heap and used and discarded in a manner not directly tied to the flow of control.

Fortran global common is the primeval example of global static storage (this relates to Fortran as it was in the early days of programming). But global static storage exists in all languages. In Ada if we declared

```
package Calendar_Data is
  type Month is (Jan, Feb, Mar, ... , Nov, Dec);
  Days_In_Month: array (Month) of Integer :=
    (Jan => 31, Feb => 28, Mar => 31, Apr => 30,
     May => 31, Jun => 30, Jul => 31, Aug => 31,
     Sep => 30, Oct => 31, Nov => 30, Dec => 31);
end;
```

then storage for the array Days_In_Month would naturally be declared in fixed global storage.

The stack is an important storage structure in all modern programming languages. Note that we are here talking about the underlying stack used by the implementation and not an object of the type Stack used for illustration in an earlier chapter. The stack is used for parameter passing in subprogram calls (actual parameters, the return address, saved registers, and so on) as well as for local variables within a subprogram. In a multitasking program where several threads of activity occur in parallel, each task has its own stack.

Now consider the function Nfv_2000 used in the program for interest rates in the chapter on Safe Pointers

```
function Nfv_2000 (X: Float) return Float is
  Factor: constant Float := 1.0 + X/100.0;
begin
  return 1000.0 * Factor**2 + 500.0 * Factor – 2000.0;
end Nfv;
```

The object Factor will typically be stored in the stack. It will come into existence when the function is called and will cease to exist when the function returns. This is all managed safely and automatically by the call/return mechanism. Note that although Factor is marked as a constant nevertheless it is not static since each call of the function will provide a different value for it. Moreover, the function might be called by two different tasks at the same time in a multitasking program and so Factor certainly cannot be stored globally.

The values of any parameters such as X are also stored on the stack.

Now consider a more elaborate subprogram which declares a local array whose size is not known until the program executes – consider for example a function to return an arbitrary array in reverse order. In Ada we might write

```
function Rev(A: Vector) return Vector is
  Result: Vector(A'Range);
begin
  for K in A'Range loop
    Result(K) := A(A'First+A'Last–K);
  end loop;
  return Result;
end Rev;
```

where Vector is declared as an unconstrained array type

```
type Vector is array (Natural range <>) of Float;
```

As explained in the Arrays and constraints section of the chapter on Safe Typing, this notation indicates that Vector is an array type but the bounds of different objects may be different. When we declare an actual object of the type Vector we must supply bounds. So we might have

```
L: Integer := ... ;
My_Vector, Your_Vector: Vector(1 .. L);          -- L need not be static
...
Your_Vector := Rev(My_Vector);
```

In most programming languages we would be forced to place an object such as the local variable Result on the heap rather than the stack because its size is not known until the program executes. This is certainly not necessary because a stack is flexible and storage for local variables can always be managed on a last-in–last-out basis. But the heap is often used for simplicity of implementation. It requires a bit of thought to design and manage dynamically sized data efficiently, and without care the subroutine calling mechanism can suffer a loss of performance.

Although this is not required by the language standard, production-quality implementations of Ada always use the stack for local data – an efficient technique is to use both ends of the stack, one end for return links and fixed local data and the other end for dynamically sized local data. This enables the location of return addresses to be computed more efficiently and yet keeps full flexibility. Furthermore, Ada systems usually guard against the stack running out of storage and raise the exception Storage_Error if it does (or rather if it is about to).

The above example illustrates a number of nice points about Ada. By contrast it is quite tricky to write in C. This is because C has no proper abstraction for arrays and so we cannot pass an array as a parameter but only a pointer to an array. Moreover, C cannot return an array object as a result. We could of course simply declare a function that reverses the argument *in situ* and leave it to the user to make a copy first. But doing the reverse *in situ* is tricky since we have to take care not to destroy the values as we swap them. So perhaps it is best to pass pointers to both the original array and the result as distinct parameters. The other difficulty is that C does not know how long its arrays are and so we have to pass the length of the array as well (or maybe the upper bound). This is yet another hazard since it is all too easy to pass a length that does not correspond to that of the array. So we might have

```
void rev(float *a, float *result, int length);
{
  for (k=0; k<length; k++)
    result[k] = a[length–k–1];
}
...
float my_vector[100], your_vector[100];
...
rev(my_vector, your_vector, 100);
```

Although this chapter is meant to be about storage management it is perhaps worth pausing to list some of the risks and difficulties in the above C code.

- Arrays in C always have lower bound 0 and so if the application has a different natural lower bound such as 1 then confusion can arise. Ada allows any lower bound.
- The length of the array has to be passed separately, there is a risk of getting the length wrong and confusing the length with the upper bound. In Ada the attributes of the array are passed as part of the array itself.
- The address of the result array has to be passed separately. There is the danger of confusing the two arrays which cannot happen in Ada because the assignment clarifies which is which.
- The loop has to be written out explicitly whereas the Ada notation ties it to the range of the array automatically.

However, we have strayed from the topic. The key point is that if we did declare a local array in C++ whose size was not static as in

```
void f(int n, ... );
{   float a[] = new float [n];
...
}
```

then the array a will be placed in the heap and not on the stack. In C we would have to use malloc which does explicitly reveal the use of the heap.

The general danger of using the heap is that storage might be deallocated when it is still in use or left allocated when it is not needed. Because Ada allows dynamically sized objects on the stack, the heap is basically only used when allocators are invoked as mentioned in the chapter on Safe Pointers. This results in better performance and less chance of memory leaks.

Storage pools

We now turn to the use of the heap in Ada. The proper term is storage pool. If we do an allocation such as in the procedure Push discussed in the chapter on Safe Object Construction thus

```
procedure Push(S: in out Stack; X: in Float) is
begin
  S := new Cell'(S, X);
end Push;
```

then the space for the new Cell will be taken from a storage pool. There is always a standard storage pool but we can declare and manage our own storage pools as well.

LISP was the first language to take storage management out of the hands of the programmer, and to incorporate a garbage collector in order to reclaim storage. This approach is used in a number of other languages including Python and Java. The presence of a garbage collector simplifies programming substantially, but has its own problems. For example, the garbage collector may interrupt the execution of the program at unpredictable times, which presents a significant technical challenge in a real-time environment. A programmer of a real-time system must retain fine control over memory and deallocation and must be able to guarantee predictability of execution time, which can be compromised by a garbage collector. Indeed, one of the reasons that the original Ada 83 design did not include full support for OOP was that the language designer, Jean Ichbiah, who was one of the early implementers of the Object-Oriented language Simula, felt that OOP required a garbage collector for effective storage reclamation, and that this was completely inappropriate for a language intended for high-integrity real-time applications. As was eventually demonstrated by C++ and Ada 95, a language can effectively support OOP without a garbage collector as long as it is sufficiently powerful so that the application programmer can express the necessary storage management control.

Ada provides the user with a choice of mechanisms. Storage control can be done

- by hand. That is by programming the release of storage on an individual basis.
- by using storage pools. Individual items can be deleted from a specific pool and the whole pool can be discarded when no longer required.
- by a garbage collector. This might not be available in all implementations.

In order to return a lump of storage that is no longer used we call an instantiation of a predefined generic procedure named Unchecked_Deallocation. In order to do this we have to use a named access type so we will suppose that the type Cell is declared by

```
type Cell;
type Cell_Ptr is access all Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Value: Float;
  end record;
```

Note that we have an intrinsic circularity here which is broken by first giving an incomplete declaration of the type Cell. We now write

```
procedure Free is new Unchecked_Deallocation(Cell, Cell_Ptr);
```

In order to deallocate storage we simply call the procedure Free with an access value referring to the storage concerned. Thus the procedure Pop should now be written as

```
procedure Pop(S: in out Stack; X: out Float) is
  Old_S: Stack := S;
begin
  X := S.Value;
  S := S.Next;
  Free(Old_S);
end Pop;
```

Note that we are here using the version of the type Stack that is limited private and not the version that is controlled.

It might seem that the use of Free runs the risk of a dangling reference since in general there might be another access value pointing to the deallocated storage. But in this example the user's view of the type is limited and so the user cannot have made a copy of the structure. Moreover, the user cannot see the details of the type Stack and in particular cannot see the types Cell and Cell_Ptr at all and therefore cannot call Free. Thus once we have assured ourselves that Pop is correct then no trouble is possible. Finally, the instantiation of Unchecked_Deallocation provides a cross-check by requiring the use of named access types and thus checks that the parameters match.

We must also change Clear as well. The easy way is to write

```
procedure Clear(S: in out Stack) is
  Junk: Float;
begin
  while S /= null loop
    Pop(S, Junk);
  end loop;
end Clear;
```

Although this technique ensures that storage is deallocated properly whenever Pop and Clear are called, there is still the risk that the user might declare a stack and leave its scope when it is not empty. Thus

```
procedure Do_Something ...
  A_Stack: Stack;
begin
  ...                    -- play with A_Stack
  ...                    -- is it empty as we leave?
end Do_Something;
```

If A_Stack were not null when Do_Something returns then the storage would be lost. We cannot leave the onus on the user to take care not to lose storage so we should make the stack a controlled type as illustrated at the end of the chapter on Safe Object Construction. We can then declare our own procedure Finalize perhaps simply as

```
overriding
procedure Finalize(S: in out Stack) is
begin
  Clear(S);
end Finalize;
```

Note the use of the overriding indicator just to ensure that we have not misspelled Finalize or mistyped its formal parameters.

Ada also permits users to declare their own storage pools. This is straightforward but would take too much space to explain in detail here. But the general idea is that there is a predefined type Root_Storage_Pool (which itself is a limited controlled type) and we can declare our own storage pool type by deriving from it thus

```
type My_Pool_Type(Size: Storage_Count) is
                  new Root_Storage_Pool with private;
overriding
procedure Allocate( ... );
overriding
procedure Deallocate( ... );
-- also overriding Initialize( ... ) and Finalize( ... );
```

The procedure Allocate is automatically called when a new object is allocated by an allocator and Deallocate is automatically called when an object is discarded by calling an instance of Unchecked_Deallocation such as Free. The user then writes appropriate code to manage the pool as desired. Since a pool type is also controlled the procedures Initialize and Finalize are automatically called when the whole pool is declared and finally goes out of scope.

In order to create a pool we then declare a pool object in the usual way. And finally we can link a particular access type to use the pool.

```
Cell_Ptr_Pool: My_Pool_Type(1000);          -- pool size is 1000
for Cell_Ptr'Storage_Pool use Cell_Ptr_Pool;
```

An important advantage of declaring our own pools is that the risk of fragmentation can be minimized by ensuring a constant "block size" for each allocated object (e.g., by keeping different types in different pools and avoiding allocating objects of unconstrained types). Moreover, we can write our own storage allocation mechanisms and even do some storage compaction if we so wish. A further point is that if the access type concerned is declared locally then the pool can be local as well and will automatically be discarded so that there can be no possibility of storage being lost.

Storage pool control has been enhanced in Ada 2012 with the introduction of subpools. This topic is beyond the scope of this chapter, but we will simply point out that a subpool is a separately reclaimable part of a storage pool.

Finally, there is a safeguard against misuse of Unchecked_Deallocation and that is that since it is a predefined library unit, any unit we write that calls it will have

```
with Unchecked_Deallocation;
```

written boldly at the start of the text. This will then be clearly visible to anyone reviewing the program and especially to our Manager.

Restrictions

There is a general mechanism for ensuring that we do not use certain features of the language, and that is the pragma Restrictions. Thus if we write

```
pragma Restrictions(No_Dependence => Unchecked_Deallocation);
```

then we are asserting that the program does not use Unchecked_Deallocation at all – the compiler will reject the program if this is not true.

