Ada

for Software Engineers

M. Ben-Ari

Ada for Software Engineers

M. Ben-Ari

Weizmann Institute of Science

Originally published by John Wiley & Sons, Chichester, 1998.

Contents

Preface

Albert Einstein once said that 'things should be as simple as possible, but not simpler'. Einstein could have been talking about programming languages, as the landscape is strewn with 'simple' languages that, several versions later, have 500-page reference manuals!

The truth is that we expect a lot of our programming languages. Turing machines just aren't sophisticated enough for modern software development; we demand support for encapsulation and abstraction, type checking and exception handling, polymorphism and more. Ada, unlike other languages which grew by gradual addition of features, was designed as a coherent programming language for complex software systems. As if to justify Einstein's saying, Ada is no more complex than the *final* versions of 'simpler' languages.

However, the complexity of modern programming languages leaves textbook writers with a painful dilemma: either gloss over the gory details, or write books that are heavy enough to be classified as lethal weapons. Is there another way?

Strange as it may seem, you can get an excellent *concise* description of Ada 95 for free: the *Ada Reference Manual (ARM)* (Taft & Duff 1997), which is the document defining the language standard. The *ARM* has a reputation for being ponderous reading meant for 'language lawyers'. Nevertheless, I believe that, with a bit of guidance, software engineers can learn to read most of the *ARM*. *Ada for Software Engineers* is written to equip you with the knowledge necessary to use the Ada 95 programming language to develop software systems. I will try to teach you how the individual language constructs are used in actual programs, and I will try to explain the terminology and concepts used in the language standard.

The book is intended for software engineers making the transition to Ada, and for upper-level undergraduate and graduate students (including those who have had the good fortune to study Ada as their first programming language!). No specific Ada knowledge is assumed; the prerequisites are a basic knowledge of computer science and computer systems, and significant programming experience (not necessarily in Ada). As the title implies, this book is for you if you are a software engineer or training to become one.

The Ada language will be taught using a few—relatively large—case studies, rather than a large number of small examples each crafted to demonstrate a particular construct or rule. Experienced programmers know that the key to mastering a programming language is not to memorize the syntax and semantics of individual constructs, but to learn how to integrate the constructs into language-specific paradigms. We will need to gloss over details when explaining a case study; rest assured that everything will eventually be explained, or you will find a pointer to the explanation

in the *ARM*. Certain sections marked with one or two asterisks should be omitted during your initial study of Ada. This material is not necessarily more difficult, but you can't learn everything at once, and these are topics that can be left for your second and third reading of the book.

After an introductory chapter, Chapters 2–4 quickly cover elementary language constructs such as statements, subprograms, arrays, records and pointers that should be familiar from your previous programming experience. The core of the book is the progressive development a of case study demonstrating the Ada constructs for object-oriented programming: packages and private types (Chapter 5), type extension, inheritance, class-wide types and dynamic polymorphism (Chapter 6) and generics (Chapter 7). Chapters 8–12 cover specialized topics: the type system in depth, input–output and program structure.

Chapters 13–16 form a second core of the book, discussing topics broadly covered by the term systems programming: multitasking, hardware interfaces, systems programming, and real-time and distributed systems. Most of these chapters are relatively independent of the material following Chapter 5 on packages.

Appendix A Tips for Transition will help Pascal, C, C++ and Java programmers relate constructs in those languages to the constructs of Ada. A special feature of the book is the Glossary (Appendix B), which explains the *ARM* terminology *with examples*. In addition, discussions in the text are cross-referenced to the relevant paragraphs in the *ARM*.

Appendix D contains a set of quizzes: these are programs designed to demonstrate lesser-known rules of the language. The best way to approach them is to use the hints in Appendix E, which are references to the relevant *ARM* paragraph, and to look up the answers in Appendix F only after you believe that you have solved a quiz. I have not included programming exercises; my suggestion is that you modify, extend and improve the case studies.

All the case studies, quizzes and programs in the book were compiled and executed. LaTeX-formatted text was automatically produced from the source code with a small amount of additional manual formatting. The program that transforms Ada source code to LaTeX is itself the subject of one of the case studies! Source code that is unchanged from one case study to another is usually omitted; the full source code of each executable program is available on the companion CD-ROM. Files names are given in the margin at the beginning of each program. The CD-ROM also contains Ada 95 compilers, plaintext and hypertext versions of the *ARM* and other material as specified in Appendix C. Appendix G is a guide to sources of information on Ada, both printed and electronic.

The painting on the cover is *The Railway Bridge, Argentueil* by Claude Monet. Ada has been extensively used in the construction of software for rail systems which are known for their high level of safety and reliability.

Acknowledgements

I would not have been able to learn Ada 95 without the GNAT compiler from the NYU/ACT team led by Robert Dewar and Edmond Schonberg. In particular, they deserve thanks not so much for fixing the bugs I uncovered, but for their patience with me when I tried to pass off my own misunderstandings as bugs in their compiler.

I would like to thank Michael Feldman, Kevlin Henney, Richard Riehle, Reuben Sumner, Tucker Taft and Peter Wegner for reviewing the manuscript, Tucker Taft for helping me with the fine points of the Ada language, and my editor Simon Plumtree for his support and assistance.

M. Ben-Ari
Rehovot, 1998

Preface to the online edition

This edition contains the same text as the printed book, except that: a few minor errors have been corrected; the case study in Section 16.5 has been revised; the documentation of the contents of the CD-ROM in Appendix C has been removed, as have two of the web sites listed in Appendix G.

The document has been reset in a different font and with dimensions suitable for office printers.

M. Ben-Ari
Rehovot, 2005

1 The Language for a Complex World

1.1 Programming or software engineering?

Successful computer science students often extrapolate from their demonstrated ability to write programs, and believe that the same methods and techniques will enable them to develop large software systems. Only later, when they gain experience and mature into competent engineers, do they realize that the real world does not correspond to the ideal setting of a classroom exercise.

Modern software systems are built by tens, even hundreds, of software engineers, not all of whom are as talented as you are. Even if your company has been successful in recruiting a team of highly competent engineers, a large team will suffer from inconsistencies caused by growth and rapid turnover. Throw in human personality traits such as ambition and envy, and it is a wonder that a large system can even be built!

The work of a software engineer is often the most complex in the entire project. The reason is that tasks that are implemented in software rather than hardware are precisely those that concern the entire system. Other engineers typically work on individual components and subsystems which are then integrated into a software-controlled project. For example, a mechanical engineer who designs the landing gear of an airplane is not doing a system-wide task. But the software engineer who writes the control program of the aircraft must understand the general principles of all the subsystems. Even in fields not traditionally considered engineering, the same situation holds: the software engineer working on a stock exchange must be familiar with the basic principles of the instruments being traded, together with the communications system and requirements of the traders using the system. Software engineering is significantly more complex than just programming, and it should not be surprising that different tools are needed.

1.2 Reliable software engineering

It is socially acceptable for a reporter to miss his deadline because his word-processor refused to save his article. It is socially acceptable for a saleswoman to falter during a long-sought meeting with a client's top executive because the operating system on her laptop crashed. Just blame the computer.

The structure of the software market for personal computers has caused reliability to be consciously neglected. Software packages are compared by lists of features (201 or just 200), performance (46 seconds is better than 47 seconds), and occasionally price. Vendors feel pressured to

bring new versions to market, regardless of the reliability of the product. They can always promise to fix the bug in the next version.

But word-processors, presentation graphics and interactive games are not the only type of software being developed. Computers are now controlling the most complex systems in the world: airplanes and spacecraft, power plants and steel mills, communications networks, international banks and stock markets, military systems and medical equipment. The social and economic environment in which these systems are developed is totally different from that of packaged software. Each project pushes back the limits of engineering experience, so delays and cost overruns are usually inevitable. A company's reputation for engineering expertise and sound management is more important in winning a contract than a list of features. Consistent, up-to-date, technical competence is expected, not the one-time genius of a startup.

Above all, system reliability cannot be compromised. The result of a bug is not just a demoted reporter or the loss of a sales commission. A bug in a medical system can mean loss of life. The crash of a communications system can disrupt an entire economy. The failure of a spacecraft can cost hundreds of millions of dollars. In fact, all these have occurred because of software faults.

1.3 Language in software engineering

Software engineering is the term used to denote the ensemble of techniques for developing large software projects. It includes, for example, managerial techniques such as cost estimation, documentation standards, configuration management and quality assurance procedures. It also includes notations and methodologies for analysis, design and testing of the software itself. There are many of us who believe that programming languages play an essential role in software engineering.

In the end, a software system is successful if it—the 'code' of the program—executes reliably and performs according to the system requirements. The best-managed project with a superb design is a failure if the delivered 'code' is no good. Thus, managerial techniques and design methodologies must be supplemented by the use of a programming language that supports reliable programming.

The alternative to language support for reliability is 'bureaucracy'. The project manager must write conventions for interfaces and specifications of data representations, and each convention must be manually checked in code inspections. The result is that all the misunderstandings, to say nothing of cases where conventions were ignored by clever programmers, are discovered at best when the software components are integrated, and at worst after the software is delivered. Why can't these conventions be formalized in the programming language and checked by the compiler? It is strange that software engineers, who make their living from automating systems in other disciplines, are often resistant to formalizing and automating the programming process itself.

1.4 Ada for software engineering

The Ada language is complex because it is intended for developing complex systems, and its advantages are only apparent if you are designing and developing such a system. Then, and only

then, will you have to face numerous dilemmas, and you will be grateful for the Ada constructs that help you resolve them:

- How can I decompose the system? Into packages that can be flexibly structured using containment, hierarchial or client-server architectures.
- How can I specify interfaces? In a package specification that is separate from its implementation.
- How can I describe data? With Ada's rich type system.
- How can I ensure independence of components of my system? By using private types to define abstract data types.
- How can data types relate to one another? Either by composition in records or by inheritance through type extension.
- How can I reuse software components from other projects? By instantiating generic packages.
- How can I synchronize dozens of concurrent processes? Synchronously through rendezvous or asynchronously through protected actions.
- How can I get at the raw machine when I need to? By using representation specifications.
- How can I make the most efficient use of my expensive testing facility? By testing as much of the software as possible on a host machine using a validated compiler that accepts exactly the same standard language as the target machine.

Programming in Ada is not, of course, a substitute for the classical elements of software engineering. Ada is simply a better tool. You design your software by drawing diagrams of the package structure, and then each package becomes a unit of work that you assign to an engineer. The effects caused by incompetent engineers, or by personnel turnover, can be localized. Many, if not most, careless mistakes are caught by type checking during compilation, not after the system is delivered. Code inspections can focus on the logical structure of the program, because the consistency of the conventions and interfaces is automatically checked by the compiler. Software integration is effortless, leaving you more time to concentrate on system integration.

Though Ada was originally intended for critical military systems, it is now the language of choice for any critical system. As off-the-shelf software packages become themselves more complex and are used in critical applications, I hope that Ada will eventually be used to make them more reliable.

1.5 The development of Ada

The Ada language was developed at the request of the US Department of Defense which was concerned by the proliferation of programming languages for mission-critical systems. Military systems were programmed in languages not commonly used in science, business and education, and dialects of these languages proliferated. Each project had to acquire and maintain a development environment and to train software engineers to support these systems through decades of

deployment. Choosing a standard language would significantly simplify and reduce the cost of these logistical tasks.

A survey of existing languages showed that none would be suitable, so it was decided to develop a new language based on an existing language such as Pascal. The ensuing competition was won by a team led by Jean Ichbiah, and the language published as an ANSI/MIL standard in 1983 and as an ISO standard in 1987.

There were several unique aspects of the development of Ada:

- The language was developed to satisfy a formal set of requirements. This ensured that from the very beginning the language included the necessary features for its intended applications.
- The language proposal was published for scientific review *before* it was fully implemented and used in applications. Many mistakes in the design were corrected before they became entrenched by widespread use.
- The standard was finalized early in the history of the language, and facilities were established to validate compilers against the standard. Adherence to the standard is especially important for training, software reuse and host/target development and testing.

A decade later, a second round of language design was performed by a team led by S. Tucker Taft. This design followed the same principles as the previous one: proposals by the design team were published and critiqued, and finally accepted as an international standard in 1995. This language is called Ada 95 when it is necessary to distinguish it from the previous version called Ada 83. Ada 95 supersedes Ada 83, and almost all Ada 83 programs will run unchanged in Ada 95. Aside from the following short section, this book will present the Ada language as defined in 1995.

1.6 From Ada 83 to Ada 95

For the benefit of readers familiar with Ada 83, we summarize the major differences between that language and Ada 95.