There are over fifty such restrictions which can be used to give assurance about various aspects of the program. Many are rather specialized and relate to multitasking programs. Others which concern storage generally and are thus relevant to this chapter are

```
pragma Restrictions(No_Allocators);
pragma Restrictions(No_Implicit_Heap_Allocations);
```

The first completely prevents the use of the allocator **new** as in **new** Cell'(...) and thus all explicit use of the heap. Just occasionally some implementations

might use the heap temporarily for objects in certain awkward circumstances. This is rare and can be prevented by the second pragma.

8 Safe Startup

We can carefully write a program so that it behaves properly when running, but it is all to no avail if it will not start properly.

The motor car that will not start is no good even if when going it behaves like a Rolls-Royce.

In the case of a computer program, the key things are to ensure that data is initialized properly and this often means to ensure that its various components are initialized in the correct order.

Elaboration

A program typically consists of a number of library packages P, Q, R and so on, plus a main subprogram M. The general idea is that when the program is started the various packages are *elaborated*, after which the main subprogram is called. The elaboration of a package consists of the creation of the various entities declared at the top level in the package – but not entities declared within subprograms in the package because these are only created when the subprograms are called.

Thus consider again the package Stack in the chapter on Safe Architecture. In outline it was

```
package Stack is
  procedure Clear;
  procedure Push(X: Float);
  function Pop return Float;
end Stack;

package body Stack is

  Max: constant := 100;
  Top: Integer range 0 .. Max := 0;
  A: array (1 .. Max) of Float;

  ... -- procedures Clear and Push and function Pop

end Stack;
```

The elaboration of the specification of the package does nothing in this case because there are no objects declared in it. The elaboration of the body of the package notionally causes the space for the integer Top and the array A to be reserved. In this particular case the size of the array is known before the

program executes because it is given by the constant Max which happens to have a static value and so the storage can be effectively reserved even before the program is loaded.

But Max need not have had a static value – it might have been given the result of some function call thus

```
Max: constant := Some_Function;
Top: Integer range 0 .. Max := 0;
A: array (1 .. Max) of Float;
```

and then the space required for A would be computed as part of the elaboration of the package body. If we had been careless and declared Max as a variable and forgotten to give it an initial value thus

```
Max: Integer;
Top: Integer range 0 .. Max := 0;
A: array (1 .. Max) of Float;
```

then the size of the array would be given by the value that Max happened to have. If Max were negative then the attempt to declare the array would raise Constraint_Error and if Max were too large than it might raise Storage_Error. (Note that most compilers will issue a warning in such a situation, since straightforward data flow analysis reveals the reference to an uninitialized variable.)

It should also be noted that we gave an initial value of zero to the variable Top so that the user did not have to call the procedure Clear before calling Push or Pop.

Alternatively we can give the package body an explicit initialization part so that it becomes

```
package body Stack is

  Max: constant := 100;
  Top: Integer range 0 .. Max;
  A: array (1 .. Max) of Float;

  ... -- procedures Clear and Push and function Pop

begin                              -- initialization part
  Top := 0;
end Stack;
```

The initialization part can contain any statements at all. It is executed as part of the elaboration of the package body and so before any of the subprograms in the package can be called by code outside the package.

Readers might feel that it is surely always best to give all variables an initial value anyway "just in case". In the example given here the value zero is indeed a sensible initial value and corresponds to a call of Clear. In some situations there is no obvious initial value and giving a value "just in case" is not always wise because it can actually obscure real errors. We will come back to this briefly when we discuss SPARK in the final chapter.

In the case of numeric variables, the consequences of using a value that has not been set are not necessarily disastrous. But the consequence of using an access value or some other implicit address which has not been set could be. In the case of access types in Ada these either have a default value of **null** or must be initialized as we have seen.

A related kind of potential error concerns "access before elaboration". This means attempting to use something before it has been properly elaborated. Consider

```
package P is
  function F return Integer;
  X: Integer := F;                -- raises Program_Error
end;
```

where the body of F is of course in the body of the package P. We cannot successfully call F to give an initial value to X before the body has been elaborated, since the body of F may reference X or variables declared after X that themselves have not yet been initialized. So Ada requires the exception Program_Error to be raised. The same sort of error in C could have unpredictable effects.

Elaboration pragmas

Within a single compilation unit the rule is that declarations are elaborated in the order in which they appear in the text.

In the case of a program linked from several different units, a unit is always elaborated after all those on which it depends. Thus a body is elaborated after the corresponding specification, the specification of a child is elaborated after the specification of its parent, and any unit is elaborated after the specifications of all those mentioned in a (nonlimited) with clause.

However, this only partially dictates the order and is sometimes not enough to ensure the correct behavior of the program. We can extend the example above as follows

```
package P is
  function F return Integer;
end P;

package body P is
  N: Integer := some_initial_value;
  function F return Integer is
  begin
    return N;
  end F;
end P;

with P;
package Q is
  X: Integer := P.F;
end;
```

It is important that the body of P is elaborated before the specification of Q is elaborated because otherwise the call of F would return some unknown value for N. Elaborating the body of P ensures that N is correctly initialized to *some_initial_value*. But this initial value could itself invoke functions from other packages, and those functions might be referring to data that are initialized in the bodies of the packages containing the functions. So, we need to ensure not only that the body of P is elaborated before the specification of Q, but also that the bodies of all units on which P depends are likewise elaborated before the specification of Q is elaborated. But the above rules, with elaboration order simply constrained by the dependence relation between units, do not ensure this and Program_Error might be raised at run time.

We can force the required order of elaboration by inserting a pragma in the context clause for Q thus

```
with P;
pragma Elaborate_All(P);
package Q is
  X: Integer := P.F;
end;
```

Note that the All in Elaborate_All indicates the transitive nature of the pragma. Its effect is that at run time the elaboration code for package P (and all the packages on which it depends) will be executed before the elaboration code for Q.

There is also a pragma Elaborate_Body which can be given with a specification and indicates that its body must be elaborated immediately after the specification. In many cases this pragma will be sufficient to prevent elaboration order problems.

At this point the reader might be wondering if the elaboration order problems could be solved by a simple language rule, for example to behave as if pragma Elaborate_Body were applied everywhere. Unfortunately, this does not work, since it would make illegal the common case of mutually recursive packages (that is, where the body of package P1 *with*s package P2, and the body of package P2 *with*s P1).

Elaboration order issues can be troublesome, especially in large systems, and sometimes these problems are better avoided than solved. One approach is to eliminate all initialization code in package bodies, and instead to have the main program call explicit procedures to set up the data structures that the program will be manipulating.

Dynamic loading

A related topic concerns dynamic loading. Some languages are designed to create a single coherent program that is fully linked and loaded before being run. Ada, C and Pascal are like that. The operating system may swap lumps of the program in and out of memory using paging algorithms but that is an implementation detail.

Other languages are designed to be much more dynamic and enable new code to be compiled, loaded and executed while the program is running. COBOL, Java, and C# are like that.

An approach used with programs written in languages such as C is to use dynamic linked libraries (DLLs) whereby an indirect call is used to invoke the new code. But this is not safe since there is no checking that the parameters of the new code match those of the old calling sequence.

One approach that can be used with Ada is to use the dispatching mechanism as the hook to dynamic loading. The point about dispatching is that it enables existing compiled code containing a class (such as Geometry.Object'Class) to call operations (such as Area) of further types (such as Pentagon, Hexagon and so on) without the central code having to be recompiled. This was briefly mentioned in the chapter on Safe Object-Oriented Programming. Moreover, the mechanism is completely type safe.

A good example of how dynamic loading can be added within this framework is given in [7].

9 Safe Communication

A program that doesn't communicate with the outside world in some way is useless although very safe. Such a program might almost be in solitary confinement. A prisoner in solitary confinement is safe in the sense that he cannot hurt other people but he is equally of no use to society either.

So for a program to be useful it must communicate. And if the program is written in a safe way so that it does not have internal dangers, it is largely futile if its communication with the world is unsafe. So safety in communication is important since it is here that the program truly has a useful effect.

It is perhaps worth recalling from the introduction that we characterized the difference between safety-critical and security-critical systems as that the former is where the program must not harm the world whereas the latter is where the world must not harm the program. So communication is the ultimate lynchpin of both safety and security.

Representation of data

An important aspect of communication concerns the mapping between the abstract software and the actual hardware. Most languages leave this sort of thing to individual implementations. But Ada gives the user quite specific control over many aspects of data representation.

For example we might decide that we want data in a record to be laid out in a particular manner – perhaps to match that of an existing file structure. Suppose the record is the type Key in the chapter on Safe Object Construction

```
type Key is limited
  record
    Issued: Date;
    Code: Integer;
  end record;
```

where the type Date is

```
type Date is
  record
    Day: Integer range 1 .. 31;
    Month: Integer range 1 .. 12;
    Year: Integer;
  end record;
```

We will assume that we are using a 32-bit machine with four bytes to a word. The day and month easily fit into one byte each because of their explicit range constraints, and the year needs at most 16 bits (we will ignore "Year 32768" problems) so the whole date can be neatly packed into a single word. We can express this by

```
for Date use
  record
    Day at 0 range 0 .. 7;
    Month at 1 range 0 .. 7;
    Year at 2 range 0 .. 15;
  end record;
```

In the case of the type Key, the required structure is simply two words and almost inevitably the implementation will use the representation we require. But we can ensure this by writing a record representation clause

```
for Key use
  record
    Issued at 0 range 0 .. 31;
    Code at 4 range 0 .. 31;
  end record;
```

As another example consider the type Signal of the chapter on Safe Typing. It was

```
type Signal is (Danger, Caution, Clear);
```

Unless we say otherwise, the compiler will encode this type using 0 for Danger, 1 for Caution and 2 for Clear. But in a real application the value of the signal might enter the program encoded as 1 for Danger, 2 for Caution and 4 for Clear. We can instruct the program to use this encoding by writing an enumeration representation clause

```
for Signal use (Danger => 1, Caution => 2, Clear => 4);
```

Note that the **for** keyword has nothing to do with the **for** statement; the keyword was chosen in the interest of program readability.

Furthermore, suppose we would like to ensure that each object of type Signal is stored in one byte; this is especially relevant for components of arrays and records. We can achieve this effect by supplying a representation attribute as follows:

```
for Signal'Size use 8;
```

Size can be specified equivalently in Ada 2012 by supplying an aspect specification with the associated declaration

```
type Signal is (Danger, Caution, Clear)
  with Size => 8;
```

Continuing this example, suppose we have a variable

```
The_Signal : Signal;
```

which we want to ensure is autonomously loaded into the program at the hexadecimal address 0ACE. We can arrange this with an address clause

```
for The_Signal'Address
use System.Storage_Elements.To_Address(16#0ACE#);
```

The value specified after **use** must be of type System.Address, which is typically a private type. Thus the To_Address function from the predefined package System.Storage_Elements is invoked to convert the integer value to type Address.

Equivalently, in Ada 2012 we can write the associated declaration with an Address aspect specification thus

```
The_Signal : Signal
  with Address => System.Storage_Elements.To_Address(16#0ACE#);
```

Validity of data

An important part of all programming is to ensure that data received from the outside world is valid. In most case we can simply program various checks using normal programming techniques. But sometimes this is awkward.

The type Signal is a case in point. We have instructed the compiler to hold the value as an enumeration type with a certain representation. If by some misfortune a value turns up which does not have a recognized pattern (perhaps two bits are set because of a transient in the external device) then we cannot express a test of that in the normal way because that would take us outside the domain of definition of the type Signal. Instead we can write

```
if not The_Signal'Valid then ...
```

The Valid attribute may be applied to any scalar object; it returns a Boolean result that is True if the object contains a value that is consistent with its subtype and is False otherwise.

Another approach is to use Unchecked_Conversion. We can read the value in, perhaps as a byte, check it and then if it is acceptable, convert it to the type Signal. First we need the type Byte and the conversion routine

```
type Byte is range 0 .. 255;
for Byte'Size use 8;
-- or in Ada 2012:
--   type Byte is range 0..255
--      with Size => 8;

function Byte_To_Signal is new Unchecked_Conversion(Byte, Signal);
```

and then

```
Raw_Signal: Byte;
for Raw_Signal'Address use To_Address(16#0ACE#);
-- or in Ada 2012:
--   Raw_Signal: Byte
--     with Address => To_Address(16#0ACE#);
The_Signal: Signal;

case Raw_Signal is
  when 1 | 2 | 4  =>
                                  -- raw value OK, convert it
    The_Signal := Byte_To_Signal(Raw_Signal);
    ...                           -- process valid value
  when others =>
    ...                           -- raw value invalid
    ...                           -- take corrective action
end case;
```

The idea of course is that since the type Byte is simply an integer type we can do normal arithmetic on the value in order to check it. The corrective action might include logging the particular invalid value and so on.

The reader should note a flaw in the above if the value truly is loaded autonomously. Between checking and the conversion, a new value might arrive. So it should be copied into a local variable before being tested and processed.

Communication with other languages

Many modern large systems are written in a mixture of languages each appropriate to the part of the system concerned. The safety-critical control routines and security-critical input routines might be written in Ada (perhaps in SPARK), the GUI interface might be written in C++, some complex mathematical analysis might be written in Fortran, some device drivers might be in C and so on.

Many languages have some facilities for interworking with other languages (C++ with C for example) but these are often loosely defined. Ada is perhaps unique in providing well-defined mechanisms within the language standard for interfacing to programs in other languages in general. Ada provides specific facilities for communication with programs and data in C, C++, Fortran and COBOL. In particular, Ada recognizes the representation of types in these other languages such as the arrangement of matrices in Fortran and strings in C so that communication retains type safety.

In a mixed language situation it is thus a good idea to use Ada as the central language so that communication with other languages has the benefit of the type checking provided by the Ada conversion routines.

The general means of communication uses pragmas. Thus suppose we have a C routine called next_int and we wish to call it from our Ada program as the function Next_Int. We simply write

```
function Next_Int return Interfaces.C.int;
pragma Import(C, Next_Int);
```

The pragma indicates that the calling convention is C and also tells the compiler that there is no Ada body for this function. The pragma can supply a different external name and link name if necessary. The predefined package Interfaces.C contains declarations for types that may be used in an Ada program to match the various primitive types in C. By using the types in this package the Ada programmer does not have to know how the various Ada types correspond to the C types.

Similarly, if we wish the external C program to call the Ada procedure Action that takes two C ints then we can make the name of the Ada procedure available externally by writing

```
procedure Action(X, Y: in Interfaces.C.int);
pragma Export(C, Action);
```

Access-to-subprogram types are important for communication with other languages especially when programming interactive systems. For example, suppose we want the procedure Action to be called by the GUI when the mouse is clicked. Suppose that there is a C function set_click that takes the address of the routine to be called when the mouse is clicked. We can express this in Ada as follows

```
type Response is access procedure (X, Y: in Interfaces.C.int);
pragma Convention(C, Response);

procedure Set_Click(P: in Response);
pragma Import(C, Set_Click);
```

```
procedure Action(X, Y: in Interfaces.C.int);
pragma Convention(C, Action);
...
Set_Click(Action'Access);
```

In this case we have not made the name of the procedure Action visible to the C program because it is called indirectly but we do have to ensure that it uses the C calling convention.

Streams

A potential difficulty occurs when we transmit values of different types to and from the external world. Output is straightforward because we know the type of the value being transmitted and can use the appropriate format. But input is a problem because typically we do not know what is coming. If a file is uniform and all values are of the same type then we simply have to ensure that we have connected to the correct file. The real difficulty arises when values of different types are involved in the same file. Ada has a number of different filing mechanisms, some are for homogeneous files such as files of all integers or text files; for heterogeneous files we use a stream file.

As a very simple example suppose a file is to have a mixture of values of types Integer, Float and Signal. All types have special attributes 'Read and 'Write for use with streams. On output we simply write

```
S: Stream_Access := Stream(The_File);
...
Integer'Write(S, An_Integer);
Float'Write(S, A_Float);
Signal'Write(S, A_Signal);
```

and this results in a mixture of values of different types on The_File. In the space available we cannot give the full details but S identifies the stream associated with the file.

On input we simply do the reverse

```
Integer'Read(S, An_Integer);
Float'Read(S, A_Float);
Signal'Read(S, A_Signal);
```

If we do the calls in the wrong order then the exception Data_Error will be raised if the bit pattern that is read is not a valid value for the type concerned.

If we do not know the order in which things are to be read then we need to create a class to cover all the different types involved. In this simple case we might declare a root type

```
type Root is abstract tagged null record;
```

to act as a sort of wrapper and then a series of individual types to encapsulate the real data thus

```
type S_Integer is new Root with
  record
    Value: Integer;
  end record;

type S_Float is new Root with
  record
    Value: Float;
  end record;
...
```

and so on. On output we write

```
Root'Class'Output(S, (Root with An_Integer));
Root'Class'Output(S, (Root with A_Float));
Root'Class'Output(S, (Root with A_Signal));
```

Note that the same procedure is used for all the calls. It first outputs the value of the tag of the specific type and then calls (by dispatching) the appropriate Write attribute.

For input we might write

```
Next_Item: Root'Class := Root'Class'Input(S);
...
Process(Next_Item);
```

The procedure Root'Class'Input reads the tag from the stream and then dispatches to the Read attribute to read the item and finally assigns it as the initial value of the object Next_Item. We can then call some other procedure such as Process by dispatching to do whatever we want. We might assign the value to a particular variable according to its type.

To do this we first declare the abstract procedure for the root type thus

```
procedure Process(X: in Root) is abstract;
```

and then specific procedures such as

```
overriding
procedure Process(X: S_Integer) is
begin
  An_Integer := X.Value;                    -- extract value from wrapper
end Process;
```

The procedure Process could of course do anything we like with the value concerned.

This has been a somewhat artificial example. Its purpose has been to illustrate that Ada can process items of various types in a way that preserves the security of the type model.

Object factories

We have just seen how the predefined stream mechanism enables us to manipulate values whose types are not known until they are input in some way. The underlying mechanism of reading a tag and then creating an object of the appropriate type is also available to the user in Ada 2005.

Suppose we are manipulating the geometrical objects discussed in the chapter on Safe Object-Oriented Programming. These are of various types such as Circle, Square, Triangle and so on and are all derived from the root type Geometry.Object. We might wish to read values of these objects from a keyboard. For a circle we would expect the values of its two coordinates followed by the radius. For a triangle we would expect the two coordinates plus the values of the three sides and so on. We could declare functions Get_Object to read these values such as

```
function Get_Object return Circle is
begin
  return C: Circle do
    Get(C.X_Coord);  Get(C.Y_Coord);  Get(C.Radius);
  end return;
end Get_Object;
```

The internal calls of Get are calls of predefined procedures to read simple values from the keyboard. The user will have to type some code to indicate which type of object is being supplied. Perhaps the values for a circle could be preceded with the string “Circle”; we will also suppose that we have written a simple function Get_String to read and return such a string.

So now all we have to do is to read the code string, and then call the appropriate procedure Get_Object to create an object of the correct type. The

key to this is to use a predefined generic function which, given a tag, returns an object of the corresponding type. In essence it is

```
generic
  type T(<>) is abstract tagged limited private;
  with function Constructor return T is abstract;
function Generic_Dispatching_Constructor(The_Tag: Tag) return T'Class;
```

This generic function has two generic parameters, the first identifies the class of types concerned (such as Geometry.Object from which the types Circle, Square and Triangle are derived) and a dispatching operation to make objects of the specific types (such as functions Get_Object).

We can now instantiate this generic function to give a constructor function for geometrical objects

```
function Make_Object is
        new Generic_Dispatching_Constructor(Object, Get_Object);
```

A call of Make_Object takes the tag of the specific type concerned, then dispatches to the appropriate function Get_Object and finally returns the value created.

We might decide to declare an access variable to refer to the newly created object thus

```
Object_Ptr: access Object'Class;
```

If the tag value is in a variable Object_Tag (of the type Tag which is defined in the predefined language package Ada.Tags – the generic constructor function is also in this package), then we call Make_Object thus

```
Object_Ptr := new Object'(Make_Object(Object_Tag));
```

and now we have made the new object (perhaps a circle) with the values of its coordinates and radius which were read from the keyboard.

We are not quite finished since we have to convert the string "Circle" which identifies the type concerned into the tag value used for dispatching. A simple way to do this is to write

```
for Circle'External_Tag use "Circle";
for Triangle'External_Tag use "Triangle";
```

and then we can read and convert the external string into the internal tag value by

```
Object_Tag: Tag := Internal_Tag(Get_String);
```

There is of course no need to declare the variable Object_Tag since we can combine the operations into one single statement thus.

```
Object_Ptr := new Object'(Make_Object(Internal_Tag(Get_String));
```

Finally, it should be noted that the above discussion has been slightly simplified. The actual constructor has an auxiliary parameter which we have ignored.

10 Safe Concurrency

In real life many activities happen in parallel. Human beings do things in parallel with considerable ease. Females seem to do this better than males – perhaps because they have to rock the baby while cooking the food and keeping the tiger out of the cave. The male typically just concentrates on one thing at a time such as catching that rabbit for dinner – or trying to find a bigger cave or perhaps even inventing a wheel.

Computers traditionally only do one thing at a time, and the operating system makes it look as if several things are going on in parallel. This is not quite so true these days, since many computers do truly have multiple processors or multiple cores but it still does apply to the vast majority of small computers including those used in process control.

Operating systems and tasks

Operating systems vary enormously in the amount of parallel activity that they permit. Operating systems supporting POSIX provide the programmer with multiple threads of control. These various threads of control can flow through the program quite independently and so support parallel activities.

On some hardware there will only be one processor, which will be allocated to the different threads according to some scheduling algorithm. One approach is simply to give the processor to each thread in turn for a small amount of time; more sophisticated approaches (especially for systems with real-time requirements) are to use priorities or deadlines to ensure that the processor is used effectively.

Some hardware might have multiple processors in which case several threads can truly be active in parallel. Again a scheduler will allocate the processors in a hopefully effective way to the active threads of control.

In a programming language the concurrent activities are generally called *threads* or *tasks*. Here we will use the latter which is the Ada term. Languages take very different approaches to tasking. Some languages have intrinsic facilities for tasking built into the language itself. Others provide simple access to the underlying primitives of the operating system. Yet others ignore the subject completely.

Ada, Java, C#, and most recently C++ are languages with intrinsic tasking facilities. C has no built-in support for tasking, so programmers using C need to rely on third-party libraries or make direct calls to operating system services.

There are at least three advantages of having tasking within the language itself

- Built-in syntactic constructions make it much easier to write correct programs because the language can prevent a number of errors from being made. It is essentially the old story about abstraction. By hiding low-level details certain errors are prevented.
- Portability is difficult if operating system facilities are used directly because they vary widely from system to system.
- General operating systems do not provide the range of timing and related facilities needed by many real-time applications.

The operations typically required in a tasking program are

- Tasks must be prevented from violating the integrity of data if several tasks need access to the data concurrently.
- Tasks need to communicate with each other in order to transfer data between them.
- Tasks need to be controlled in order to meet specific timing requirements.
- Tasks need to be scheduled in order to use resources efficiently and to meet their overall deadlines.

This chapter will briefly look at these topics and illustrate how Ada addresses them in a reliable manner. This is a design challenge, since programs with tasking are much harder to write correctly than ordinary sequential programs. But first we introduce the simple idea of an Ada task and the overall program structure.