- Derived types were of limited expressive power and use in Ada 83. In Ada 95, *tagged* derived types are the basis for type extension and dynamic polymorphism, which are the constructs required for object-oriented programming.
- Packages in Ada 83 could be nested, but this introduced excessive dependencies among the packages. Child packages in Ada 95 can be used to construct subsystems as flexible hierarchies of packages that share abstractions (private types).
- The rendezvous is an extremely successful construct for task-to-task communication, but is rather inefficient for mutual exclusion. Ada 95 introduces protected objects that are far more efficient for simple synchronization.
- New numeric types have been introduced: modular types for unsigned arithmetic and decimal fixed point types for financial calculations.
- Ada 95 extends support for hardware interfacing as well as for programming in a mixed-language environment. Data types are defined for machine words, as well as for objects shared

with libraries and modules written in Fortran, Cobol and C.

- Libraries for character and string handling, and for mathematical function are now standardized, and international character sets are supported.
- The language is divided into a core that must be supported by all implementations and into special-needs annexes that are optional. The core language is of a reasonable size; extensions which are of interest in specialized applications only can be implemented as needed. There are annexes for systems programming, real-time systems, distributed systems, information systems, numerics (scientific computation), and for systems with additional safety and security requirements.

Aside from these major extensions, many local improvements have been made to Ada. These include relaxing rules that were overly stringent and improving the explanations in the standard.

1.7 The Ada Reference Manual

The Ada 95 programming language is defined by a document called *Ada 95 Reference Manual: Language and Standard Libraries*, International Standard ISO/IEC-8652:1995, published in book form as (Taft & Duff 1997). Henceforth we will refer to this document as the *ARM*. The *ARM* is intended to serve as a *contract* between software engineers writing applications in Ada and compiler implementors. If you write your program according to the rules in the *ARM*, the executable file generated by the compiler will execute as described in the *ARM* on any computer, running any operating system. In other words, an Ada program is *portable*. In practice, the *ARM* does not specify every aspect of the language so as not to overly constrain the implementors of a compiler, but even this freedom is carefully documented.

The Ada approach of creating a standard as early as possible should be contrasted with the situation in other languages such as Fortran or C which were extensively used before standardization. By then, quirks of the first implementations had become part of the language, and the spread of dialects made it extremely difficult to port existing programs. The danger of early standardization is that constructs that are of little use or are difficult to implement may be required by the standard. The Ada 95 development process dealt with this danger by arranging for compiler developers and applications software engineers to study and test constructs before the standard was finalized.

Like any contract, the *ARM* is written in very precise language, and the term 'language lawyer' is often used for people who are experts at interpreting the document. Like any contract, however, the rules are binding upon you even if you don't exactly understand the text of the rule! Fortunately, many of the most difficult parts of the *ARM* are intended for compiler implementors, and you don't need to understand them in order to write applications. For example:

> 8 ... The type *determined by* a subtype_mark is the type of the subtype denoted by the subtype_mark. **§3.2.2**

is a rule that only a language lawyer could love, but:

> 6 ... The letter E of an exponent can be written either in lower case or in upper case, with the same meaning. **§2.4.1**

is a rule which needs neither explanation nor paraphrasing.

The *ARM* consists of thirteen sections, fourteen annexes and an index. The sections and annexes are divided into clauses and subclauses, which are in turn divided into numbered paragraphs. We use the notation §C(P) to refer to paragraph(s) P within clause or subclause C. Framed extracts from the *ARM* are identified by the clause number in the margin and the paragraph number(s) within the text. An ellipsis (...) indicates omissions from a paragraph. The extracts from the *ARM* are intended to simplify your initial understanding. Always read the full *ARM* paragraphs when you start to program in Ada.

Most of the text of the *ARM* is *normative*, meaning that an implementation must conform to the text. The last five annexes and the index are *informative*, meaning that they are not part of the contract. For example, Annex §K 'Language-Defined Attributes' is a convenient list of all the attributes defined throughout the text. It is useful if you are searching for an attribute, but the rules of the language are determined by text where the attribute is defined. In addition, the text contains *Notes* and *Examples* which are informative. The examples are usually too simple to be useful, but the notes are quite important because they describe rules that are *ramifications* of other rules, meaning that they can be deduced from other rules. Sometimes you have to be an experienced language lawyer to understand the ramification; for all practical purposes you will be not be led astray if you trust the notes.

The clauses and subclauses have a common structure §1.1.2, starting with the syntax of constructs in BNF. (The complete syntax is collected in Annex §P.) For an Ada programmer, the most important clauses are:

Legality rules These rules prescribe what it means for a program to be *legal*, that is, to compile successfully. For example, the statement:

```
case C of
  when 'A' => Proc1(Z);
  when 'A' => Proc2(Y);
  when others => null;
end case;
```

is not legal because:

> 10 Two distinct discrete_choices of a case_statement shall not cover the same value. **§5.4**

Static semantics These clauses define the compile-time effect of a construct and are also used extensively for defining terms. A large part of §3 on declarations and types consists of static-semantics rules that define compile-time properties of types. An example of a static-semantics rule is that in a for-loop statement:

```
for N in 1 .. 10 loop
  ...
end loop;
```

the loop parameter N is implicitly declared at compile-time:

> 6 A loop_parameter_specification declares a *loop parameter*, ... **§5.5**

Dynamic semantics Here is where the action is. These clauses tell you what a construct does at run-time. The following clause should come as no surprise:

> 5 For the execution of an if_statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, then the corresponding sequence_of_statements is executed; otherwise none of them is executed. **§5.3**

The other clauses are mostly of interest to implementors, though you will eventually want to study them for a full understanding of a construct. Beware of the *Name resolution rules*. These are a subset of the legality rules which are used to disambiguate multiple possible interpretations. Since they come right after the syntax clauses, you may tend to start reading them, but they can be quite complex and common sense will serve in most cases. For example, the name resolution rule of an if-statement is:

> 4 A condition is expected to be of any boolean type. **§5.3**

This means that if the condition in an if-statement is a function call Func(X,Y) and there are two such functions, one returning Boolean type and one returning Integer type, the compiler will select the one returning Boolean. But you could have guessed that anyway!

Do not attempt to read the *ARM* serially from start to finish, as it is arranged in a logical order regardless of level of difficultly. For example, §3.4 is extremely difficult, while §5 and §6 are relatively easy. When you develop a familiarity with the *ARM* style, you will be able to display selections in hypertext format alongside your source code as you program.

Finally, if you wish to become a language lawyer, you will need to study the *Annotated Ada 95 Reference Manual*, which justifies the rules, and explains obscure points or implementation aspects. This document is not normally needed by applications engineers.

2 A Simple Ada Program

The purpose of this chapter is to present the constructs of Ada used for writing simple programs. Even if you are already familiar with Ada, or if the language seems to be just another version of Pascal, I suggest that you carefully read the sections on types and subtypes.

2.1 Case study: country of origin

The case study used in this chapter is a program that reads the name of a car manufacturer and prints the country of origin. The program is admittedly artificial and incomplete,[1] but it will serve its purpose of presenting the basics of Ada.

```
 1  --                                                    -- File: COUNTRY1
 2  -- Read the manufacturer of a car and write the country
 3  -- of origin of the car.
 4  --
 5  with Ada.Text_IO; use Ada.Text_IO;
 6  procedure Country1 is
 7
 8    type Cars is
 9      (Ford, Chevrolet, Pontiac, Chrysler, Dodge,
10      Rover, Rolls_Royce,
11      Peugeot, Renault, Citroen,
12      BMW, Volkswagen, Opel,
13      Honda, Mitsubishi, Toyota,
14      Daewoo, Hyundai
15      );
16
17    type Countries is (US, UK, France, Germany, Japan, Korea);
18
19    function Car_to_Country(C: Cars) return Countries is
20    -- Associate country with car using a case statement
21    begin
22      case C is
23        when Ford | Chevrolet..Pontiac | Chrysler..Dodge
                                        => return US;
24        when Rover | Rolls_Royce      => return UK;
```

[1] My car is designed in one country and assembled in a second by a company based in a third country!

```
25          when Peugeot..Citroen      => return France;
26          when BMW..Opel             => return Germany;
27          when Honda..Toyota         => return Japan;
28          when Daewoo | Hyundai      => return Korea;
29       end case;
30    end Car_to_Country;
31
32    S: String(1..80);                          -- Input string
33    Last: Integer;                             -- Last character of input string
34    Car: Cars;
35
36 begin
37    loop
38       Put("Enter the make of the car: ");
39       Get_Line(S, Last);
40       exit when Last = 0;
41       Car := Cars'Value(S(1..Last));
42       Put_Line(Cars'Image(Car) & " is made in " &
43          Countries'Image(Car_to_Country(Car)));
44    end loop;
45 exception
46    when Constraint_Error =>
47       Put_Line(S(1..Last) & " is not recognized");
48 end Country1;
```

2.2 Structure of a program

Ada uses the term *subprogram* to refer to either a procedure or a function. The main subprogram §10.2(7), here Country1, is a subprogram that is not nested within any other unit.

A subprogram §6.3(2) consists of a specification, followed by a *declarative part* and a *handled sequence of statements*. The specification ‡6[2] declares the subprogram Country1. The declarative part ‡7–35 consists of two type declarations Cars and Countries, the declaration of a function Car_to_Country and three variable declarations: S, Last and Car. There is no required order for the declaration of entities, except that a declaration must appear before it is used. The executable part of the subprogram is a sequence of statements ‡37–44: in this case a single loop statement with other statements nested within.

Non-nested units such as a main subprogram are called *library units* §10.1.1. A *compilation unit* is a library unit,[3] optionally preceded by a *context clause* §10.1.2. The context clause ‡5 lists packages that are imported into the unit. Packages—the Ada construct for creating modules—will be discussed in depth in Chapter 5; until then we will only use a context clause to import input–output packages, in particular, Ada.Text_IO §A.10 for characters and strings.

[2] This notation will be used for line numbers within program listings.

[3] A compilation unit can also be a subunit (Section 12.2.)

There are no input–output statements in Ada; instead, standard libraries are provided that use general language constructs such as subprograms. For example, Get_Line ‡39 is a procedure which is specified in §A.10.7(18–20). Note here the use of the operator "&" for string concatenation ‡42.

2.3 Statements

Ada has the usual simple and compound executable statements §5.1: assignment, if, case, loop (both for-loop and while-loop) and even a goto statement §5.8.

Note that a loop need not have a termination condition; this is particularly useful when writing servers that are not intended to terminate. The exit statement §5.7 may be used to leave a loop at any point within its execution. There is a special syntax **exit when** that makes the termination condition particularly readable ‡40. If you want to leave a nested loop, you can use a loop identifier §5.5(2), §5.7(2).

For-loops are used when you can compute the number of iterations in advanced. The following statement will print the country of origin of all the cars declared by the type:

```
for C in Cars loop
  Put_Line(Cars'Image(C) & " is made in " &
    Countries'Image(Car_to_Country(C)));
end loop;
```

The loop parameter C is a constant §3.3(19), which is implicitly declared §5.5(6) and its scope is restricted to the loop statement §8.1(4).

A return statement ‡23–28 §6.5 may be used to leave a procedure or function, even from within a loop.

It is often claimed—as a legacy of Pascal programming—that the exit from a loop should be either at its beginning or at its end. I believe that the position of the exit is not as important as the clarity of the termination condition; see Ben-Ari (1996*a*).

There is a rich syntax for case statements §5.4, as (artificially) demonstrated in the case study ‡22–29. The basic rule is that each possible value of the expression must be covered by exactly one of the alternatives. An **others** alternative is allowed if you do not want to explicitly list an action for all alternatives.

2.4 Predefined exceptions

1 … An *exception* represents a kind of exceptional situation; an occurrence of such a situation (at run-time) is called an *exception occurrence*. To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called *handling* the exception.	**§11**

The main subprogram in the case study has an exception handler ‡45–47. The predefined exception Constraint_Error §11.1 will be raised if you enter a string that is not the name of a car declared in the type declaration. If this occurs, the normal execution will be abandoned and the sequence of statements ‡47 in the exception handler will be executed instead. The exception handler in this program prints an error message; then the execution of the program will terminate.

There are four predefined exceptions: Constraint_Error, which you will encounter frequently, as any run-time violation of the type system will raise this exception; Program_Error, which is raised in unusual situations, such as 'falling-off' the end of a function without executing a return statement; Storage_Error if the program runs out of memory; Tasking_Error, which is raised for errors in the multitasking constructs (Chapter 13).

2.5 Types

Types are the heart and soul of Ada. The executable statements of the language are almost indistinguishable from those of other languages, but the type system is rich and, yes, complex. After introductory material on types in Ada, we will return to the use of types in the case study.

Why types?

Why are types so important? Experience has shown that the key to good software design is not in the algorithms—however complex they may be—but in the data. You can write small programs to manipulate raw hardware memory such as numbers and bytes, but any large program is necessarily modelling some complex applications area such as banking, medicine or transportation. The only way to write software that can be understood, validated and maintained is to model the application within the program. In Chapter 5 we will discuss how to decompose the program to reflect the structure of the application; in this section we will discuss types which are used to model the data of the application. Chapter 6 is on object-oriented programming, which is a method of modelling both the data itself and the operations on the data.