An Ada program can have many tasks running concurrently. A task is written in two parts rather like a package. It has a specification which describes the interface it presents to other tasks and a body which contains the code saying what it actually does. In simple cases the specification simply names the task so we might have

```
task A;                          -- task specification

task body A is                   -- task body
begin
   ...                           -- statements saying what the task does
end A;
```

Sometimes it is convenient to have several similar tasks in which case we can introduce a task type

```
task type Worker;

task body Worker is ...
```

We can then declare several tasks by declaring objects in the usual way

```
Tom, Dick, Harry: Worker;
```

This creates three tasks called Tom, Dick and Harry. We can also declare arrays of tasks and have task components inside records and so on. Tasks can be declared wherever other objects can be declared such as in a package or in a subprogram or even within another task. Not surprisingly, task types are limited types, since assigning one task to another is not a meaningful operation.

The main subprogram of a complete program is invoked by the so-called *environment task* and it is this environment task that elaborates library packages, as described in the chapter on Safe Startup. An overall program with library packages A, B and C and main subprogram Main can therefore be thought of as

```
task Environment_Task;

task body Environment_Task is
  ...              -- declarations of library packages A, B, C
  ...              -- and main subprogram Main
begin
  ...              -- call of main subprogram Main
end;
```

A task becomes active simply by being declared. It finishes by reaching the end of the task body. An important rule is that a local task (that is, a task declared within a subprogram, block, or another task) must finish before the enclosing unit can itself be left and the enclosing unit will be suspended until the local task terminates. This rule prevents dangling references to data that no longer exists.

Protected objects

Suppose that the three tasks Tom, Dick and Harry are using a stack as some sort of temporary storage device. From time to time one of them pushes an item onto the stack and from time to time one of them (perhaps the same one, perhaps a different one) pops an item off the stack.

The three tasks run concurrently – possibly with actual parallelism on multiple processors/cores, or else under the control of a task scheduler on a single processor where the run-time system allocates the processor according to some algorithm. Let's assume the latter; perhaps each task gets 10 ms in turn with a "round robin" task dispatcher.

Suppose the stack they are using is as declared in the chapter on Safe Architecture. Suppose that Harry is calling Push when his time slot expires and control then passes to Tom who calls Pop. To be precise, suppose Harry loses the processor just after he has executed the statement to increment Top in

```
procedure Push(X: Float) is
begin
  Top := Top + 1;                -- Harry loses processor just after this
  A(Top) := X;
end Push;
```

At this point Top has been incremented but the new value X has not been assigned to the component of the array. When Tom calls Pop, he gets the old and possibly meaningless value in the array component that was about to be overwritten by the new value. When Harry gets the processor back (and assuming no other stack activity occurs meanwhile) he will write the value X into a component of the array that is a part of the stack that is not in use. In other words the value X is lost.

A worse situation can occur if the processor is switched part way through a statement. Thus Harry might lose the processor just after he has picked up Top into a register but before he replaces Top with the new value. Suppose Dick now comes along and also does a Push thereby adding 1 to the old value of Top. When Harry resumes he will replace the value that Dick computed by the same value. In other words the two calls of Push add just 1 to Top rather than 2 as expected.

These unwanted behaviors are prevented by using a *protected object* for the stack, one of the major features introduced in Ada 95. We write

```
protected Stack is
  procedure Clear;
  procedure Push(X: in Float);
  procedure Pop(X: out Float);
private
  Max: constant := 100;
  Top: Integer range 0 .. Max := 0;
  A: Float_Array(1 .. Max);
end Stack;

protected body Stack is

  procedure Clear is
  begin
    Top := 0;
  end Clear;
```

```
    procedure Push(X: in Float) is
    begin
      Top := Top + 1;
      A(Top) := X;
    end Push;

    procedure Pop(X: out Float) is
    begin
      X := A(Top);
      Top := Top – 1;
    end Pop;

  end Stack;
```

Note that **package** has been changed to **protected**, the data which was in the body now appears in the private part of this new construct, and for reasons explained below the function Pop has been changed into a procedure Pop. Note also that type Float_Array is assumed to be defined elsewhere in the program as **array** (Integer **range** <>) **of** Float.

The three procedures Clear, Push and Pop are called *protected operations* and are invoked in the same way as procedures. Their behavior is that only one task can access the operations of the object at a time. If a task such as Tom attempts to call the procedure Pop while Harry is executing Push then Tom is forced to wait until Harry returns from Push. This is done automatically with no effort on the part of the programmer. Inconsistency problems are thus avoided.

Behind the scenes the protected object has a lock, and a task attempting to access an operation of the object has to acquire the lock first. If another task already has the lock then the first one has to wait until that other task has finished with the protected operation of the object that it was using and so relinquishes the lock. (The lock may be implemented through some operating system primitive, but the feature has been designed so as to permit alternative strategies with lower run-time cost.)

We can modify this example to show how we might cope with an attempt to push an item on the stack when it is full or to pop an item from an empty stack. In the package formulation, either of these attempts would raise Constraint_Error. In the case of Push this would be because of the attempt to assign the value Max+1 to Top; analogous problems would occur with Pop. As it is written the same thing would happen here and the lock would be automatically relinquished, because the exception terminates the call of the protected procedure.

But we can do much better. We can use *barriers* as follows

```
protected Stack is
   procedure Clear;
   entry Push(X: in Float);
   entry Pop(X: out Float);
private
   Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: Float_Array(1 .. Max);
end Stack;

protected body Stack is

   procedure Clear is
   begin
      Top := 0;
   end Clear;

   entry Push(X: in Float) when Top < Max is
   begin
      Top := Top + 1;
      A(Top) := X;
   end Push;

   entry Pop(X: out Float) when Top > 0 is
   begin
      X := A(Top);
      Top := Top – 1;
   end Pop;

end Stack;
```

The operations Push and Pop are now *entries* rather than procedures, and they have Boolean barrier expressions such as Top < Max. The effect of a barrier is to prevent the body of the entry from being executed if the barrier is False. Note that this does not prevent the entry from being called. All that happens is that the calling task is suspended until the barrier becomes True. So if Harry tries to call Push when the stack is full then he has to wait until some other task (Tom or Dick) calls Pop and removes the top item. Harry will then automatically proceed. The user does not have to program anything special.

Note that entries, like protected procedures, are also called in the same way as normal procedures, thus

```
Stack.Push(Z);
```

In summary, the protected object mechanism provided by Ada gives a structured mechanism for arranging mutually-exclusive access to a shared data object. A protected object declares its protected operations (procedures, functions, or

entries) in the visible part of its specification, and the protected components in its private part. The body of the protected object contains the implementation of the protected operations. A protected procedure and a protected entry have "read/write" access to the protected components – that is, they can reference and/or assign to them – whereas a protected function only has "read" access. This restriction enables an optimization whereby multiple tasks may simultaneously read a protected object (through protected function calls) but only one task at a time is allowed to write to it. (This is sometimes called "Concurrent Read, Exclusive Write".) The prohibition against protected functions assigning to protected components required Pop to be expressed as a procedure rather than a function in the first protected object version of Stack above.

Note also that, just as we can declare a task type as a template for task objects, we can likewise declare a *protected type* as a template for protected objects. And like task types, protected types are limited.

It is instructive to consider how we might program this example using lower level primitives. The historic basic primitives are the operations *P (acquire)* and *V (release)* acting on objects called *semaphores*. The effect of P(sem) is to acquire the lock associated with sem, if the lock is available, and otherwise to suspend the calling task on a queue associated with sem. The effect of V(sem) is to release the lock associated with sem and to awaken one of the tasks (if any) suspended on the queue of sem.

The idea is that we put pairs of calls of *P* and *V* around the operations for which we wish to ensure mutually exclusive access. Thus, using the same Ada syntax, Push would become

```
procedure Push(X: in Float) is
begin
  P(Stack_Lock);              -- secure the lock
  Top := Top + 1;
  A(Top) := X;
  V(Stack_Lock);              -- release the lock
end Push;
```

with similar pairs of calls around the body of Clear and Pop. This is essentially a Do-It-Yourself operation or assembly type coding for tasking. The opportunities for errors are many

- We might omit one of a *P* and *V* pair thus creating an imbalance.
- We might forget them altogether around one group of statements that should be protected.
- We might use the wrong semaphore name.
- We might inadvertently bypass a closing *V*.

The last problem would arise if, in the model without barriers, Push was called when the stack was full. This causes Constraint_Error to be raised. If we omit to provide a local exception handler to call *V* then the system will be permanently locked.

None of these difficulties can arise when using Ada protected objects because all this low-level mechanism is done automatically. Although, with care, semaphores can be used successfully in simple situations, it is very difficult to use them correctly in more complicated situations such as the example with barriers. Not only is it difficult to program correctly with semaphores but it is extremely difficult to prove that a program is correct.

Entry barriers are higher level and more reliable than the "condition variable" mechanism found in other concurrency approaches such as the wait/notify feature of Java. With the latter, the programmer is responsible for explicitly invoking wait, notify, or notifyAll on the variables that represent state information such as stack full or stack empty. This is error-prone and susceptible to race conditions that are prevented by the semantics of the protected objects of Ada.

The rendezvous

In addition to accessing shared data with a guarantee of mutually exclusive access, the other important communication requirement between tasks is for one task to convey information (data) to another. This is done in Ada with a mechanism known as a *rendezvous*. The two tasks that communicate have a client–server relationship. The client that requests some service needs to know the identity of the server task, but the server task that provides it will accept a request from any client.

Here is the general pattern of a server that only needs to offer one kind of service

```
task Server is
  entry Some_Service(Formal: in out Data);
end;

task body Server is
begin
  ...
  accept Some_Service(Formal: in out Data) do
    ...              -- statements providing the service
  end Some_Service;
  ...
end Server;
```

The specification of the server indicates that it has an entry Some_Service. This is called by a client task in the same way as calling an entry of a protected object. The difference is that the code to be obeyed is given by an *accept statement* and that is only executed when the server task reaches the accept statement. Until that happens the calling task is suspended. When the server reaches the accept statement, it executes it using any parameters supplied by the client. The client remains suspended until the accept statement is finished and after any **out** or **in out** parameters have been updated.

The body of a client might look like

```
task body Client is
  Actual: Data;
begin
  ...
  Server.Some_Service(Actual);
  ...
end Client;
```

Each entry has an associated queue. If a task calls an entry of a server and the server is not waiting at an accept statement for that entry, then the caller is queued. On the other hand, if the server reaches an accept statement and there are no tasks waiting on the associated entry queue, then the server is suspended. An accept statement can appear anywhere that a statement is allowed in the task body, for example within a branch of a conditional (if) statement, or within a loop, and so the mechanism is very flexible.

The rendezvous is a high level mechanism (like the protected object) and as such is relatively easy to use correctly. The corresponding queuing mechanisms programmed at a low level are hard to write correctly.

Here is an example of how the rendezvous can be used to enable a service to be provided without the client waiting. The idea is that the client gives the server an entry to be called when a job is done. First we declare a mailbox type to manipulate objects of some type Item which we assume is already declared

```
task type Mailbox is
  entry Deposit(X: in Item);
  entry Collect(X: out Item);
end;
```

```
task body Mailbox is
  Local: Item;
begin
  accept Deposit(X: in Item) do
    Local := X;
  end;
  accept Collect(X: out Item) do
    X := Local;
  end;
end Mailbox;
```

A task of this type acts as a simple mailbox. An item can be deposited and collected later. The client passes the identity of a mailbox to the server so that the server can deposit the item in the mailbox from which the user can collect it later. We need an access type

```
type Mailbox_Ref is access Mailbox;
```

The tasks Server and Client now take the following form

```
task Server is
  entry Request(Ref: in Mailbox_Ref; X: in Item);
end;

task body Server is
  Reply: Mailbox_Ref;
  Job: Item;
begin
  loop
    accept Request(Ref: in Mailbox_Ref; X: in Item) do
      Reply := Ref;
      Job := X;
    end;
    ...                          -- work on job
    Reply.Deposit(Job);
  end loop;
end Server;
```

```
task Client;

task body Client is
  My_Box: Mailbox_Ref := new Mailbox;      -- create mailbox task
  My_Item: Item;
begin
  Server.Request(My_Box, My_Item);
  ...                          -- do something whilst waiting
  My_Box.Collect(My_Item);
end Client;
```

In practice the client might poll the mailbox from time to time to see if the item is ready. This is easily done using a conditional entry call which takes the form

```
select
  My_Box.Collect(My_Item);
  -- item collected successfully
else
  -- not ready yet
end select;
```

It is important to realize that the mailbox agent task serves several purposes. It decouples the deposit and collect operations so that the server can get on with the next job. Moreover, it means that the server need know nothing about the client; calling the client directly would require the client to be of a particular task type and this would be most impractical. The mailbox agent task enables us to factor out the only property required of the client, namely the existence of the entry Deposit.