Types are not merely descriptive. The Ada language provides for *type checking*. Since each type is modelling a different kind of real-world data, you cannot assign a value of one type to a variable of another type. If A contains a number of type Apples and O contains a number of type Oranges, the statement A := A+O almost certainly contains an error, and an Ada compiler is required to reject such statements.

The advantage of type checking is a significant reduction in the number of logic and run-time errors in the software. Figure 2.1 shows a hierarchy of errors:

Compile-time errors These are the errors that are easiest to find and correct. I characterize them as errors that need not be reported to your boss! Some compile-time errors like missing punctuation are trivial; others, like visibility errors caused by misplaced declarations, are more difficult to find, and you may need to consult an expert.

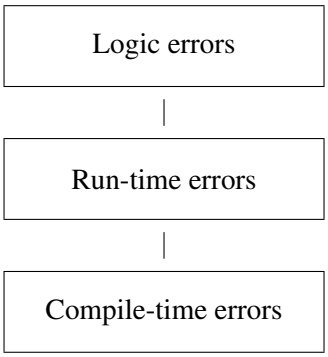

Figure 2.1: Hierarchy of errors

Run-time errors These are errors that cause the program to stop running and to display an error message.[4] Examples are trying to index an array beyond its bounds and trying to dereference a null pointer. In a language that supports run-time checking, these errors are relatively easy to diagnose because you know where the program stopped and why. If the error is not caught when it occurs, it can cause memory assigned to unrelated variables to be 'smeared'. The run-time error is then disguised as a logic error, and can be exceedingly difficult to diagnose because the code that accesses the smeared variables is likely to be correct.

Logic errors This is the term used for errors that manifest themselves as incorrect functioning of a running program. For example, you might compute a thermostat setting that has no physical meaning even though the value is within the range of an integer variable. These are extremely serious because they are often found only after the system is delivered and installed.

The philosophy of Ada is to push errors *down* the hierarchy: to turn logic errors into run-time errors and if possible into compile-time errors. For critical software systems, logic errors are not merely embarrassing and expensive, they can be truly dangerous. Because it helps prevent logic errors, the Ada type system is an excellent technological tool for producing safe and reliable software.

Effective use of the Ada type system requires an investment of effort on your part:

- During the design of the software system, you must select types that best model the data in the system and that are appropriate for the algorithms to be used. If you write a program in which every variable is declared as Integer, Float or Character, you might as well use a simpler language than Ada.
- Occasionally, type checking rules simply must be ignored. For example, a sequence of raw bytes received from a communications line is to be interpreted as a structured message. You must learn to carefully use the Ada constructs for bypassing type checking (Sections 4.6, 8.3).
- Certain language rules are checked at run-time and the overhead must be taken into account in the project design when you select the hardware. Optimizing compilers can reduce this

[4] Of course, run-time errors are extremely problematical in embedded systems which do not 'print an error'.

overhead to a minimum; in extreme cases, run-time checks can be suppressed in critical inner loops (Section 4.4).

The latter two points are often raised as objections to type checking, but in practice they turn out to be of relatively minor importance. My advice is to invest time in a comprehensive study of the Ada type system, an investment that will repay you many times over in lower rates of logic and run-time errors in your programs.

Definition of types

> 1 A *type* is characterized by a set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. ... **§3.2**

Some types, like Integer, are predefined by the language. The set of values of the type Integer is implementation-defined. Primitive operations will not be fully explained until Chapter 6, but they include the predefined operations. Assignment is defined for all types (except limited types as described in Chapter 5). Other predefined operations of Integer type are the usual arithmetical and relational operations as described in §4.5. In addition to the predefined types, types like Cars can be explicitly defined by the programmer.

Types are collected into *classes*, which group together types with similar properties.

Enumeration types

Type Cars ‡8–15 is an *enumeration type* §3.5.1. The names listed in the type declaration are the *enumeration literals*, which are the values of the type. Predefined operations on enumeration types are assignment, the relational operations such as "=" and "<" and *attributes* §4.1.4. These are predefined operations, usually values or functions, associated with each type in the family. A list of all predefined attributes is given in §K; some attributes that are defined for enumeration and integer types are given in the following table, where T is a type, V is a value of the type, N is an integer and S is a string:

T'First	First value of T
T'Last	Last value of T
T'Pred(V)*	Previous value before V
T'Succ(V)*	Next value after V
T'Pos(V)	Position of V
T'Val(N)*	Value whose position is N
T'Min(V1,V2)	Minimum of V1,V2
T'Max(V1,V2)	Maximum of V1,V2
T'Image(V)	String denoting the value V
T'Value(S)*	Value denoted by S
T'Width	Maximum length of T'Image

Attributes marked with an asterisk may raise Constraint_Error. (In certain circumstances, Pos may raise Program_Error §3.5.5(8).)

In the case study we used the attribute Cars'Image ‡42, which is a function that converts a value of type Cars into a string.[5] The converse attribute Cars'Value ‡41 is a function that converts strings to values. The notation S(1..Last) in ‡41 selects the characters of S up to index Last. If the function parameter is not a string that can converted to a value of type Cars, the exception Constraint_Error will be raised.

The fundamental principle of type checking is that a value of one type may not be assigned to a variable (or parameter) of another type §5.2(4), §6.4.1(3). This violation of type checking results in a compile-time error. Thus we cannot write:

```
Last := Chevrolet;
```

because we are attempting to assign a value of type Cars to a variable of type Integer. Similarly, we cannot write the following function call:

```
Car_to_Country(63);
```

because the assignment of an actual parameter of type Integer to a formal parameter of type Cars is not legal.

Objects

> 1 ... An *object* of a given type is a run-time entity that contains (has) a value of the type. **§3.2**

> 13 An object is either a *constant* object or a *variable* object. The value of a constant object cannot be changed between its initialization and its finalization, whereas the value of a variable object can be changed. ... **§3.3**

All objects must be declared and explicitly given a type. An initial value is optional for a variable, but is required for a constant §3.3.1(5-6):

```
Current_Car:     Cars := Opel;
The_Car_I_Want:  constant Cars := Rolls_Royce;
```

Elaboration*

> 11 The process by which a construct achieves its run-time effect is called *execution*. This process is also called *elaboration* for declarations and *evaluation* for expressions. ... **§3.1**

While we normally don't think of a declaration as being executed, it is clear that it may have a run-time effect. For example, since memory must be allocated for a variable, every variable declaration may potentially raise the predefined exception Storage_Error when memory is exhausted. In this

[5] For extensive input–output of enumeration types, it is more convenient to instantiate the generic package Enumeration_IO. However, generics is a relatively advanced concept that we do not discuss until Chapter 7.

case, we say that the exception is raised when the variable declaration is elaborated, and there are rules governing the handling of such exception occurrences.

In a language that allows initial values for objects, the declaration may be associated with an arbitrarily complex computation:

```
B: Boolean := Is_Prime(2e30 + 1);
```

Declarations are elaborated in order of their appearance:

> 7 The elaboration of a declarative_part consists of the elaboration of the declarative_items, if any, in the order in which they are given in the declarative_part. **§3.11**

Once a declaration is elaborated, it can be used in subsequent declarations:

```
B1: Boolean := Is_Prime(2e30 + 1);
B2: Boolean := not B1;
```

There is one syntactical problem that must be resolved. In Ada, as in most programming languages, it is possible to declare more than one object in a single declaration. The question arises whether the initial value is evaluated once for all objects, or once for each object:

```
A1, A2, A3: A_Type := Get_Initial_Value_From_User;
```

> 7 Any declaration that includes a defining_identifier_list with more than one defining_identifier is equivalent to a series of declarations each containing one defining_identifier from the list, with the rest of the text of the declaration copied for each declaration in the series, in the same order as the list. ... **§3.3.1**

In the example, the user would be prompted three times for initial values, as if the declarations were written:

```
A1: A_Type := Get_Initial_Value_From_User;
A2: A_Type := Get_Initial_Value_From_User;
A3: A_Type := Get_Initial_Value_From_User;
```

Name equivalence*

Given the declarations:

```
type American_Cars is (Ford, Chevrolet, Pontiac, Chrysler, Dodge);
type US_Cars is (Pontiac, Chevrolet, Chrysler, Ford, Dodge);
AC: American_Car;
UC: US_Car;
```

we cannot assign AC to UC or vice versa. Each type declaration defines a separate type. The *name*, rather than the structure, of the type determines type equivalence.

In this example, we see an example of *overloading*, where the same name is used to denote two or more entities. Context (or qualification—see Section 4.6) can be used to disambiguate the name:

```
AC := Ford;                          -- OK, refers to Ford of American_Cars
```

2.6 Subtypes

Pascal was the first language based on the concept of programmer-defined types and compile-time type checking. However, it became apparent that the Pascal rules were too restrictive. Consider a procedure to sort an array:

```
type Array_Type = array[1..100] of Integer;
procedure Sort(var A: Array_Type);
```

The procedure is restricted to sorting integer arrays of exactly 100 elements. Even passing the lower and upper bounds of the array would make no difference, because the actual parameter must be of type Array_Type.

Definition of a subtype

> 8 A *subtype* of a given type is a combination of a type, a constraint on the values of the type, and certain attributes specific to the subtype. ... **§3.2**

The constraint, that puts a restriction on the values of the type, is *checked at run-time*. In the next chapter, we will see that the specific bounds of an array are considered to be a constraint, so a single procedure Sort can be called with arrays of different sizes. In this section, we will look at subtypes in the context of enumeration types.

First a bit of syntax and terminology. When declaring an object, we supply a *subtype_indication*:[6]

§3.3.1

```
2  object_declaration ::=
     defining_identifier_list : [constant]
       subtype_indication [:= expression];
```

§3.2.2

```
3  subtype_indication ::= subtype_mark [constraint]
4  subtype_mark ::= subtype_name
```

When creating an object, we can append a constraint to a subtype name (called the subtype *mark*) and thus restrict the values that can be contained in the object. For an enumeration type, the appropriate constraint is a *range constraint*:

```
Car:         Cars;
French_Car:  Cars range Peugeot..Citroen;
German_Car:  Cars range BMW..Opel;
```

When executing an assignment statement, constraints are checked:

> 11 The value of the expression is converted to the subtype of the target. The conversion might raise an exception (see 4.6). **§5.2**

[6] The syntax for object_declaration has been simplified.

Thus:

```
French_Car := Car;                    -- Might raise Constraint_Error
French_Car := German_Car;             -- Will raise Constraint_Error
Car        := French_Car;             -- OK
```

Note that appending a constraint does not affect the *type* of an object. The second statement above is not a compile-time error, nor is French_Car := BMW! The compiler checks only that the types are the same: in this case, both variables are of type Cars. It is left to the executable code to check that the constraints are satisfied. An Ada compiler is allowed to diagnose the situation as one that necessarily raises an exception, and to emit machine code that simply raises the exception without doing the superfluous check. A friendly Ada compiler will also warn you of the inevitable error.

Declaration of a subtype

If you are planning to declare many objects with the same subtype indication, you can simply declare a subtype and then use its name §3.2.2(2). An alternate way of implementing the function associating countries with cars is:

```
 1 function Car_to_Country(C: Cars) return Countries is          -- File: COUNTRY2
 2   -- Associate country with car using subtypes
 3   subtype US_Car          is Cars range Ford..Dodge;
 4   subtype UK_Car          is Cars range Rover..Rolls_Royce;
 5   subtype French_Car      is Cars range Peugeot..Citroen;
 6   subtype German_Car      is Cars range BMW..Opel;
 7   subtype Japanese_Car    is Cars range Honda..Toyota;
 8   subtype Korean_Car      is Cars range Daewoo..Hyundai;
 9 begin
10     case C is
11       when US_Car       => return US;
12       when UK_Car       => return UK;
13       when French_Car   => return France;
14       when German_Car   => return Germany;
15       when Japanese_Car => return Japan;
16       when Korean_Car   => return Korea;
17     end case;
18 end Car_to_Country;
```

‡3–8 contain declarations of subtypes for each country by constraining the range of values of the type. A choice of a case statement can be a subtype mark; see §5.4(3), §3.8.1(4–5), §3.6.1(3).

Note that a formal parameter must be a subtype mark and not a subtype indication §6.1(15):

```
function Korean_Car_to_Country(C: Korean_Cars)
    return Countries is                    -- OK, subtype mark
function Korean_Car_to_Country(C: Cars range Daewoo..Hyundai)
    return Countries is                    -- Error, subtype indication
```

Membership tests

A membership test can be used to ask if an expression is **in** a subtype §4.5.2:

```
if C in French_Car then ...
```

If you want to check membership in a range, a subtype mark need not be given, and there is a convenient syntax for negations:

```
if C in Peugeot..Citroen then ...
if not (C in French_Car) then ...          -- OK
if C not in French_Car then ...            -- OK, nicer syntax
```

First subtype**

§6.1(15) requires that formal parameters be *subtype* marks. What about *type* marks? In order to simplify the presentation of the language, the phrase 'type or subtype' is avoided by defining types to be nameless. Instead, the identifier in a type declaration is the name of the *first subtype* §3.2.1(1) of the type. Additional (named) subtypes may of course be declared as we saw above. In normal usage, no confusion will result if we talk about the type Cars; in fact, such usage is sanctioned by the standard §3.2.1(7).

2.7 Lexical elements

We conclude this chapter with an overview of the lexical elements of an Ada program §2.

An Ada program is written in free-format:

> 1 ... The text of each compilation is a sequence of separate *lexical elements*. ... The meaning of a program depends only on the particular sequence of lexical elements that form its compilations, excluding comments. **§2.2**

Upper/lower case is not significant §2.3(5). There are reserved words such as **begin** that cannot be used for any other purpose §2.9. This book follows the style recommended in the *ARM*: reserved words in lower case and identifiers in mixed case. Certain identifiers such as Integer and String are predefined (in package Standard §A.1); they can be redefined, but normally you wouldn't do that. Comments are denoted by two minus signs and extend to the end of the line §2.7.

String §2.6 and character §2.5 literals and comments can use the character set defined by the BMP subset of ISO 10646 §2.1(4). The Latin characters have explicit names §A.3.3, so you can easily use all these characters is a program intended for an international market, even if your computer display and keyboard support only your local character set. Within the language definition, the Latin character subset (corresponding to ISO 8859-1) is used §2.1(5).

Numeric literals §2.4, as well as identifiers, can use the underscore character. Integer literals can have an exponent, and both integer and real literals can be written in any base from 2 to 16. Each of the following columns shows three equivalent numeric literals:

1000000	15	3.75
1_000_000	2#1111#	0.375E1
1E6	16#F#	2#11.11#

3 Arrays

In this chapter and the next one, we present the basic concepts of composite types (arrays and records) and access types (pointers). The main conceptual innovation in Ada is the use of types and subtypes to separate compile-time from run-time aspects of data structures.

3.1 Case study: fill and justify text

The case study is to implement a core algorithm used in word-processors:

> Read a text file and write it with the text filled (as many words as possible on a line) and justified (set flush with both margins). A word is a maximal sequence of non-space characters. Assume that the file name, the output line width and the margin size are fixed.

The following example shows (1) the input data, (2) the text after filling and (3) the text after justifying in a line of length 25:

```
The␣␣␣quick␣␣brown
␣␣fox
jumped␣␣␣␣over
␣␣␣the␣␣lazy␣dog

The␣quick␣brown␣fox
jumped␣over␣the␣lazy␣dog

The␣␣␣quick␣␣␣brown␣␣␣fox
jumped␣over␣the␣lazy␣␣dog
```

In this and other relatively long programs, we will give the source code in chunks with consecutive line numbers. The program begins with declarations of subtypes and constants and a file object Input:

```
1  --                                                    -- File: JUSTIFY
2  -- Read text from a file and write it filled and justified.
3  --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure Justify is
6
7   subtype Lines      is String(1..80);                  -- A line of text
```

```
8   subtype Index    is Integer range 0..Lines'Last;                -- Extra zero value
9
10  -- Constant file name and margins.
11  File_Name:       constant String := "example.tex";
12  Margin:          constant String(1..10) := (others => ' ');
13
14  -- Compute number of characters in printed line.
15  Width:           constant Index := Lines'Length - 2*Margin'Length;
16
17  Input:           File_Type;
```

Get_Word reads the next word from the input file; it will not be difficult to understand if you study it together with the specification of Ada.Text_IO §A.10.

```
18  procedure Get_Word(                                  -- Get next word from input
19      Word: out Lines;                                 -- The next word
20      Length: out Index;                               -- Its length
21      EOF:  out Boolean) is                            -- True if eof encountered
22    C: Character;                                      -- Character buffer
23  begin
24    Length := 0;
25    EOF := False;
26    loop                                               -- Skip leading ends-of-line, blanks
27      if End_Of_File(Input) then
28        EOF := True;
29        return;
30      elsif End_Of_Line(Input) then Skip_Line(Input);
31      else
32        Get(Input, C);
33        exit when C /= ' ';
34      end if;
35    end loop;
36
37    loop                                               -- Read characters until space or EOL
38      Length := Length + 1;
39      Word(Length) := C;
40      if Length > Width then                           -- Truncate word longer than line
41        Skip_Line(Input);
42        Length := Width;
43        return;
44      end if;
45      exit when End_Of_Line(Input);
46      Get(Input, C);
47      exit when C = ' ';
48    end loop;
49  end Get_Word;
```

The most difficult part of the program is the function Insert_Spaces, which performs the justification. The function receives the output buffer Line, the Length of the valid data in the buffer and a Word count. It returns a string containing the justified line. The algorithm is implemented by creating an array S of the spaces ‡59–60, 67–74 to be inserted *after* each word. This array is initialized ‡60 to the minimum one space, plus the number of extra spaces that can be evenly distributed among the words. Any remaining spaces are then distributed between successive words, starting from the left or right on alternate lines ‡68 to avoid excessive space on one side of the page. The new line is built in Buffer ‡76–85. The slice construct ‡81–82 is explained in the next section.

```
50  function Insert_Spaces(                          -- Insert extra spaces in output line
51    Line:    Lines;                                -- Current output line
52    Length:  Index;                                -- Length of current output line
53    Words:   Index)                                -- Number of words in line
54      return Lines is                              -- Return modified line
55
56    Spaces:  Natural := Width-Length;                  -- Extra spaces
57    -- S is number of spaces after each word
58    -- Initially, divide extra spaces evenly among words
59    S:       array(1..Words) of Natural :=
60               (others => (Spaces / (Words-1)) + 1);
61    Buffer:  Lines := (others => ' ');                 -- Build new line here
62    K1, K2:  Index := 1;                               -- Indices for copying line
63    L:       Index;                                    -- Length of word
64
65  begin
66  -- Distribute remaining spaces alternately left and right.
67    for N in 1 .. Spaces mod (Words-1) loop
68      if Ada.Text_IO.Line mod 2 = 1 then
69        S(Words-N) := S(Words-N) + 1;
70      else
71        S(N) := S(N) + 1;
72      end if;
73    end loop;
74    S(Words) := 0;                                   -- Zero spaces after last word
75
76    for W in 1..Words loop
77      L := 1;
78      while Line(K1+L) /= ' ' loop                   -- Search for end of word
79        L := L + 1;
80      end loop;
81      Buffer(K2 .. K2+L + S(W)) :=                   -- Move word and extra spaces
82        Line(K1 .. K1+L) & (1..S(W) => ' ');
83      K1 := K1 + L + 1;
84      K2 := K2 + L + S(W);
```

```
85      end loop;
86      return Buffer;
87    end Insert_Spaces;
```

Put_Word implements the fill operation by inserting a Word into the output buffer Line at index Position ‡109. If there is no room for the new word, Insert_Spaces is called ‡100 to justify the line, and then the buffer is reset ‡103–105 before inserting the word.

```
88    procedure Put_Word(                          -- Put word input output line
89      Word:      in Lines;                       -- The word to insert
90      Word_Length: in Index;                     -- Its length
91      Words:     in out Index;                   -- Current number of words
92      Line:      in out Lines;                   -- Output line buffer
93      Position:  in out Index) is                -- Position to insert word
94    begin
95      -- Note that Position points past the trailing space
96      -- Print full line
97      if Position - 1 + Word_Length > Width then
98        if Words >= 2 then
99      -- Make sure there are at least two words for inserting spaces
100         Line := Insert_Spaces(Line, Position-2, Words);
101       end if;
102       Put_Line(Margin & Line(1..Width));
103       Line := (others => ' ');
104       Position := 1;
105       Words := 0;
106     end if;
107
108     -- Append word to line and update counters
109     Line(Position..Position+Word_Length) := Word(1..Word_Length) & ' ';
110     Position := Position + Word_Length + 1;
111     Words := Words + 1;
112   end Put_Word;
```

The main subprogram ‡129–133 opens the input file, calls the main loop and then closes the input file. The main loop ‡113–127 was written as a separate procedure so that its variables are encapsulated in a local, rather than in a global, scope. The main loop is very simple: it gets the next Word ‡122 and then places it in the output Buffer ‡124. If EOF is returned from Get_Word ‡123, the current line is flushed before returning from the procedure ‡126.

```
113   procedure Main_Loop is
114     Word:      Lines;                          -- Word buffer
115     Word_Length: Index;                        -- Its length
116     EOF:       Boolean;                        -- True if EOF encountered
117     Buffer:    Lines := (others => ' ');           -- Output line buffer
118     Position:  Index := 1;                     -- Next position to insert
119     Word_Count:  Index := 0;                   -- Number of words
```

```
120    begin
121      loop
122        Get_Word(Word, Word_Length, EOF);
123        exit when EOF;
124        Put_Word(Word, Word_Length, Word_Count, Buffer, Position);
125      end loop;
126      Put_Line(Margin & Buffer(1 .. Position-1));          -- Flush last line
127    end Main_Loop;
128
129  begin
130    Open(Input, In_File, File_Name);
131    Main_Loop;
132    Close(Input);
133  end Justify;
```

We now discuss the array constructs used in the program.

3.2 Array types

Unconstrained arrays

An array is defined by giving the number of dimensions, their types and bounds, and the subtype of the component. All these characteristics, except the bounds of the dimensions, are declared in an *unconstrained array definition* §3.6(3,15). For example, the unconstrained array subtype[1] String is predefined §A.1(37) as follows:

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

The type String has one dimension whose type is Integer, constrained to positive values by the subtype Positive §3.4.5(13); the component type is Character. (The type is also declared to be Packed, see §13.2.) The bounds of any particular string are *not* part of its type. This is indicated by the syntax **range** <>, where the last two symbols are pronounced 'box'. For example, the declaration of the predefined procedure Put for strings §A.10.1(48) is:

```
procedure Put(Item : in String);
```

The procedure can be called with any string as its actual parameter.

To create a string object, you must give an *index constraint* §3.6.1 which specifies the bounds for each dimension. There are three ways that you can specify the index constraint:

- Explicitly append an index constraint as part of the subtype indication in an object declaration:

  ```
  S: String(1..80);                                    -- See Country1
  ```

- Declare a (constrained) subtype ‡7 and then declare a object of the subtype ‡61:

  ```
  subtype Lines is String(1..80);
  Buffer: Lines := (others => ' ');
  ```

[1] That is, the first subtype (Section 2.6) of the array type is unconstrained. For all practical purposes, you can talk of the *type* String.

- If an initial value is given for an array ‡11 (not necessarily a constant array), the compiler will determine the index constraint §3.3.1(9) from the number of characters in the initial value:

```
File_Name: constant String := "example.tex";
Current_File_Name: String := File_Name;
```

Operations on arrays

Assignment and the equality operators are defined for array types; that is, you can assign an array object to another one, or compare two array objects, provided, of course, that they are of the same subtype. An *indexed component* §4.1.1 is obtained by appending a parenthesized expression (or sequence of expressions for multi-dimensional arrays) to the name of an array object.

For any array object A the following attributes are defined §3.6.2:

A'First	The lower bound of the index of A
A'Last	The upper bound of the index of A
A'Range	The range A'First..A'Last
A'Length	The number of components in A

Note that A'First and A'Last are indices, not components:

```
A( (A'First + A'Last) / 2 );          -- Middle element of the array
( A(A'First) + A(A'Last) ) / 2;       -- Average of first and last elements
```

These attributes are also defined for constrained array subtypes like Lines, but not, of course, for unconstrained subtypes like String. There are also versions for multi-dimensional arrays §3.6.2(4,6,8,10).

It is impossible to over-emphasize the importance of using attributes. Once a constrained array subtype or an array object has been declared, subsequent declarations and statements should use the attributes so that changes in the array bounds are automatically reflected in the source code. For example, given the following declarations ‡7,8,15:

```
subtype Lines is String(1..80);
subtype Index is Integer range 0..Lines'Last;
Width: constant Index := Lines'Length - 2*Margin'Length;
```

changing 80 to 120 in the declaration of Lines does not require any additional change to the source code.

As a matter of style, I do not recommend using the following Pascal-like sequence of declarations, even though it is legal:

```
Line_Width: constant Integer := 80;
subtype Line_Index is Integer range 1..Line_Width;
subtype Lines is String(Line_Index);
```

The reason is that the attributes supply predefined names for these entities: Lines'Last and Lines'-Range, and there is no point in adding (and documenting!) duplicate names.