Restrictions

The pragma Restrictions which can be used to ensure that we do not use certain features of the language in a particular program was mentioned in the chapters on Safe Object-Oriented Programming and Safe Memory Management.

Many of the restrictions relate to tasking (some were introduced in Ada 95, and others in Ada 2005 and Ada 2012). The tasking features in Ada are very comprehensive and provide a whole range of facilities necessary to meet the programming needs of a variety of real-time applications. But some applications are quite simple and do not need many of these facilities. Here are some samples of the sort of restrictions that can be applied.

```
No_Task_Hierarchy
No_Task_Termination
Max_Entry_Queue_Length => n
```

The restriction No_Task_Hierarchy prevents tasks from being declared inside other tasks or inside subprograms – all tasks are therefore inside library-level packages. No_Task_Termination means that all tasks run for ever – this is common in many control applications where each task essentially has an endless loop doing some repetitive action. And the restriction on entry queues places a limit on the number of tasks that can be queued on a single entry at any time.

The advantage of giving appropriate restrictions are twofold

- It might enable a somewhat simpler run-time system to be used. This could be smaller and faster and thus more appropriate for some time- and space-critical embedded applications.
- It might enable various properties of the application to be proved correct, concerning matters such as determinism, absence of deadlock, and ability to meet deadlines. This might be vital for certain safety-critical applications.

There are many other tasking restrictions and most of these concern tasking facilities that we have not described.

Ravenscar

A particularly important group of restrictions is imposed by the Ravenscar profile, which was developed in the mid 1990s and standardized as part of Ada 2005. In order to ensure that a program conforms to this profile we write

```
pragma Profile(Ravenscar);
```

in the program. Use of any of the excluded features (summarized below) would then cause a compile-time error.

The key purpose of the Ravenscar profile is to restrict the use of tasking facilities so that the effect of the program is predictable. (The profile was defined by the International Real-Time Ada Workshops which met twice at the remote village of Ravenscar on the coast of Yorkshire in North-East England.)

The profile is simply defined to be equivalent to a number of restrictions plus a few other related pragmas concerning matters such as scheduling. The restrictions include those mentioned earlier so there are no task hierarchies, all tasks run for ever, and entry queues have a limit size of one (that is, there can be only one task blocked at a time on a given entry).

The original version of the Ravenscar profile assumed a uniprocessor environment. In Ada 2012 the profile has been specified with semantics that apply also to multiprocessors, with the restriction that tasks may not change CPU. See below for further discussion of Ada 2012 support for multiprocessors.

The combined effect of the restrictions is that it is possible to make statements about the ability of a particular program to meet stringent requirements for the purposes of certification.

No other programming language offers the reliability of Ada as constrained by the Ravenscar profile. A description of the principles and use of the profile in high integrity systems will be found in an ISO/IEC Technical Report [8].

Safe shutdown

In some applications (for example process monitoring and control) the tasks will run forever; in other applications the tasks will naturally reach their end points and simply terminate. In these situations task termination is not an issue. But there are common cases where a task that is potentially in an infinite loop needs to be terminated – for example when there is a hardware failure of some sort so that the task is no longer needed. The question is how to arrange task termination that is both safe and immediate. Unfortunately these goals conflict, and the designer needs to consider the tradeoffs. Fortunately, the Ada language provides sufficient flexibility so that the designer has an appropriate set of features to choose from based on how the tradeoff is decided. But even if immediacy is deemed important, the Ada language semantics still ensure that critical operations being performed by the terminating task are completed before the task is allowed to terminate.

The key to the approach used in Ada is a concept known as an *abort-deferred region*. Such a region is a section of code that must be executed to completion or else there would be the risk of corrupting a shared data structure. Examples of abort-deferred regions are the bodies of protected operations and the bodies of accept statements. Note that abort deferred does not mean non-preemptible; it is still possible for a task in an abort-deferred region to be preempted by a higher-priority task for example.

To illustrate these concepts, consider the following version of the server task that we saw earlier:

```
task Server is
  entry Some_Service(Formal: in out Data);
end;
```

```
task body Server is
begin
  loop
    accept Some_Service(Formal: in out Data) do
      ...           -- statements updating Data
    end Some_Service;
  end loop;
end Server;
```

This task will execute the loop body repeatedly. If at some point there are no further calls of Some_Service the task will be suspended – something like being put in suspended animation but with no prospect of being awakened. Not a pleasant thought and not a pleasant style of programming; the program will "hang" instead of terminating gracefully.

There are several ways to deal with this situation. One is to define a client task with the specific responsibility to shut down the server when no other client calls are possible. Here is a possible way to express such an "executioner" task:

```
task Grim_Reaper;

task body Grim_Reaper is
begin

  abort Server;
end Grim_Reaper;
```

The intent behind the abort statement is to cause the target task (here Server) to terminate. However, it would be risky if this were a "pull the plug" sort of termination where the task is shut down immediately regardless of what it was doing. For example, if Server were in the midst of executing its accept statement then the parameter might be in an inconsistent state and the calling task would end up with a corrupted actual parameter, perhaps a semi-updated array. But as noted above, an accept statement is "abort deferred". If an attempt is made to abort a task while it is executing an accept statement, then the attempt will be noted by the run-time system (formally the task becomes "abnormal") but the task will not be terminated until it is outside the abort-deferred region (in this example, when it has completed the accept statement). For the reader who is familiar with Gilbert and Sullivan's *The Mikado*, we may summarize the semantics of aborting a task while it is executing an accept statement as the greeting that Pooh-Bah gave to Nanki-Poo offering congratulations on his wedding but condolences on his execution that would soon follow: "Long life to you — till then".

Even if a task is aborted when it is not within an abort-deferred region, the effect is not necessarily immediate. In brief, aborting a task changes its state to "abnormal" where it is treated rather like an ostracized leper. If some other task

unwisely attempts to communicate with the poor wretch (for example, by calling any of the aborted task's entries) then Tasking_Error is raised in the unlucky caller. Finally, if/when the aborted task reaches any scheduling point, such as an entry call or an accept statement, then it will be put out of its misery and terminated. (For implementations that comply with the Real-Time Systems Annex the termination requirement is more demanding: basically an abnormal task will be terminated as soon as it is outside an abort-deferred region.)

All of this may sound a bit depressing and somewhat complicated, and indeed programming with abort statements makes programs hard to write and harder to prove correct. For most purposes, the advice is "don't do it". Abort may be appropriate in contexts such as mode changes where an entire collection of tasks must be terminated, but otherwise it is better to use one of the following techniques, where a task itself decides when it is ready to be terminated – that is, termination is only allowed at specific points.

One approach is to reserve a special entry for shutdown requests; accepting that entry will lead to the termination of the called task through normal control flow. Here is a variation on the server task that illustrates this style:

```
task Server is
  entry Some_Service(Formal: in out Data);
  entry Shutdown;
end;

task body Server is
  … -- initialization
begin
 loop
    select
      accept Shutdown;
      exit;
   or
      accept Some_Service(Formal: in out Data) do
        ...                    -- statements updating Data
      end Some_Service;
    end select;
  end loop;
  … -- cleanup
end Server;
```

This version of the server task illustrates a common form of the select statement comprising a set of alternatives each starting with an accept statement for some entry. When the select statement is executed, the run-time system checks whether a call is pending on any of these entries. If none is pending, the task will be suspended until a call on one of these is received (at which point control goes to the corresponding branch). If exactly one is pending, then control goes

to that branch. If more than one is pending, then the effect of which branch is selected is based on the entry queuing policy (for implementations compliant with the Real-Time Systems Annex, the default policy is priority-based).

This form of select statement is common for server tasks that offer more than one service and where the number of calls or their order is not known in advance. In this specific example we can anticipate a particular order – first the calls on Some_Service and then a call on Shutdown – but we don't know the number of calls on Shutdown and thus we need to express the logic as an infinite loop.

As in the previous example, the program requires some other task to trigger Server's termination. But now there is no need for the correctness-challenged abort statement, the triggering task simply calls the Shutdown entry.

```
task Grim_Reaper;

task body Grim_Reaper is
begin

  Server.Shutdown;
end;
```

Assuming that Grim_Reaper is programmed correctly – i.e., that it only calls Shutdown after all possible calls on Some_Service have been served – then the Shutdown entry approach will meet our requirement for safe termination. The price is some latency, since the request for termination will not be honored immediately.

Ada provides another approach that does not put the onus on some other task to trigger the termination; in this case the idea is that a task will terminate automatically (under the control of the run-time system) when it can be guaranteed that it is safe to do so. The basic semantics of the guarantee is that a task that offers one or more services (expressed as an infinite loop on a select statement with one or more accept alternatives) can terminate when it is at the select statement and when no further calls on any of the entries are possible. The syntax that conveys this intent is a special form of the select statement with a terminate alternative

```
task Server is
  entry Some_Service(Formal: in out Data);
end;
```

```
task body Server is
  … -- initialization
begin
 loop
    select
      terminate;
   or
      accept Some_Service(Formal: in out Data) do
        ...                    -- statements updating Data
      end Some_Service;
    end select;
  end loop;
end Server;
```

So when Server reaches (or is suspended at) the select statement the run-time system will look around and check if any tasks are still alive that could reference Server (and thus call its entry). If there are none, then it is safe for Server to be terminated and this will happen automatically. There is no opportunity to execute explicit cleanup code in the task, as there was in the previous version, but a feature introduced in Ada 2005 allows the programmer to define a termination handler that is invoked as part of the termination of a task.

The terminate alternative has the advantage of readability and reliability; there is no risk, as with the abort approach or the explicit shutdown entry, to make the mistake of either failing to trigger the shutdown or triggering it too soon. The disadvantage is distributed overhead, since the run-time system has additional work to do even if the feature is not used.

To summarize, Ada provides various features and supports several styles for arranging task termination. The abort statement offers the lowest latency (albeit with deferral during abort-deferred regions) but has the problem of arranging a task to terminate when the task might not be in a state where termination is advised. An explicit shutdown request entry solves this last problem (the task will only shut down when it accepts this entry) but increases the latency and requires care in programming to ensure that the entry is called at the appropriate time (in particular, not called when some other tasks might still need service). Finally, the select statement with a terminate alternative is the most reliable, placing the complete control over shutdown with the server task itself and not with its clients, but introduces extra overhead in the run-time support.

We also note a feature that Ada has intentionally omitted: the ability to raise an exception in a task asynchronously. This sort of feature, for example as embodied in the deprecated Thread.stop() method that was available in the initial release of Java, is basically a semantic nightmare with the risk of leaving shared data objects in an inconsistent state. Exceptions in Ada are always

synchronous and do not suffer from such problems. Care must certainly still be taken when using exceptions; for example the programmer needs to be aware that if an exception is not handled by a task then it is not propagated but rather the task will terminate. But the complexities of asynchronous exceptions are avoided.

Timing and scheduling

No survey of Ada tasking would be complete without a few words about timing and scheduling.