Aggregates

Recall that a type consists of a set of values and a set of operations on those values. Strangely enough, most programming languages have no way of denoting a *value* of an array type! You are required to work explicitly in terms of components:

```
type Vector is array(Integer range <>) of Float;
subtype Samples is Vector(0..255);
Zero_Sample: Sample;

for S in Samples'Range loop
  Zero_Sample(S) := 0.0;
end loop;
```

An *aggregate* denotes a value of a composite type. Array aggregates have a very rich syntax §4.3.3. The simplest form is to use **others** to give every component the same value ‡12:

```
Margin: constant String(1..10) := (others => ' ');
Zero_Sample: Samples := (others => 0.0);
```

Named array aggregates can be used to explicitly associate index values with component values:

```
1  Car_to_Country: constant array(Cars) of Countries :=          -- File: COUNTRY3
2    ( Ford..Dodge                 => US,
3      Rover..Rolls_Royce          => UK,
4      Honda..Toyota               => Japan,
5      Peugeot | Renault | Citroen => France,
6      BMW | Volkswagen | Opel     => Germany,
7      Daewoo..Hyundai             => Korea);
```

Note that parentheses are used delimit both parameters and indices[2] so the *function* Car_to_Country can be replaced with an *array* without otherwise modifying the program.

When named aggregates are used, the order in which the component associations are written is not significant; **others** is allowed as a final component association to cover index values not explicitly named:

```
Step: Samples := (32..63 => 0.5, 0..31 => 1.0, others => 0.0);
```

Positional array aggregates associate component values according to the sequence in which they appear. You cannot mix positional and named notation, but an **others** choice is allowed as the final component in a positional aggregate §4.3.3(3):

```
Initial_Sample: Samples := (0.1, 0.2, 0.3, 0.4, others => 0.0);
```

The examples we have shown use aggregates for initial values; however, it is important to understand that, syntactically, aggregates are expressions and can be used in any context that an expression is allowed, such as in an assignment statement, a return statement or as an actual parameter. Furthermore, the components of the aggregate are also expressions and can be dynamically computed. In the following function, the sequence of statements is a single return statement whose expression is a positional array aggregate, all of whose components are dynamic expressions that depend on the formal parameters:

[2]Unlike Pascal, C, C++ and Java which use brackets to delimit indices.

```
subtype Vector3 is Vector(1..3);

function "+"(Left, Right: Vector3) return Vector3 is
begin
  return ( Left(1)+Right(1), Left(2)+Right(2), Left(3)+Right(3) );
end "+";
```

Aggregates for n-dimensional arrays are constructed from subaggregates of $n-1$-dimensional arrays §4.3.3(6):

```
type Matrix is array(Integer range <>, Integer range <>) of Float;
M: Matrix(1..3, 0..2) :=
  ((1.0, 2.0, 1.0), (2=>1.0, 1=>0.5, 0=>0.0), (others => 0.0));
```

Aggregates are always to be preferred over explicit loops because of the check that the number of components of the aggregate matches the context in which it is used:

```
function "+"(Left, Right: Vector3) return Vector3 is
  Temp: Vector3;
begin
  for N in 1..2 loop                          -- Sorry !!
    Temp(N) := Left(N)+Right(N);
  end loop;
  return Temp;
end "+";
```

Slices and sliding

Slices reduce the need for explicit loops.

§4.1.2

2 slice ::= prefix(discrete_range)

5 A slice denotes a one-dimensional array formed by the sequence of consecutive components of the array denoted by the prefix, corresponding to the range of values of the index given by the discrete_range.

‡81–82 show the assignment of a string obtained by concatenating a slice of Line and an aggregate to a slice of Buffer.

The following function creates a palindrome from a string by copying it twice to the target string, the first copy in the original order of the source string ‡9, followed by a second copy in reverse order ‡10–12. A slice is used to denote the first half of the string.

```
1   --                                                          -- File: PALIN
2   -- Create a palindrome from a string.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure Palin is

6    function Palindrome(S: in String) return String is
7      T: String(1..2*S'Length);
8    begin
9      T(1..S'Length) := S;                          -- Slice as a variable
10     for N in S'Range loop
11       T(T'Length - (N-S'First)) := S(N);
12     end loop;
13     return T;
14   end Palindrome;
15
16   S1: String := "Hello world";
17   S2: String(100..100+2*S1'Length-1) := Palindrome(S1);
18   S3: String(1..2*S2'Length) := Palindrome(S2);
19
20 begin
21   Put_Line(S1);
22   Put_Line(S2);
23   Put_Line(S3);
24 end Palin;
```

A slice can be used either as a variable which is the target of an assignment as shown above, or as an expression. Consider the following program for swapping the halves of an even-length array:

```
1   --                                                          -- File: SWAP
2   -- Swap halves of a string.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure Swap is
6    S: String := "HelloWorld";
7    Temp: String := S(1..S'Length/2);
8  begin
9    S(1..S'Length/2) := S(S'Length/2+1..S'Length);
10   S(S'Length/2+1..S'Length) := Temp;
11   Put_Line(S);
12 end Swap;
```

In ‡7 the slice is an expression, in ‡10 the slice is a variable, while in ‡9 one slice is the source expression of the assignment statement and the other is the target variable. Note that the subtype of the target is String(1..5), which is not the same as the subtype of the source: String(6..10). Assignment is permitted if the types are the same, and the subtypes are *convertible*.

11	The value of the expression is converted to the subtype of the target. The conversion might raise an exception (see 4.6).	§5.2

For arrays, the compiler will automatically convert array bounds as long as the number of components is the same in both the source and the target.

37	If the target subtype is a constrained array subtype, then a check is made that the length of each dimension of the value of the operand equals the length of the corresponding dimension of the target subtype. The bounds of the result are those of the target subtype.	§4.6

The operation is called *sliding*, because we can think of sliding the indices of the source slice to match the indices of the target slice.

3.3 Constrained array subtypes and objects*

Unconstrained arrays are flexible because you can declare a subprogram with formal parameters of the unconstrained subtype, and then call the subprogram with actual parameters that are of any subtype obtained by constraining the type. Very often, however, the nature of the problem is such that the bounds will be identical for all arrays of the type. In this case, you can reduce the number of names in the program and simplify the implementation of parameter passing by declaring a *constrained array subtype* §3.6(5,16):

```
type Telephone_Key is (
  One, Two, Three, Four, Five, Six, Seven,
  Eight, Nine, Star, Zero, Hash);
type Key_State is array(Telephone_Key) of Boolean;
Pressed: Key_State := (others => False);

type Spatial_Transform is array(1..3, 1..3) of Float;
```

Of course, objects and formal parameters of type Key_State or Spatial_Transform are necessarily constrained and cannot have further constraints appended §3.6.1(5):

```
First_Row: Key_State(One..Three);                    -- Error!
```

Even if an array subtype is constrained, you should always use attributes so that expressions need not be modified if the bounds are changed. The following four ways of specifying the range of a loop traversing the array Pressed are all correct, but are progressively more robust to possible changes in the definition of the array:

```
for K in One .. Hash          loop
for K in Telephone_Key        loop
for K in Key_State'Range      loop
for K in Pressed'Range        loop
```

A further shortcut is possible if you want to declare a single array. This is the only case in Ada where an object can be declared without giving an explicit type name §3.3.1(2). Since the type is *anonymous* §3.2.1(7), the array cannot be directly used as an actual parameter because all formal parameters have named types. Single array objects are frequently used to declare a global table of constants:

```
Sine_Table: constant array(0..90) of Float := (0.0, ..., 1.0);
```

3.4 Type conversion*

In Ada, type conversion is allowed only in carefully defined situations that will not break the type system §4.6. One permissible case is that arrays that 'look' the same are *convertible* §4.6(9–12); that is, they can be converted to each other §4.6(36–39):

```
Sine_Table: constant array(0..90) of Float := (0.0, ..., 1.0);

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

V1: Vector := Vector(Sine_Table);
V2: Vector(180..270) := Vector(Sine_Table);
```

Note that Sine_Table has no named type, so it is not possible to convert another array to its type.

3.5 Operations on one-dimensional arrays*

Most languages allow you to perform operations such as concatenation on strings. Since a string is nothing more than a one-dimensional array of characters (with a special syntax for literals), Ada generalizes these operations.

> 3 The concatenation operators & are predefined for every nonlimited, one-dimensional array type *T* with component type *C*. They have the following specifications: **§4.5.3**
>
> 4 **function** "&"(Left : *T*; Right : *T*) **return** *T*
> **function** "&"(Left : *T*; Right : *C*) **return** *T*
> **function** "&"(Left : *C*; Right : *T*) **return** *T*
> **function** "&"(Left : *C*; Right : *C*) **return** *T*

The reason for the four functions is to save us from having to define a one-component aggregate; instead, a value of the component type can be concatenated to an array:

```
'X' & "-Files";                 -- This is better than ...
(1 => 'X') & "-Files";          -- ... this
```

(For another example, see ‡110 of the fill-and-justify case study.)

Note that ('X') is not legal as a positional aggregate of one component because it cannot be distinguished from a parenthesized expression §4.3.3(3,32). ('X')&"-Files" is legal because it is the same as 'X'&"-Files" except for the unnecessary parentheses.

Lexicographic order between two one-dimensional arrays whose component is of *discrete* type may be tested using the relational operators §4.5.2(26). The restriction to discrete components is obvious: since the arrays are compared by sequentially comparing individual components, a comparison operator on the components must be available.

The logical operators may be used on one-dimensional arrays whose component type is Boolean §4.5.1. This is not intended for bitwise operations on numbers; modular types §3.5.4 should be used instead.

3.6 The context of array aggregates**

Consider the following sequence of declarations:

```
S1: String(1..5)  := (1..5 => '*');          -- OK
S2: String(1..5)  := (2..6 => '*');          -- OK
S3: String        := (1..5 => '*');          -- OK
S4: String        := (2..6 => '*');          -- OK
S5: String        := (0..4 => '*');          -- Error

S6: String(1..5)  := (others => '*');        -- OK
S7: String        := (others => '*');        -- Error
```

The index constraints in S1 and S2 match the discrete range defined by the named component of the aggregate (sliding if necessary). The index constraints of S3 and S4 can be determined from the ranges of the aggregate. S5 is illegal because zero is not within the range of the index subtype Positive of String §4.3.3(28).

The meaning of **others** in the aggregate for S6 can be determined from the index constraint. The declaration of S7 is illegal because it is not possible to determine the bounds of the aggregate.

> 10 An **others** choice is allowed for an array_aggregate only if an *applicable index constraint* applies to the array_aggregate. An applicable index constraint is a constraint provided by certain contexts where an array_aggregate is permitted that can be used to determine the bounds of the array value specified by the aggregate. ... **§4.3.3**

§4.3.3 goes on to specify the contexts where an **others** choice is permitted, and in §4.3.3(24–27) the method for determining the bounds of an aggregate. The basic problem is this: for a named aggregate without **others** such as (1..5 => '*'), the index bounds of the aggregate are obvious. But for a positional aggregate (10.0,6.2,1.4), or for any aggregate with **others**, the index bounds cannot be deduced from the aggregate itself. Instead, they are determined from the index constraint of the object to which the aggregate is assigned. It is not essential to learn the rules in detail; if the compiler refuses to accept an aggregate, you can easily specify the bounds in more detail either in the index constraint (as for S6) or in the aggregate (as for S3). Specifying both, as in S1, should usually be avoided so that if the bounds change, you only have to change one or the other.

3.7 Parameter modes

Most programming languages define a parameter-passing mechanism such as call-by-value or call-by-reference. In call-by-value, the value of the actual parameter is copied into the variable denoted by the formal parameter, whereas in call-by-reference, the formal parameter contains a pointer to the actual parameter. In Ada, each parameter has a *mode* associated with it that defines the permitted uses of the parameter, *not* the parameter-passing mechanism. (Parameters can also be passed as *access parameters*; see Section 9.5.)

in The formal parameter is considered to be a *constant* §3.3(17), and the actual parameter is an *expression* §6.4.1(4) that is used to initialize the constant. This is the default mode if none is specified. Functions may only have **in** parameters.

out The formal parameter is an uninitialized variable §6.4.1(15). The actual parameter must be a *variable*. An **out** parameter can be used to pass data from the subprogram to the calling program.

in out This is like an **out** parameter, except that the formal parameter is initialized with the value of the actual parameter.

In the fill-and-justify case study, Get_Word uses **out** parameters ‡19–21 in order to pass data back to the calling program. Put_Word uses **in** parameters ‡89–90 for the word to be appended and its length, and **in out** parameters ‡91–93 for buffer state that is manipulated in the procedure.

As a matter of style, I explicitly write **in** for parameters of procedure declarations even though it is the default mode, because it helps document the data flow to and from the procedure. I do not write **in** mode for function parameters; since a function can only have **in** parameters, there is no decision here to document.

The formal parameter of the function Palindrome is of type String, which is an unconstrained array subtype. The rules of parameter passing §6.4.1 specify that the actual parameter is *converted* to the formal parameter. An unconstrained target takes its constraints from the source §4.6(38), so the function can be called with any string.

```
Strange: String(17..27) := "Hello World";

Put(Palindrome(Strange));
```

will print "Hello WorlddlroW elloH". Of course, this only works because we were careful to use attributes rather than absolute values for the array index expressions in the subprogram.

It is important to understand the trade-off in the Ada design: in order to allow you to write generalized subprograms, *compile-time* type checking of array bounds has been traded for *run-time* checking of constraints. By using attributes in expressions, the bounds of the actual parameters need not be explicitly passed.

Implementation of parameter modes**

The three parameter modes **in**, **out** and **in out**, are defined in terms of use rather than implementation. However, the *ARM* does specify most aspects of the implementation, and you will occasionally need to be aware of these details.

§6.2

2 A parameter is passed either *by copy* or *by reference*. When a parameter is passed by copy, the formal parameter denotes a separate object from the actual parameter, and any information transfer between the two occurs only before and after executing the subprogram. When a parameter is passed by reference, the formal parameter denotes (a view of) the object denoted by the actual parameter; reads and updates of the formal parameter directly reference the actual parameter object.

3 A type is a *by-copy type* if it is an elementary type, ...
A parameter of a by-copy type is passed by copy.

4 A type is a *by-reference type* if it is ...

10 A parameter of a by-reference type is passed by reference. ...

11 For parameters of other types, it is unspecified whether the parameter is passed by copy or by reference.

Thus numbers, enumerations and pointers are passed by copy. We have not yet studied types that are passed by reference, but these are types such as tasks that represent internal data structures rather than 'normal' data.

What is most important is the last sentence §6.2(11): the language does not specify if arrays and records are passed by copy or by reference. By aliasing two parameters, or a parameter and a global variable, it is not difficult to create a procedure whose result depends on the implementation. Such a program is not portable, and you should avoid such programming techniques.

4 Elementary Data Structures

This chapter is an introduction to the construction of data structures in Ada using arrays, records and access types (pointers). Data structures are normally implemented as abstract data types using packages and private types to be discussed in the next chapter. The case study is the implementation of a priority queue, first using arrays and then using pointers.

4.1 Case study: array priority queue

A priority queue is a data structure that stores items in such a way that retrieval the of 'highest-priority' item is can be done efficiently, even if insertion of items will be less efficient. In the case study, we assume that the items are simply integers and that higher-priority items have lower values. This is a common situation: customers in a store take numbered tickets and the lowest outstanding number is served first.

The operations supported by the queue are: Get the lowest number, Put a new number in the queue, and check if the queue is Empty.[1] A Get operation from an empty queue will raise the exception Underflow and a Put operation to a full queue will raise the exception Overflow.

```
1   --                                                    -- File: PROGPQA
2   -- Priority queue implemented as an array.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
6  procedure ProgPQA is
7
8    type Vector is array(Natural range <>) of Integer;
9    type Queue(Size: Positive) is
10     record
11       Data: Vector(0..Size);                    -- Extra slot for sentinel
12       Free: Natural := 0;
13     end record;
14
15   Overflow, Underflow: exception;
16
```

[1] Style: the operations are intended to be read 'put item on queue' and 'get item from queue', and this can be formalized using named parameter associations (Section 4.3). Putting the queue parameter first would be more consistent with languages for object-oriented programming that use distinguished-receiver syntax (Appendix A).

```
17    function Empty(Q: in Queue) return Boolean is
18    begin
19       return Q.Free = 0;
20    end Empty;
21
22    procedure Put(I: in Integer; Q: in out Queue) is
23       Index: Integer range Q.Data'Range := 0;
24    begin
25       if Q.Free = Q.Size then
26          raise Overflow;
27       end if;
28
29    --  Sentinel search for place to insert
30       Q.Data(Q.Free) := I;
31       while Q.Data(Index) < I loop
32          Index := Index+1;
33       end loop;
34
35    --  Move elements to free space and insert I
36       if Index < Q.Free then
37          Q.Data(Index+1..Q.Free) := Q.Data(Index..Q.Free-1);
38          Q.Data(Index) := I;
39       end if;
40       Q.Free := Q.Free+1;
41    end Put;
42
43    procedure Get(I: out Integer; Q: in out Queue) is
44    begin
45       if Q.Free = 0 then
46          raise Underflow;
47       end if;
48       I := Q.Data(0);
49       Q.Free := Q.Free-1;
50       Q.Data(0..Q.Free-1) := Q.Data(1..Q.Free);
51    end Get;
52
53    Q: Queue(10);                                  --  Create queue of size 10
54    I: Integer;                                    --  Element of the queue
55    Test_Data: array(Positive range <>) of Integer :=
56       (10, 5, 0, 25, 15, 30, 15, 20, -6, 40);
57
58 begin
59    for N in Test_Data'Range loop
60       Put(Test_Data(N), Width => 5);
61       Put(Test_Data(N), Q);
62    end loop;
```

```
63    New_Line;
64
65    Put(17, Q);                                   -- Test overflow (array only!)
66
67    while not Empty(Q) loop
68       Get(I, Q);
69       Put(I, Width => 5);
70    end loop;
71    New_Line;
72
73    Get(I,Q);                                     --  Test underflow
74
75 exception
76    when Underflow    => Put_Line("Underflow from queue");
77    when Overflow     => Put_Line("Overflow from queue");
78 end ProgPQA;
```

The program is tested by inserting ten elements into the queue ‡59–62 and then retrieving them ‡67–70. As listed, the program contains a test for overflow ‡65; of course, this will cause the program to terminate and must be commented-out to test the rest of the program.

4.2 Records

The queue is stored in an array Data; the data structure must also include an indication of the next Free space in Data (Figure 4.1). The type Queue will thus be implemented as a record with two

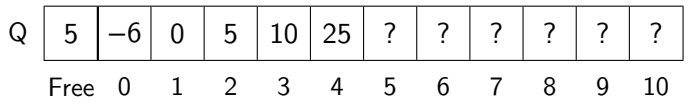

Figure 4.1: Array priority queue

components ‡9–13. The record definition also includes the declaration of a special component Size, which is called a *discriminant*. We will discuss discriminants in detail in Section 8.2; for now, you need only know that a discriminant is a read-only component of a record whose value is supplied by a constraint when a record object is declared ‡53. Note that the component Data includes an index constraint and that Free has a *default expression*. The default expression is an initial value given to a component whenever a record object is declared §3.8(6), §3.3.1(18).

Selection of a record component is done using dotted notation §4.1.3. The selected component is itself an object or value, and further index or selection operations can be applied as appropriate for the type of the component. Thus if Q is of type Queue, Q.Data is of type Vector and Q.Data(Index) is of type Integer.

Implementation of the array priority queue

The elements of the queue are stored in increasing order. The smallest number is in Q.Data(0) and can be retrieved in constant time. In this implementation, we 'close-up' the space that has been vacated by assigning one slice to another ‡50:

```
Q.Data(0..Q.Free-1) := Q.Data(1..Q.Free);
```

Assigning slices is more efficient than explicit loops, because on most computers a single machine instruction can move a block of bytes.

The Put operation is necessarily less efficient, because it must search for the correct place to insert a new item and move existing items to free the space. The existing items in the queue are stored in Q.Data(0..Q.Free-1). We use a sentinel search ‡30–33, where the new item I is placed in Q.Data(Q.Free) prior to beginning the search for an item greater than or equal to I. The sentinel ensures that even if I is greater than all existing items, the loop will terminate. In this case, no items need be moved.

Note that Get is implemented as a procedure rather than as a function. A function is allowed to have only **in** parameters, but here we need an **in out** parameter because the queue is modified. An alternate solution is to use access parameters (Section 9.5).

Record aggregates

Aggregates can be used to create values of a record type. Both positional and named aggregates may be used. **others** is also permitted, but is not quite so useful as it is for array aggregates, because record components are normally of different types. In an aggregate, components for the discriminants must also be given. The following examples show some legal aggregates for values of the type Queue; note that either a subaggregate or an array value may be given for the component Data:

```
V1: Vector(0..10) := (0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10);

Q1: Queue := (10, (1,2,3,4,5,6,7,8,9,10,11), 0);
Q2: Queue := (10, (1..4 => 7, others => 1), 0);
Q3: Queue := (Data => V1, others => 10);
Q4: Queue := (Size => 10, Data => (others => 1), Free => 0);
Q5: Queue := (Free => V1'First, Data=> V1, Size => V1'Length-1);
```

4.3 Parameter associations and overloading

Overloading

6 Two or more declarations are *overloaded* if they all have the same defining name and there is a place where they are all directly visible.	**§8.3**

The name Put can be used for a subprogram of the case study ‡22, even though the name already is directly visible in Ada.Text_IO and Ada.Integer_Text_IO. In fact, there are numerous overloaded subprograms with the name Put in these packages. When an overloaded name is used, the context determines which declaration is intended.

> 30 For a complete context, if there is exactly one overall acceptable interpretation ... then that one overall acceptable interpretation is chosen. Otherwise, the complete context is *ambiguous*. **§8.6**
> 31 A complete context ... shall not be ambiguous.

‡60–61 contain two calls to procedures named Put, the first with two actual parameters of integer type and the second with one parameter of integer type and one of type Queue. This type information is used for *overload resolution* §8.6, which is the algorithm used by the compiler to disambiguate a call. The first call is to Put from Ada.Integer_Text_IO §A.10.8 and the second is to the procedure declared at ‡22.

Overloading is a great convenience though also a possible source of confusion, since given a name, you have to 'search' the list of all visible names to find the correct one. The precise rules for overloading in §8.6 are extremely complicated; in practice, it is easy to use overloading. If a problem arises, you can always use additional syntax to disambiguate the use of a name:

```
Ada.Integer_Text_IO.Put(Test_Data(N), 6);
```

Parameter associations and default parameters

Most programming languages associate actual with formal parameters by position. Operating-system command languages typically use a different method: parameters are introduced by keywords, and for each parameter a default value is specified so that all parameters need not be specified.

> **§6.4**
> 5 parameter_association ::=
> [*formal_parameter*_selector_name =>]
> explicit_actual_parameter
> 7 A parameter_association is *named* or *positional* according to whether or not the *formal_parameter*_selector_name is specified. Any positional associations shall precede any named associations. ...
> 9 A subprogram call shall contain at most one association for each formal parameter. Each formal parameter without an association shall have a default_expression ...
> 10 For the execution of a subprogram call, the name or prefix of the call is evaluated, and each parameter_association is evaluated (see 6.4.1). If a default_expression is used, an implicit parameter_association is assumed for this rule. ...

In one Ada style, parameter names are chosen so that they facilitate named association. For example:

```
procedure Put(Item: in Integer; Into: in out Queue);

Put(Item => Test_Data(N), Into => Q);
```

The advantage of this style is improved readability; a disadvantage is that the source code can become too 'wordy'.

One problem with named association is that the parameter names create a dependence between the subprogram specification and the caller of the subprogram that would not otherwise exist. That is, we cannot change the formal parameter names without changing every call that uses named association!

Default parameters are extensively used in libraries where you want to supply many options, but default values are sufficient for most uses. For example, Ada.Integer_Text_Put is declared §A.10.8(11) as:

```
procedure Put(
    Item:  in Num;
    Width: in Field := Default_Width;
    Base:  in Number_Base := Default_Base);
```

Normally, you would print an integer in the default field width and the default base (decimal). Either or both can easily be changed:

```
Put(N, Width => 5);          -- From ‡69
Put(N, 5);                   -- Equivalent
Put(N, Base => 16);          -- Print in hexadecimal
```

Default parameters can cause difficulty in overloading resolution. Given the following two procedure declarations, the call Proc(5) is ambiguous:

```
procedure Proc(N: in Integer; K: in Integer := 10);
procedure Proc(M: in Integer);
```

4.4 Declaring and raising exceptions

The case study declares two exceptions ‡15:

```
Overflow, Underflow: exception;
```

Unlike predefined exceptions that are raised by the run-time system, exceptions that you declare must be explicitly raised using a **raise** statement ‡26,46.

> 3 When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is *abandoned*; that is, any portions of the execution that have not yet taken place are not performed. … Then: **§11.4**
>
> 5 If the construct is the sequence_of_statements of a handled_sequence_of_statements that has a handler with a choice covering the exception, the occurrence is handled by that handler;

In the country of origin case study, the exception was handled in the same subprogram where it was raised. Here we demonstrate another possibility: *propagating* the exception to the caller.