There are statements to enable a program to be synchronized with a clock. We can delay a program for a specific amount of time (this is referred to as a *relative delay*) or until a specific time thus

```
delay 2*Minutes;
delay until Next_Time;
```

assuming suitable declarations for Minutes and for Next_Time. Small relative delays might be useful for interactive use, whereas a delay until a particular time can be used to program periodic events. Time itself can be measured either by a real-time clock (which is guaranteed to have a certain accuracy) or by the local wall clock which might be subjected to changes such as occur because of Daylight Savings. In Ada, it is even possible to take account of time zones and leap seconds.

Ada 2005 introduced a number of standard timers whose expiry can be used to trigger actions defined by a protected procedure (a handler). There are three kinds of timers, one enables the monitoring of the CPU time used by an individual task, one concerns the CPU budget for a group of tasks, and the third concerns time as measured by the real-time clock. The handler is attached to a timing event by a call of a procedure such as Set_Handler.

This is illustrated by the following amusing example concerning the boiling of an egg. We declare a protected object Egg thus

```
protected Egg is
  procedure Boil(For_Time: in Time_Span);
private
  procedure Is_Done(Event: in out Timing_Event);
  Egg_Done: Timing_Event;
end Egg;
```

```
protected body Egg is

  procedure Boil(For_Time: in Time_Span) is
  begin
    Put_Egg_In_Water;
    Set_Handler(Egg_Done, For_Time, Is_Done'Access);
  end Boil;

  procedure Is_Done(Event: in out Timing_Event) is
  begin
    Ring_The_Pinger;
  end Is_Done;

end Egg;
```

The consumer can then write

```
Egg.Boil(Minutes(4));
-- now read newspaper whilst waiting for egg
```

and the pinger will ring when the egg is ready.

The approach to task scheduling used by Ada is captured by the pragma Task_Dispatching_Policy(*policy*). Ada 95 formalized the FIFO_Within_Priorities policy and Ada 2005 introduced several others; these are defined in the Real-Time Systems Annex. A policy can be applied to all tasks in a program or just to those in certain priority ranges by the use of pragmas. The policies are

FIFO_Within_Priorities – Within each priority level to which it applies tasks are dealt with on a first-in–first-out basis. Moreover, a task may preempt a task of a lower priority.

Non_Preemptive_FIFO_Within_Priorities – Within each priority level to which it applies tasks run to completion or until they are blocked or execute a delay statement. A task cannot be preempted by one of higher priority. This sort of policy is widely used in high integrity applications.

Round_Robin_Within_Priorities – Within each priority level to which it applies tasks are timesliced with an interval that can be specified. This is a very traditional policy widely used since the earliest days of concurrent programming.

EDF_Across_Priorities – This provides Earliest Deadline First dispatching. The general idea is that within a range of priority levels, each task has a deadline and that with the earliest deadline is processed. This is a new policy and has mathematically provable advantages with respect to processor utilization.

Ada also has comprehensive facilities concerning the setting and changing of task priorities and the so-called ceiling priorities of protected objects. These avoid problems of priority inversion as described in [9].

Ada 2012 has added a number of useful enhancements in the area of timing and scheduling. Most are outside the scope of this booklet but we will mention one specific area that is now explicitly addressed by the language standard: support for multiprocessor and/or multicore platforms. Features include:

- The package System.Multiprocessors which declares a function that returns the number of CPUs.
- A task aspect specification for assigning a task to a given CPU.
- The child package System.Multiprocessors.Dispatching_Domains that allows defining ranges of CPUs as "dispatching domains" and then assigning a task to a specific domain either through a task aspect specification or a procedure call. A task assigned to a given domain can then be executed on any CPU in the range of that domain.
- A revised and more flexible definition of pragma Volatile that simply ensures that reads and writes occur in the correct order rather than imposing a requirement that the target variable be in memory.

11 Certified Safe with SPARK

For some applications, especially those that are safety-critical or security-critical, it is essential that the program be correct, and that correctness be established rigorously through some formal procedure. For the most severe safety-critical applications the consequence of an error can be loss of life or damage to the environment. Similarly, for the most severe security-critical applications the consequence of an error may be equally catastrophic such as loss of national security, degradation of commercial reputation, or just plain theft.

Applications are graded into different levels according to the effect of a software failure. For avionics applications the DO-178B [1] and DO-178C [2] standards define the following

level E none: no problem; e.g. entertainment system fails? – could be a benefit!

level D minor: some inconvenience; e.g. automatic lavatory system fails.

level C major: some injuries; e.g. bumpy landing, cuts and bruises.

level B hazardous: some dead; e.g. nasty landing with fire.

level A catastrophic: aircraft crashes, all dead; e.g. control system fails.

As an aside, note that although a failure of the entertainment system in general is level E, if the failure is such that the pilot is unable to switch it off (perhaps in order to announce something unpleasant) then that failure raises the entertainment system to level D.

For the most demanding applications, which require certification by an appropriate authority, it is not enough for a program to be correct. The program also has to be shown to be correct and that is much more difficult. Formal, mathematically based methods are needed, and that is why the SPARK language was originally designed.

This chapter gives a very brief introduction to SPARK 2014, the most recent version of the language and a major revision of its predecessor SPARK 2005. Using the Ada 2012 syntax for contracts such as preconditions and postconditions, SPARK 2014 is an Ada 2012 subset with several additional features that facilitate formal analysis (these features are provided through the aspect mechanism introduced in the chapter on *Safe Typing*). An Ada 2012 program can have some parts in full Ada and other parts in the SPARK subset (specially marked as such). SPARK components can be compiled by a standard Ada compiler and will have standard Ada run-time semantics, but are amenable to a more formal treatment than full Ada.

The SPARK components as well as being compiled by an Ada compiler are also analyzed by a SPARK analysis tool. This tool may be able to statically verify the contracts (such as preconditions and postconditions) that would be checked at run time in full Ada, and then the run-time checks can be suppressed with confidence that they would always succeed. Note that the SPARK tool used with GNAT is known as GNATprove and unless stated otherwise we shall assume that it is the tool being used for analysis.

It is important to note that SPARK can be used at various levels. At the simplest, an Ada program that conforms to the SPARK subset can be analyzed without any further effort. However, we can strengthen the description of the program by adding various aspects regarding flow of information and proof and these will enable the analyzer to give a more rigorous analysis of the program. Ultimately whether to include such aspects is a choice for the user and will depend on the project context, including issues such as the integrity level for the software and the verification objectives for using the SPARK tools.

We start by considering in a little more detail the important concepts of correctness and contracts.

Contracts

What do we mean by correct software? Perhaps a general definition is: software that does what the user had in mind. And "had in mind" might literally mean just that for a simple one-off program written to do an *ad hoc* calculation; for a large avionics application, it might mean the text of some written contract between the ultimate client and the software developer.

This idea of a software contract is not new. If we look at the programming libraries developed in the early 1960s, particularly in mathematical areas and perhaps written in Algol 60 (a language favored for the publication of such material in respected journals such as the *Communications of the ACM* and the *Computer Journal*), we find that the manuals tell us what parameters are required, what constraints apply on their range and so on. In essence there is a contract between the writer of the subroutine and the user. The user promises to hand over suitable parameters and the subroutine promises to produce the correct answer.

The decomposition of a program into various component parts is very familiar and the essence of the programming process is to define what these parts do and therefore what the interfaces are between them. This enables the parts to be developed independently of each other. If we write each part correctly (so that it satisfies its side of the contract implied by its interface) and if we have defined the interfaces correctly, then we are assured that when we put the parts together to create the complete system, it will work correctly.

Bitter experience shows that life is not quite like that. Two things go wrong: on the one hand the interface definitions are not usually complete (there are holes in the contracts) and on the other hand, the individual components are not correct or are used incorrectly (the contracts are violated). And of course the contracts might not say what we meant them to say anyway.

Interestingly, there are several ways in which contracts can be supplied for program components. First, and the style that historically has been strongly encouraged by SPARK, is to include contracts from the start. This has been referred to as "correctness by construction", and is also known as "declarative verification" where all program units include contracts on their specifications. Contracts may be regarded as explicitly specified low-level requirements for program units; if the contracts are not consistent with the code then such mismatches are detected before program execution. There is strong evidence from a number of years of use of SPARK in application areas such as avionics, banking, and railway signaling that indeed, not only is the program more likely to be correct, but the overall cost of development is actually less in total after all the testing and integration phases are taken into account.

Although such an approach is effective for new projects, it can present challenges when existing software is to be extended with new components in SPARK. Accordingly, a second style is supported by SPARK 2014 and is known as "generative verification": if code does not include contracts, then GNATprove can synthesize some of the implicit contracts from the bodies of the units. In this manner a project can move from generative verification to declarative verification as the system evolves. Furthermore, as will be explained later, SPARK 2014 allows developers to combine formal verification with traditional methods based on testing.

We will now look in a little more detail at the two problem areas introduced above, first giving complete interface definitions, and secondly ensuring that the code correctly implements the interface. It is simplest to present this in terms of the declarative verification model, but the concepts also apply in the context of generative verification.

Ideally, the definition of the interfaces between the software components should hide all irrelevant detail but expose all relevant detail. Alternatively we might say that an interface definition should be both complete and correct.

As a simple example of an interface definition consider the interface to a subprogram. As just mentioned, the interface should describe the full contract between the user and the implementer. The details of how the subprogram is implemented should not concern us. In order that these two concerns be clearly distinguished it is helpful to use a programming language in which they are lexically distinct. Some languages present subprograms (functions or methods) as one lump, with the interface physically bound to the implementation. This is a nuisance: not only does it make checking the interface less straightforward

since the compiler wants the whole code, but it also encourages the developer to hack the code at the same time as writing the interface and this confuses the logic of the development process.

Ada has a structure separating interface (the specification) from implementation (the body). This applies both to individual subprograms and to groups of entities encapsulated into packages and this is a key reason why Ada forms such a good base for SPARK.

As mentioned earlier it is often convenient for only parts of a program to be written in SPARK with other parts in full Ada. The parts in SPARK are indicated by the aspect SPARK_Mode. This can be applied to individual subprograms or (perhaps more likely) to a package in which case it will apply to all the subprograms in the package. Thus the specification of a package P might begin

```
package P
  with SPARK_Mode is
  ...
```

In addition, the aspect can be supplied by a pragma which is convenient if it is to apply to the whole program thus

```
pragma SPARK_Mode;
```

In SPARK, additional information regarding an interface can be provided, and this is done through the mechanism of Ada 2012 aspects. A key purpose of these aspects is to increase the amount of information about the interface without providing unnecessary information about the implementation. In fact SPARK allows the information to be added at various levels of detail as appropriate to the needs of the application.

Consider the information given by the following Ada specification

```
procedure Add(X: in Integer);
```

Frankly, it tells us very little. It just says that there is a procedure called Add and that it takes a single parameter of type Integer whose formal name is X. This is enough to enable the compiler to generate code to call the procedure. But it says nothing about what the procedure does. It might do anything at all. It certainly doesn't have to add anything nor does it have to use the value of X. It could for example subtract two unrelated global variables and print the result to some file. But now consider what happens when we add a SPARK 2014 construct that defines how a global variable is used. We will assume that the procedure is within a package to which SPARK_Mode has been applied. The specification of Add might become

```
procedure Add(X: in Integer)
  with Global => (In_Out => Total);
```

The Global aspect states that the only global variable that the procedure can access is that called Total. Moreover, the In_Out identifier tells us that the initial value of Total must be used and that a new value will be produced. The SPARK rules also say more about the parameter X. Although in Ada a parameter need not be used at all, nevertheless an **in** parameter must be used (by the body) in SPARK or else the analyzer will issue a warning.