> 6 Otherwise, the occurrence is *propagated* to the innermost dynamically enclosing execution, which means that the occurrence is raised again in that context. **§11.4**
>
> 8 Note that exceptions raised in a declarative_part of a body are not handled by the handlers of the handled_sequence_of_statements of that body.

If you try to Get from an empty queue, the Get subprogram can only diagnose the problem. It cannot know if you did this on purpose, nor can it know what action is appropriate in this situation. (This will become clear in the next chapter, where you will learn how to encapsulate the queue.) Thus we do not handle Overflow and Underflow where they are raised; instead, they are propagated to the caller—in this case the main subprogram—and are handled there by printing an error message ‡75–77.

The following program demonstrates what is meant by the *dynamically enclosing execution*. P3 ‡8–11 is statically nested within P1, but since it is called by P2 ‡14, the handler in P2 ‡16, rather than the handler in P1 ‡21, will be executed.

-- File: PROP

```
1   --
2   --  Propagating an exception.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure Prop is
6    Ex: exception;
7    procedure P1 is
8      procedure P3 is
9      begin
10       raise Ex;
11     end P3;
12     procedure P2 is
13     begin
14       P3;
15     exception
16       when Ex => Put("Handled in P2");
17     end P2;
18   begin
19     P2;
20   exception
21     when Ex => Put("Handled in P1");
22   end P1;
23 begin
24   P1;
25 end Prop;
```

If no handler is found, the run-time system handles the exception, usually by terminating the program and printing a message. In an embedded system, you would want to handle all possible exceptions, because termination and printing are not viable system behaviors.

A certain amount of judgement is needed when using exceptions. Exceptions can almost always be avoided by using explicit if-statements and additional parameters; conversely, Ada novices often replace too many if-statements by exceptions. A good rule-of-thumb is to use exceptions for states that should rarely, if ever, occur. In our case study, you would normally check that a queue is empty before calling Get, and you would normally declare the queue to be sufficiently large to contain all the data that you intend to store. Neither overflow nor underflow should ever occur.

It is worth noting that exceptions by themselves do not ensure reliability. It is difficult to plan for unexpected situations and even more difficult to test them. A good case study of exception handling can be found in the report on the failure of the first test of the Ariane 5 rocket (Lions 1996). One of the causes of the failure was the incorrect design of an exception handler, which shut down the navigation computers instead of taking corrective action.

Optimization and Suppress**

Transformations performed by the compiler for the purpose of optimization may subtly effect the semantics of a program. For example, in the following program fragment, a compiler could 'optimize away' the creation of the variable S, provided that S is not used elsewhere in the program. In doing so, the compiler has 'optimized away' the exception that would be raised as a result of assigning 'G', the result of Char'Succ('F'), to a value of subtype Sub:

```
subtype Sub is Character range 'A'..'F';
S: Sub := Char'Succ('F');
C: Character := S;
```

A second possibility is that in moving code for purposes of optimization, an exception may not occur exactly at the place you expect.

§11.6 gives an implementation permission to perform such optimizations.

> 2 A *language-defined check* (or simply, a "check") is one of the situations defined by this International Standard that requires a check to be made at run-time to determine whether some condition is true. A check *fails* when the condition being checked is false, causing an exception to be raised. **§11.5**

§11.5(9–25) defines the checks; for example, Index_Check checks the bounds of an array value against its index constraint. A failure of this check will cause Constraint_Error to be raised.

> **§11.5**
>
> 4 **pragma** Suppress(identifier [, [On =>] name]);
>
> 8 A pragma Suppress gives permission to an implementation to omit the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a package_specification and includes a name, to the end of the scope of the named entity. If the pragma includes a name, the permission applies only to checks performed on the named entity, or, for a subtype, on objects and values of its type. Otherwise, the permission applies to all entities. If permission has been given to suppress a given check, the check is said to be *suppressed*.

Checks are not a 'debugging aid', but an essential part of the Ada language. Good optimization techniques will generally ensure that the run-time overhead is minimal. On occasion, *if you can prove* that a specific check on a specific type or variable is causing unacceptable overhead, you will want to suppress the check, even though you risk erroneous execution of the program §11.5(26).

You should not make any semantic use of pragma Suppress.

29 There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress should be used only for efficiency reasons.	**§11.5**

Two checks that are often suppressed are Overflow_Check since it is inefficient to implement without hardware support, and Elaboration_Check since it is unlikely to occur once you have successfully built and tested a system (see Section 12.4).

4.5 Case study: tree priority queue

Dynamic data structures are created using pointers and run-time allocation of memory. You can also create pointers to existing objects and subprograms, but we will defer a discussion of this topic until Section 9.2. The case study is an implementation of a priority queue using an unbalanced binary tree, where the value of the data at a node is greater than all the values in its left subtree, and less than or equal to all the values in its right subtree (Figure 4.2).

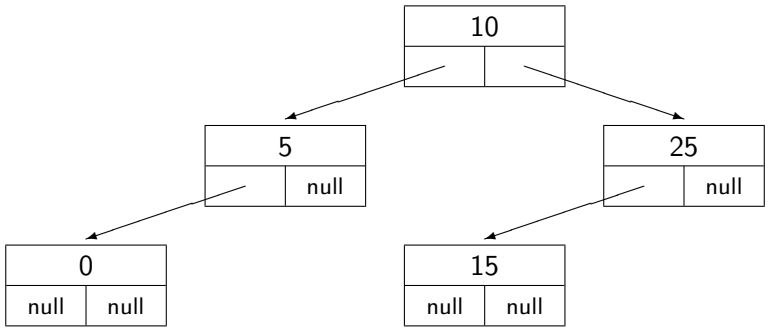

Figure 4.2: Tree priority queue

Get retrieves the smallest item in a tree by recursively traversing the leftmost path until a leaf is reached ‡47–56. The leaf is then removed and the pointer of its parent updated ‡51–52. (Note that the node becomes garbage and should be deallocated; see Section 4.6 below.) To insert an element, the tree is traversed recursively going left if the new item is less than the value stored in this node, and going right if it is greater than or equal to the value in the node ‡28–38. When a node is reached whose link is null, a new node is created for the item and linked into the tree ‡31–32. Note that the exception Overflow will be raised only if we run out of memory; this (unlikely) situation is signaled by raising the predefined exception Storage_Error, which we reraise as Overflow ‡44.

```
1   --                                                    -- File: PROGPQT
2   -- Priority queue implemented as a tree.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
6  procedure ProgPQT is
7
8    type Node;
9    type Link is access Node;
10   type Node is
11     record
12       Data: Integer;
13       Left, Right: Link;
14     end record;
15
16   type Queue is
17     record
18       Root: Link;
19     end record;
20
21   Overflow, Underflow: exception;
22
23   function Empty(Q: in Queue) return Boolean is
24   begin
25     return Q.Root = null;
26   end Empty;
27
28   procedure Put(I: in Integer; Node_Ptr: in out Link) is
29   -- Recursive procedure to insert in queue
30   begin
31     if Node_Ptr = null then
32       Node_Ptr := new Node'(I, null, null);
33     elsif I < Node_Ptr.Data then
34       Put(I, Node_Ptr.Left);
35     else
36       Put(I, Node_Ptr.Right);
37     end if;
38   end Put;
39
40   procedure Put(I: in Integer; Q: in out Queue) is
41   begin
42     Put(I, Q.Root);
43   exception
44     when Storage_Error => raise Overflow;
45   end Put;
46
```

```
47    procedure Get(I: out Integer; Node_Ptr: in out Link) is
48    -- Recursive procedure to remove from queue
49    begin
50      if Node_Ptr.Left = null then
51        I := Node_Ptr.Data;
52        Node_Ptr := Node_Ptr.Right;
53      else
54        Get(I, Node_Ptr.Left);
55      end if;
56    end Get;
57
58    procedure Get(I: out Integer; Q: in out Queue) is
59    begin
60      if Q.Root = null then
61        raise Underflow;
62      end if;
63      Get(I, Q.Root);
64    end Get;
65
66    Q: Queue;                                          -- Create queue
67    I: Integer;                                        -- Element of the queue
68    Test_Data: array(Positive range <>) of Integer :=
69      (10, 5, 0, 25, 15, 30, 15, 20, -6, 40);
70
71  begin
72    for N in Test_Data'Range loop
73      Put(Test_Data(N), Width => 5);
74      Put(Test_Data(N), Q);
75    end loop;
76    New_Line;
77
78    while not Empty(Q) loop
79      Get(I, Q);
80      Put(I, Width => 5);
81    end loop;
82    New_Line;
83
84    Get(I,Q);                                          -- Test underflow
85
86  exception
87    when Underflow    => Put_Line("Underflow from queue");
88    when Overflow     => Put_Line("Overflow from queue");
89  end ProgPQT;
```

4.6 Access types

A pointer is a value of an *access* type; it points to an object of a *designated subtype* §3.10(10). In the following declaration, Link is an access type and Node is its designated subtype. The variables L1 and L2 are access objects (pointers) that can point to objects of type Node only.

```
type Link is access Node;
L1, L2: Link;
```

As with all (nonlimited) types, pointers can be assigned and compared for equality.

As shown in ‡8–14, recursive data types are created using *incomplete type declarations* §3.10.1. The first declaration of Node simply makes its name known so that it can be used as the designated subtype of the access type declaration. The *completion* of the declaration of Node can now use the access type as a component of the record.

For reasons which will become clear in Chapter 5, we have preferred to give a separate type for the queue itself ‡16–19, even though it is implemented simply as an object of type Link. The recursive subprograms Put ‡28–38 and Get ‡47–56 overload the interface procedures Put ‡40–45 and Get ‡58–64, respectively.

> 13 For each (named) access type, there is a literal **null** which has a null access value designating no entity at all. The null value of a named access type is the default initial value of the type. ... **§3.10**

Allocators

> 1 The evaluation of an allocator creates an object and yields an access value that designates the object. **§4.8**
>
> 2 allocator ::=
>
> **new** subtype_indication |
>
> **new** qualified_expression
>
> 4 An *initialized* allocator is an allocator with a qualified_expression. An *uninitialized* allocator is one with a subtype_indication. ...

Normally, you combine the allocation of an object with setting its initial value by using an initialized allocator ‡32:

```
Node_Ptr       := new Node;                    -- OK, now initialize ...
Node_Ptr.Data  := I;
Node_Ptr.Left  := null;                        -- Default, not needed
Node_Ptr.Right := null;                        -- Default, not needed

Node_Ptr       := new Node'(I, null, null);    -- Much better!
```

Dereference

Given a value of an access type, the operation that returns the designated type is called *dereference* §4.1. Since most designated types are records, the dereference is usually followed immediately by the selection of a component; to simplify the notation, the dereference operation is implicit in Ada. Thus Node_Ptr.Left ‡34 denotes an implicit dereference of the access object Node_Ptr, followed by a selection of the component Left from the designated record.[2]

Occasionally it is necessary to do an explicit dereference, for example to assign the entire contents of one designated object to another. In addition, if the designated type is elementary rather than an array or record,[3] then explicit dereference must be used. Explicit deference is indicated by an artificial component named **all**:

```
Ptr1: Link := new Node'(1, null, null);
Ptr2: Link := new Node'(2, null, null);

Ptr1      := Ptr2;                  -- Both point at same node, or...
Ptr1.all  := Ptr2.all;              -- Contents of both nodes are equal
```

Unchecked deallocation*

Ada does not encourage explicit deallocation, because it leads to dangling pointers, which can break type checking:

```
Deallocate(Node_Ptr);               -- Not Ada
Node_Ptr.Data := I;                 -- Where does I go?
```

In the case study, discarded nodes ‡52 become ‘garbage’. An Ada implementation is allowed, but not required, to support garbage collection.

To explicitly deallocate storage obtained by an allocator, you must instantiate the generic procedure §13.11.2(3):

```
generic
  type Object(<>) is limited private;
  type Name is access Object;
procedure Ada.Unchecked_Deallocation(X : in out Name);
```

where Object is the designated subtype and Name is the access subtype. For our case study, the instantiation would be:

```
procedure Free is new Ada.Unchecked_Deallocation(Node, Link);
```

and we could then call the procedure Free with a parameter of type Link such as Node_Ptr. The storage pointed to by Node_Ptr would be freed and Node_Ptr set to null.

The use of the word Unchecked is intended to inform the reader of your program that the type system is potentially broken. Before using Unchecked_Deallocation on a project, you must study

[2] Compare this with Pascal and C, which require an explicit dereference: Node_Ptr^.Left and (*Node_Ptr).Left, respectively. C has an alternative syntax for dereference followed by selection: Node_Ptr->Left.

[3] More exactly, anything that can be a prefix.

the compiler documentation to determine if the implementation is sufficiently efficient for your requirements. Even though the procedure is part of the Ada language, the implementation is not *required* to actually reclaim storage §13.11.2(17)!