So now we know rather a lot. We know that a call of Add will produce a new value of Total and that it will use the initial value of Total and the value of X. We also know that Add cannot affect anything else. It certainly cannot print anything or have any other unspecified side effect. (If the body did this, it would be rejected by the analyzer.)

Of course, the information regarding the interface is not complete since nowhere does it require that addition be performed in order to obtain the new value of Total. In order to do this we can add a postcondition using the aspect Post and obtain

```
procedure Add(X: in Integer)
  with Global => (In_Out => Total),
       Post   => Total = Total'Old + X;
```

The postcondition explicitly says that the final value of Total is the result of adding its initial value (distinguished by 'Old) to that of X. So now the specification is complete.

It is also possible to provide preconditions. Thus we might require X to be positive and to have a value such that integer overflow is prevented; we could express this by the following

```
Pre => X > 0 and then Total <= Integer'Last – X
```

(Constraining X to be greater than zero would be better expressed by declaring the formal parameter to be of subtype Positive; we capture this constraint in the Pre aspect for purposes of illustration.)

Pre- and postconditions, like other SPARK aspects, are optional. If not specified explicitly, they are assumed to be True.

Another aspect that can be given for a subprogram is the Depends aspect. This indicates the dependencies between the initial values of parameters and globals and their final values. In the case of Add it would be

```
Depends => (Total => (Total, X))
```

which simply says that the final value of Total depends upon its initial value and the value of X. However, in this example, it can be deduced from the parameter modes and globals anyway and adds no further information.

It should be remembered that all SPARK related aspects are optional. However, an important principle is that all aspects that are given are verified statically by the SPARK analyzer when it analyzes the body of the subprogram.

In the case of the pre- and postconditions, it will attempt to prove the following

> If the precondition is met then (1) there are no run-time errors, and (2) if the subprogram terminates then its postcondition will be satisfied.

If a contract cannot be so verified, then the unit will be rejected by the analyzer but can still be compiled by an Ada compiler which will generate run-time checking code (if the Assertion_Policy is Check). The use of the same contract syntax in Ada 2012 and SPARK 2014 thus offers considerable flexibility; for example, initially the developer could perform testing with dynamic contracts and then progress to static verification as the contracts and/or the code are refined.

For critical systems such static contract verification is important. If pre- and postconditions were only checked when the program executes then it would be a bit like bolting the door after the horse has bolted (which reveals a nasty pun caused by overloading in English!). We don't really want to be told that the conditions are violated as the program runs. For example, we might have a precondition for landing an aircraft

```
procedure Touchdown( ... )
  with Pre => Undercarriage_Down;
```

It is pretty unhelpful to be told that the undercarriage is not down as the plane lands; we really want to be assured that the program has been analyzed to show that the situation will not arise.

This thought leads into the other problem with programming – ensuring that the implementation correctly implements the interface contract. This is often called debugging. Generally there are four ways in which bugs are found:

(1) By the compiler. These are usually easy to fix because the compiler tells us exactly what is wrong.

(2) At run time by a language check. This applies in languages which carry out checks that, for example, ensure that we do not write outside an array. Typically we obtain an error message saying what rule was violated and whereabouts in the program this happened.

(3) By testing. This means running various examples and poring over the (un)expected results and wondering where it all went wrong.

(4) By the program crashing. This often destroys much of the evidence as well so can be very tedious.

Type 1 should really be extended to mean "before the program is executed". Thus it includes program walkthroughs and similar review techniques and it includes the use of static analysis tools such as GNATprove.

Clearly these four ways represent a progression of difficulty. Errors are easier to locate and correct if they are detected early. Good programming tools are those which move bugs from one category to a lower numbered category. Thus good programming languages are those which provide facilities enabling one to protect oneself against errors that are hard to find. Ada is a particularly good programming language because of its strong typing and run-time checks. For example, the correct use of enumeration types makes hard bugs of type 3 into easy bugs of type 1 as we saw in the chapter on *Safe Typing*.

A major goal of SPARK is to strengthen interface definitions (the contracts) and so to move all errors to a low category and ideally to type 1 so that they are all found before the program executes. Thus the Global aspect does this because it prevents us writing a program that accidentally changes the wrong global variables. Similarly, detecting the potential violation of pre- and postconditions results in a type 1 error. However, checking that such violations cannot happen requires mathematical proof; this is not always totally straightforward but GNATprove usually automates the whole proof process.

The SPARK Ada subset

Ada is a very comprehensive language and the use of some features makes total program analysis difficult. Accordingly, the subset of Ada supported by SPARK omits certain features. These mostly concern run-time behavior. For example, there are no access types (and thus no dynamic allocation) and no exception handling.

Tasking of course is very dynamic and although full Ada tasking is too complex, the Ravenscar profile mentioned in the chapter on *Safe Concurrency* is amenable to formal analysis. Ravenscar was included in SPARK 2005 and is scheduled to be added in a future version of SPARK 2014 as noted at the end of this discussion.

Here are some other simplifications enforced by SPARK 2014:

- All expressions (including function calls) are free of side effects. Although Ada 2012 allows functions to take **out** or **in out** parameters, these are prohibited in SPARK, and functions are not allowed to modify globals These restrictions help to guarantee that the compiler's choice of order of evaluation does not affect the result, and reinforce the strong distinction between procedures whose purpose is to change state and functions whose purpose is simply to observe state.

- Aliasing of names is not permitted. For example, it is prohibited to pass an object as an **out** or **in out** parameter to a procedure that references that object as a global (unless the object is of a by-copy type). This restriction makes the program's effect clearer and helps to guarantee that the compiler's choice of by-reference versus by-copy parameter passing does not affect the result.
- The **goto** statement is prohibited. This restriction facilitates static analysis.
- The use of controlled types is prohibited. Controlled types lead to compiler-generated implicit calls, and the absence of source code for constructs requiring formal analysis complicates the user's interactions with the analysis tool.

In addition to these restrictions, SPARK enforces a stricter initialization policy than full Ada. An object passed as an **in** or **in out** parameter must be fully initialized before the subprogram is called, and a formal **out** parameter must be fully initialized before the subprogram returns.

Such restrictions notwithstanding, the SPARK subset includes a large part of Ada 2012 including discriminated types (but not access discriminants), tagged types / dispatching, types with dynamic bounds, array slicing and concatenation, static predicates, conditional and quantified expressions, expression functions, generic units, child units, and subunits. Recursion is permitted, and a formal containers library can be used for constructing complex data structures in the absence of access types and dynamic allocation. SPARK also provides facilities for dealing with interfaces to the external environment, through a set of aspects relating to volatile variables (variables susceptible to asynchronous reads or writes).

Earlier versions of SPARK have proved to be sufficiently expressive for the needs of critical applications which would not want to use features such as dynamic storage, and SPARK 2014 has considerably expanded the set of permitted features.

Formal verification

In this section we briefly summarize some aspects of the mechanism of formal verification carried out by the SPARK analysis tool.

First there are two forms of analysis regarding flow which generally check for compliance with the SPARK rules thus

- *Flow analysis of data dependencies*, which considers the initializations of variables and the data dependencies of subprograms; and

- *Flow analysis of flow dependencies*, which considers how the outputs of a subprogram relate to its inputs.

Flow analysis does not require much effort on the part of the user other than perhaps adding the optional global and depends aspects and of course sticking within the SPARK rules. Adding these optional aspects enables the flow analysis to be more detailed and like other forms of redundancy can detect inconsistencies at an early stage.

An important feature of flow analysis is the detection of the use of uninitialized variables and the overwriting of variables before they are used. This means that care should be taken not to give junk values to variables "just in case" as mentioned in the chapter on *Safe Startup* because that would hinder the detection of flow errors.

And then there are the formal verification processes concerning proof

- *Formal verification of robustness properties* (that is, ensuring that the program is free from the raising of predefined exceptions); and
- *Formal verification of functional properties*, based on contracts such as preconditions and postconditions.

In the case of functional properties which include loop invariants and type assertions as well as pre- and postconditions, the general process that the analyzer follows is to generate conjectures (potential theorems) which then have to be proved to verify that the program is correct. These conjectures are known as *verification conditions* (VCs), and proving them is usually known as *discharging* the VCs. Much progress has been made in recent years regarding automatic proof and in very many cases GNATprove will discharge them easily. At the time of writing the default technolgies used are Alt-Ergo and CVC4 but other proof engines can be invoked as well.

It is important to note that even without the addition of aspects such as pre- and postconditions, the analyzer can generate conjectures corresponding to the run-time checks of Ada such as range checks. As we saw in the chapter on *Safe Typing*, these are checks automatically inserted to ensure that a variable is not assigned a value outside the range permitted by its declaration or that no attempt is made to read or write outside the bounds of an array. The proof of these conjectures shows that the checks would not be violated and therefore that the program is free of run-time errors that would raise exceptions.

Note that the use of proof is not necessary. SPARK and its tools can be used at various levels. For some applications it might be appropriate just to perform flow analysis. But for other applications it might be cost-effective to use the proof aspects as well. Indeed, different levels of analysis can be applied to different parts of a complete program using various options provided by the aspect SPARK_Mode.

Hybrid verification

Formal verification might be impractical (or perhaps even impossible) on an entire code base for several reasons:

- Parts of the program may be in full Ada (or in another language such as C) and not amenable to formal analysis. For example a package specification might be declared with SPARK_Mode but its body might be in full Ada. The contract information present in the package specification can be used by the proof tools, even though formal analysis might not be possible for the body. But traditional dynamic verification methods (that is, testing) are needed for the non-SPARK units.
- Even if a component is within the SPARK subset, the properties that must be demonstrated might not be expressible formally or their proof may be beyond the abilities of current proof technology. Again, testing will be needed to gain confidence that the desired properties are met.

Several elements of AdaCore's GNAT technology help users to carry out such "hybrid verification" that combines static and dynamic analysis (formal methods and testing).

- *Generative verification*. GNATprove can be used on dynamically tested unit bodies that are in full Ada or in SPARK but without contracts, to compute their implicit data dependencies. This approach, referred to earlier as "generative verification", allows formal methods to be introduced for critical components during the maintenance or upgrade of an existing code base.
- *GNATprove output*. GNATprove will output the properties of a subprogram body that the tool is unable to establish, for example it may indicate that run-time errors are possible. By highlighting potential issues such output can help the user compose relevant test cases or can show where further analysis of the code is needed.
- *Compilation options*. Several compiler switches result in run-time checks for errors such as aliased parameters and invalid scalar objects, for units that are not subject to formal analysis.
- *GNATtest tool*. An implementation-defined aspect in GNAT (also realized by a pragma) allows the user to define "formal test cases" for subprograms. A formal test case identifies a required condition on entry and an expected condition on return, and indicates whether the test exercises normal processing or the subprogram's robustness (for example, responding to out-of-range input values). The GNATtest tool uses this information to generate a test harness.

Examples

As a simple example here is a version of the stack with flow dependencies (Depends aspects) but not pre- or postconditions:

```
package Stacks
  with Spark_Mode
is

  type Stack is private;

  function Is_Empty(S: Stack) return Boolean;
  function Is_Full(S: Stack) return Boolean;

  procedure Clear(S: out Stack)
    with Depends => (S => null);

  procedure Push(S: in out Stack; X: in Float)
    with Depends => (S => (S, X));

  procedure Pop(S: in out Stack; X: out Float)
    with Depends => (S => S,
                     X => S);
private
  Max: constant := 100;
  type Top_Range is range 0 .. Max;
  subtype Index_Range is Top_Range range 1 .. Max;

 type Vector is array (Index_Range) of Float;
  type Stack is
    record
      A: Vector;
      Top: Top_Range;
    end record;
end Stacks;
```

We have added functions Is_Full and Is_Empty which just read the state of the stack. They have no associated contracts.

Depends aspects have been added to the various procedure specifications. Their purpose is to say which outputs depend upon which inputs; for example in the case of Push we have (S => (S, X)) which means that the final value of S depends upon the initial value of S and the initial value of X – in this simple example they can in fact be deduced from the parameter modes. However, redundancy is one key to reliability and if they are inconsistent with the modes

then that will be detected by the analyzer and perhaps thereby reveal an error in the specification.

The declarations in the private part have been changed to give names to all the subtypes involved, although this is not necessary in SPARK 2014.

At this level there are no changes to the package body at all – no contracts are required. This emphasizes that SPARK is largely about improving the quality of the description of the interfaces.

A difference from the earlier examples is that we have not given an initial value of 0 for Top but require that Clear be called first. When GNATprove looks at SPARK client code it will perform flow analysis to ensure that Push and Pop are not called until Clear has been called; this analysis will be performed without executing the program. If GNATprove cannot deduce this then it will report that the program has a potential flow error. On the other hand if it can actually deduce that Push or Pop are called before Clear then it will report that the program is definitely in error.

In this brief overview it is not feasible to give serious examples of the proof process but the following trivial example will illustrate the ideas. Consider

```
procedure Exchange(X, Y: in out Float)
  with Depends => (X => Y,
                   Y => X),
       Post => (X = Y'Old and Y = X'Old);
```

which shows the specification of a procedure whose purpose is to interchange the values of the two parameters. The body might be

```
procedure Exchange(X, Y: in out Float) is
  T: Float;
begin
  T := X;  X := Y;  Y := T;
end Exchange;
```

Analysis by GNATprove generates a verification condition which has to be shown to the true. In this particular example this is trivial and is done automatically. (Readers might care to note that the VC is more or less simply the logical expression (x=x and y=y) which indeed is clearly true.) In more elaborate situations GNATprove might not be able to complete a proof in which case the user can provide guidance by providing intermediate assertions or can employ supplemental tools to perform further analysis.

Certification

As earlier chapters have shown, Ada is an excellent language for writing reliable software. Ada allows programmers to catch errors early in the development process. Even more errors can be detected by using SPARK without having to rely on testing – a difficult and error-prone process in itself, yet an indispensable part of the software process.

For the highest level of safety-critical and security-critical applications it is not enough for a program to be correct. It also has to be shown to be correct. This is usually called certification and is performed according to the methods of a relevant certification agency. Examples of such agencies in the US are the FAA for safety-critical applications and the NSA for security-critical applications. SPARK is of great value in developing programs to be certified as safe or secure as appropriate.

It might be thought that using SPARK adds to development costs. However, a study concerning a security system for the NSA [10] showed that using SPARK proved cheaper than conventional development methods. This again is perhaps surprising because SPARK clearly requires effort for generating the contracts. But again that effort is well spent and reduces time needed for correcting errors.

Work in progress

At the time of writing, two major future developments are in progress. One is the addition of tasking to SPARK 2014 including the Ravenscar profile. The other is the ability to indicate levels of integrity to various components of a program. These ensure that flow of information does not violate required safety and security levels.

For up-to-date information and comprehensive documentation on SPARK 2014 please visit **www.spark-2014.org/**.

Conclusion

It is hoped that this booklet will have proved interesting. It has covered a number of aspects of writing reliable software and hopefully has shown that Ada is a good language and source of inspiration to use for programs that matter. We conclude with some background notes on the development of languages.

The balance between hardware and software is interesting. Hardware has evolved in an amazing way in the last half century. The hardware of today bears no resemblance whatever to the hardware of 1960. By contrast, software has progressed but little. Most languages of today are in many ways little different to those of 1960. I suspect that the ultimate problem is that we know little about software although we probably think we know rather a lot. Moreover, society has made huge investments in badly written software and finds it hard to move forward at all. But hardware changes so rapidly that it inevitably gets discarded. And of course it is very easy for anyone to learn to write a bit of software but massive know-how is required to build any hardware.

Mainstream languages have two main origins, Algol 60 and CPL. These are the ancestors of the languages mentioned most in this booklet. Another group of languages, Fortran, COBOL and PL/I, live on but seem to be somewhat isolated.

Algol 60 was perhaps the most important step forward ever made. (There was a lesser known precursor called Algol 58 from which the US military language Jovial was derived but that is a minor detail.) Algol gave the feeling that writing software was more than just coding.

Algol made two big steps. It recognized that assignment was not equality by using := for assignment. It also introduced English words for control purposes and thereby eliminated most of the gotos, jumps and labels that made early Fortran and autocode programs so hard to understand. This second point is worth looking at in some detail.

Consider first the following two statements in Algol 60

```
if X > 0 then
  Action( ... );
Otherstuff( ... );
```

The effect is that if X is indeed greater than zero then the subroutine Action is called. Whether Action is called or not we then always go on to call Otherstuff. The interesting thing is that the conditional only governs the first statement following **then**. If we need to govern several statements such as call subroutines

This and That then we have to combine the two statements into a single compound statement thus

```
if X > 0 then
begin
   This( ... );
   That( ... );
end;
Otherstuff( ... );
```

There are two dangers here. One is simply that we might forget to insert **begin** and **end**. It would still compile of course but That would always get called whatever the value of X. But a bigger hazard is the danger of stray semicolons. Algol 60 was perhaps the first language to use semicolons to terminate or separate statements. Now consider what happens if a programmer inadvertently adds a semicolon immediately after **then**. We get

```
if X > 0 then ;
begin
   This( ... );
   That( ... );
end;
Otherstuff( ... );
```

Unfortunately, in Algol 60 the semicolon is deemed to be separating a null statement from the compound statement (a null statement does nothing – it is invisible too!) And so the conditional does nothing and the subroutines This and That are always called. There were other related problems in Algol 60 concerning the syntax of loops.

The designers of Algol 68 recognized this problem and introduced a bracketed form thus

```
if X > 0 then
   This( ... );
   That( ... );
fi;
Otherstuff( ... );
```

Other similar structures were used for loops with **do** being matched by **od** and **case** being matched by **esac**. This structure completely solves the problem. It is now crystal clear that the conditional governs the two statements. Moreover, adding a spurious semicolon after **then** is a syntax error and so is instantly detected by the compiler. Of course many thought that the reversed words **fi**, **od** and **esac** indicated that the language was bizarre and not to be taken seriously.

Whatever the reason, the designers of Pascal ignored this sensible approach and continued to use the flawed structure of Algol 60. Eventually however they did realize their error when it came to Modula 2 but this was long after Ada.

Ada was probably the first successful language to use the bracketed structure but it does sensibly avoid the peculiar backward words. Thus in Ada we write

```
if X > 0 then
  This( ... );
  That( ... );
end if;
Otherstuff( ... );
```

Many other languages have taken this safe route including even the macro language in the elegant Microsoft Word for DOS and Visual Basic which is the corresponding macro language for Word for Windows.

The other important background language was CPL. It was devised in about 1962 as the language to be used by two powerful new computers at Cambridge and London universities.

CPL (like Algol 60) used := for assignment and = for equality. Here is a small fragment of CPL

```
§ let t, s, n = 1, 0, 1
  let x be real
  Read[x]
    t, s, n := tx/n, s + t, n + 1
    repeat until t << 1
  Write[s] §
```

An interesting feature of CPL is that it used = rather than := when setting initial values on the grounds that no change was involved. CPL had many novel features such as parallel assignments and list processing. However, CPL was never implemented but remained an academic design.

CPL used essentially the same structure as Algol 60 for grouping statements. Thus we would have written

```
if X > 0 then do
  § This( ... )
    That( ... ) §
Otherstuff( ... )
```

Note that the items grouped together are surrounded by the strange brackets § and §(note that the closing bracket was the section sign with the vertical bar through it).

Although CPL was never implemented, the simple language BCPL (Basic CPL) was a simple successor devised at Cambridge. The major difference was that whereas CPL was a strongly typed language, BCPL really had no types at all and arrays were just treated as arithmetic on addresses. BCPL is the origin of the buffer overflow problem which plagues the world today.

From BCPL came B and then C, C++ and so on. BCPL used := for assignment but somewhere along the way someone missed the point and C ended up with = for assignment. Having hijacked = for assignment C uses a double equals (==) to mean equality and this gives rise to a number of problems as we saw in the chapter on Safe Syntax.

C inherited the same compound statement style from CPL but replaced the strange brackets by the braces '{' and '}'and thus in C we write

```
if (x > 0)
{
  this( ... );
  that( ... );
};
otherstuff( ... );
```

There is little of the original CPL left in C. In fact the only thing really left is the brackets.

And finally, we conclude by noting that the use of the equals sign for assignment is an example of the use of puns so hated by the late Christopher Strachey. Strachey was one of the designers of CPL. At a NATO lecture many years ago he said *"The way in which people are taught to program is abominable. They are over and over again taught to make puns; to do shifts when they mean multiplying; to confuse bit patterns and numbers and generally to say one thing when they mean something quite different. I think we will not make it possible to have a subject of software engineering until we can have some proper professional standards about how to write programs; and this has to be done by teaching people right at the beginning how to write programs properly. I'm sure that one of the first things to do about this is to say what you mean, and not to say something quite different."*

That about sums it up. We need to learn to say what we mean. Ada enables us to say what we mean clearly and that ultimately is its strength.

Bibliography

The Ada 2012 Language Reference Manual and Rationale are available at

www.ada-auth.org/standards/ada12.html

The following book presents a comprehensive description of Ada 2012:

John Barnes. *Programming in Ada 2012*. Cambridge University Press (2014).

An introduction to SPARK 2014 is provided in

John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press (2015).

And detailed coverage of the previous version of the language, SPARK 2005, may be found in

John Barnes with Altran Praxis. *SPARK – The Proven Approach to High Integrity Software.* Altran Praxis (2012).

The following award-winning book is a good introduction to lean software development.

Peter Middleton, James Sutton. *Lean Software Strategies: Proven Techniques for Managers and Developers.* Productivity Press (2005).

The following websites provide access to much useful information.

www.adacore.com – for AdaCore and its products.

www.ada-europe.org – for Ada-Europe, conferences and journal.

www.sigada.org – for ACM SIGAda.

www.adaic.org – for the Ada Information Clearinghouse.

www.sparkada.com – for news from Altran on the SPARK language and tools.

www.spark-2014.com – for up-to-date news and resources concerning the SPARK language.

The following further documents and books are referenced in the text.

[1] RTCA DO-178B / EUROCAE ED-12B. *Software Considerations in Airborne Systems and Equipment Certification* (December 1992).

[2] RTCA DO-178C / EUROCAE ED-12C. *Software Considerations in Airborne Systems and Equipment Certification* (December 2011).

[3] *MISRA-C:2004, Guidelines for the use of the C language in critical systems* (October 2004).

[4] Barbara Liskov and Jeannette Wing. "A behavioral notion of subtyping", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 16, Issue 6 (November 1994), pp 1811-1841.

[5] RTCA DO-332 / EUROCAE ED-217. *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A* (December 2011).

[6] AdaCore. *High-Integrity Object-Oriented Programming in Ada*, Release 1.3 (July 2011), www.open-do.org/hi-oo-ada

[7] Cyrille Comar and Pat Rogers. *On Dynamic Plug-in Loading with Ada 95 and Ada 2005*. AdaCore (2005). www.sigada.org/ada_letters/jun2005/Dynamic_plugin_loading_with-Pat Rogers.pdf

[8] ISO/IEC TR 24718:2004. *Guide for the use of the Ada Ravenscar profile in high integrity systems* (2004).

[9] Alan Burns and Andy Wellings. *Concurrent and Real-Time programming in Ada 2005*. Cambridge University Press (2006).

[10] Janet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper and Bill Everett. *Engineering the Tokeneer Enclave Protection Software*. Published in ISSSE 06, the proceedings of the 1st IEEE International Symposium on Secure Software Engineering. IEEE (March 2006). Also available from www.sparkada.com.

Index