Qualification*

Syntactically, an initialized allocator contains a qualified aggregate:

```
Node_Ptr := new Node'(I, null, null);
```

> 1 A qualified_expression is used to state explicitly the type, and to verify the subtype, of an operand that is either an expression or an aggregate. **§4.7**
>
> 2 qualified_expression ::=
> subtype_mark'(expression) | subtype_mark'aggregate
> 4 The evaluation of a qualified_expression evaluates the operand ... and checks that its value belongs to the subtype denoted by the subtype_mark. The exception Constraint_Error is raised if this check fails.

For example, suppose that the following two overloaded procedures have been defined:

```
procedure Display(Item: Integer);
procedure Display(Item: Long_Integer);
```

Then a call Display(28) is ambiguous because 28 is a literal both of Integer type and of Long_Integer type. Qualification can be used to specify which procedure to call:

```
Display(Long_Integer'(28));
```

Be careful not to confuse qualification with type conversion, which performs a conversion of a value from one type to another (usually at run-time). Qualification is used purely to identify or verify a type (usually a compile-time).

5 Packages and Abstract Data Types

5.1 Modularization

A large software system must be decomposed into modules. The structures in a programming language for creating modules and for describing their interconnections determine the language's suitability for the development of complex systems. It is important to distinguish among three uses of modules:

- A module is a unit of design and management. Even before a single executable statement is written, the software will be designed as a system of modules. The project manager will then assign responsibility for the development of each module to a software engineer or team of engineers.
- A module is a unit of abstraction. To abstract is to hide details of a resource so that it can be used without knowledge of its internal structure. In a widely used terminology: a *client* uses an abstraction supplied by a *server* that is responsible for its implementation.
- A module is a physical unit of source code. *Configuration management* of a large software system requires (at the very minimum) a system for storing modules and for building versions.

In Ada, the *package* is the unit of design. The package is divided into a *specification* and a *body* which are separate physical units. In addition, they serve as elements of abstraction, separating the client interface from the implementation. Ada also supports *subunits* which enable portions of a package to exist as independent physical units while retaining their semantic status as part of the package.

Creation of abstract data types in Ada is achieved by following a paradigm that combines *private types* and packages.[1] In this chapter, we will study the mechanics of creating modules using packages, as well as techniques for creating abstract data types. We will develop a progression of six packages for priority queues; the packages encapsulate the array and tree implementations shown in the previous chapter. Each version will highlight a specific paradigm for programming with packages.

[1]Most other object-oriented languages have an abstraction construct called the *class*. Classes may or may not serve as units of design and physical units. If you are coming to Ada from another language, be careful not to make inappropriate analogies: packages are a flexible constructs for encapsulation and are not restricted to the creation of abstract data types.

5.2 Case study: priority queue package

Version 1

The following package encapsulates a priority queue implemented with an array.

```
1   --                                                          -- File: PQAV1
2   -- Priority queue abstract data type implemented as an array.
3   -- First attempt.
4   --
5  package Priority_Queue is
6
7    function Empty return Boolean;
8    procedure Put(I: in Integer);
9    procedure Get(I: out Integer);
10
11   Overflow, Underflow: exception;
12
13 end Priority_Queue;
14
15 package body Priority_Queue is
16
17   type Vector is array(Natural range <>) of Integer;
18   type Queue(Size: Positive) is
19     record
20       Data: Vector(0..Size);                    -- Extra slot for sentinel
21       Free: Natural := 0;
22     end record;
23
24   Q: Queue(100);
25
26   function Empty return Boolean is
27   begin
28     return Q.Free = 0;
29   end Empty;
30
31   procedure Put(I: in Integer) is
32     Index: Integer range Q.Data'Range := 0;
33   begin
34     if Q.Free = Q.Size then
35       raise Overflow;
36     end if;
37
```

```
38    -- Sentinel search for place to insert
39       Q.Data(Q.Free) := I;
40       while Q.Data(Index) < I loop
41          Index := Index+1;
42       end loop;
43
44    -- Move elements to free space and insert I
45       if Index < Q.Free then
46          Q.Data(Index+1..Q.Free) := Q.Data(Index..Q.Free-1);
47          Q.Data(Index) := I;
48       end if;
49       Q.Free := Q.Free+1;
50    end Put;
51
52    procedure Get(I: out Integer) is
53    begin
54       if Q.Free = 0 then
55          raise Underflow;
56       end if;
57       I := Q.Data(0);
58       Q.Free := Q.Free-1;
59       Q.Data(0..Q.Free-1) := Q.Data(1..Q.Free);
60    end Get;
61
62 end Priority_Queue;
```

The package is divided into a specification ‡5–13 and a body ‡15–62. Declarations in a specification specify the interface to the package. We can also say that the entities declared in the specification are *exported* from the package. The specification may not contain executable code such as bodies of subprograms §7.1(3). The body of a package may contain both declarations and bodies.

> 10 If a declaration occurs immediately within the specification of a package, and the declaration has a corresponding completion that is a body, then that body has to occur immediately within the body of the package. **§7.1**

Thus, the body should be considered as the implementation of resources promised in the specification.

The queue package specification contains the declarations of three subprograms and two exceptions. The package body contains the bodies for the subprograms declared in the specifications as well as the declarations of the queue type ‡17–22 and of the variable that holds the queue ‡24.

The following program tests the priority queue package: it fills the queue by calling Put ‡73–76 and then calls Get to retrieve the values which are then printed ‡79–82. The exception handlers ‡88–89 print error messages if either of the exported exceptions occur.

```
63 with Priority_Queue;
64 with Ada.Text_IO; use Ada.Text_IO;
65 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
66 procedure PQAV1 is
67
68    I: Integer;                                        -- Element of the queue
69    Test_Data: array(Positive range <>) of Integer :=
70       (10, 5, 0, 25, 15, 30, 15, 20, -6, 40);
71
72 begin
73    for N in Test_Data'Range loop
74       Put(Test_Data(N), Width => 5);
75       Priority_Queue.Put(Test_Data(N));
76    end loop;
77    New_Line;
78
79    while not Priority_Queue.Empty loop
80       Priority_Queue.Get(I);
81       Put(I, Width => 5);
82    end loop;
83    New_Line;
84
85    Priority_Queue.Get(I);                             -- Test underflow
86
87 exception
88    when Priority_Queue.Underflow => Put_Line("Underflow from queue");
89    when Priority_Queue.Overflow => Put_Line("Overflow from queue");
90 end PQAV1;
```

To use a package, the context clause of the client unit (the main subprogram or another package) must contain a 'with' context item for the package ‡63. Within the client, any entity in the package specification can be accessed by giving its expanded_name: the package name followed by the entity name; for example, Priority_Queue.Empty ‡79. The client is said to *import* these entities from the package.

Note that the client does not have access to declarations within the package body. For example, a clever programmer might wish to quickly empty the queue by writing:

```
Q.Free := 0;
```

However, this is a compilation error.

Compilation

The Ada standard specifies that a compiler shall include a library mechanism called an *environment*:

> 1 Each compilation unit submitted to the compiler is compiled in the context of an *environment* declarative_part (or simply, an *environment*), ... **§10.1.4**
> 2 The declarative_items of the environment are library_items ...
> 3 The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined.

The environment is used to ensure that both the package body and the clients are always consistent with respect to the same specification. A package specification must be compiled before the compilation of its body and before the compilation of any client. However, there is no prescribed order of compilation between the body and a client.

> 26 A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. ... **§10.1.1**

> 5 When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment; ... **§10.1.4**

In fact, you can write clients even before the the package body has been written. This is extremely useful in the management of software development by project teams, because you can directly use packages as units of work:

- The designer writes and compiles the package specification.
- An engineer is assigned to develop the body.
- Engineers who program clients refer to the package specification; they may create simplified bodies to test their modules before the final package body is completed.

The fact that all programming is done relative to *compiled* specifications means that integration of all the packages in a system is immediate. There will be few, if any, last-minute surprises caused by misunderstandings or inconsistencies in the declarations within the specifications.

Is this package an adequate implementation of a priority queue in terms of abstraction and usability?

Analyze this program yourself before continuing!

Version 2

The package is an abstract priority-queue server. As long as the semantics of the priority queue are maintained, the implementation in the body can be repeatedly modified with no effect on the clients. In fact, they need not even be recompiled!

We can demonstrate this by replacing the package body with one that implements the priority queue using a binary tree.[2]

```
1   --                                                          -- File: PQTV2
2   -- Priority queue abstract data type implemented as a tree.
3   -- First attempt.
4   --
5  package body Priority_Queue is
6
7    type Node;
8    type Link is access Node;
9
10   type Queue(Size: Positive) is                  -- Size ignored!
11     record
12       Root: Link;
13     end record;
14
15   type Node is
16     record
17       Data: Integer;
18       Left, Right: Link;
19     end record;
20
21   Q: Queue(100);
22
23   function Empty return Boolean is
24   begin
25     return Q.Root = null;
26   end Empty;
27
28   procedure Put(I: in Integer; Node_Ptr: in out Link) is
29   -- Recursive procedure to insert in queue
30   begin
31     if Node_Ptr = null then
32       Node_Ptr := new Node'(I, null, null);
33     elsif I < Node_Ptr.Data then
34       Put(I, Node_Ptr.Left);
35     else
36       Put(I, Node_Ptr.Right);
37     end if;
38   end Put;
39
```

[2]The discriminant Size is obviously not needed in the tree implementation; it is retained for compatibility with subsequent versions.

```
40    procedure Put(I: in Integer) is
41    begin
42       Put(I, Q.Root);
43    exception
44       when Storage_Error => raise Overflow;
45    end Put;
46
47    procedure Get(I: out Integer; Node_Ptr: in out Link) is
48    -- Recursive procedure to remove from queue
49    begin
50       if Node_Ptr.Left = null then
51          I := Node_Ptr.Data;
52          Node_Ptr := Node_Ptr.Right;
53       else
54          Get(I, Node_Ptr.Left);
55       end if;
56    end Get;
57
58    procedure Get(I: out Integer) is
59    begin
60       if Q.Root = null then
61          raise Underflow;
62       end if;
63       Get(I, Q.Root);
64    end Get;
65
66 end Priority_Queue;
```

The package body contains the additional subprograms Put ‡28–38 and Get ‡47–56 that are needed to implement the subprograms Put ‡40–45 and Get ‡58–64 that were promised in the specification. These additional subprograms are not exported from the package.

There is, however, a problem. The package declares a single queue; if we needed many queues, we would have to write a package for each one.[3] In more formal terms, we have created an *abstract data object*, while what is usually needed is an *abstract data type* so that we can declare multiple objects of the same queue type, pass queues as parameters, embed them in more complex data structures, and so on.

Before proceeding, we must emphasize that abstract data objects are very common. Consider, for example, a air-traffic control system: we would need a type for each individual airplane being tracked, but typically all such tracks are contained in a single data structure. A good design is to encapsulate the data structure within a package body, exporting only operations such as Get_Track, Update_Track and Delete_Track.

To create a data type, we modify the package so that the type Queue is exported.

[3]To a certain extent, generics (Chapter 7) can be used to create multiple instances of a package. However, each generic instance is a separate package and it would be impossible to pass queues as parameters.

```
1   --                                                    -- File: PQAV2
2   -- Priority queue abstract data type implemented as an array.
3   -- Second attempt.
4   --
5  package Priority_Queue is
6
7    type Vector is array(Natural range <>) of Integer;
8    type Queue(Size: Positive) is
9      record
10       Data: Vector(0..Size);                    -- Extra slot for sentinel
11       Free: Natural := 0;
12     end record;
13
14   function Empty(Q: in Queue) return Boolean;
15   procedure Put(I: in Integer; Q: in out Queue);
16   procedure Get(I: out Integer; Q: in out Queue);
17
18   Overflow, Underflow: exception;
19
20 end Priority_Queue;
21
22 package body Priority_Queue is
23
24   function Empty(Q: in Queue) return Boolean is
25   begin
26     return Q.Free = 0;
27   end Empty;
28
29   procedure Put(I: in Integer; Q: in out Queue) is
30     Index: Integer range Q.Data'Range := 0;
31   begin
32     if Q.Free = Q.Size then
33       raise Overflow;
34     end if;
35
36   -- Sentinel search for place to insert
37     Q.Data(Q.Free) := I;
38     while Q.Data(Index) < I loop
39       Index := Index+1;
40     end loop;
41
42   -- Move elements to free space and insert I
43     if Index < Q.Free then
44       Q.Data(Index+1..Q.Free) := Q.Data(Index..Q.Free-1);
45       Q.Data(Index) := I;
46     end if;
```

```
47      Q.Free := Q.Free+1;
48    end Put;
49
50    procedure Get(I: out Integer; Q: in out Queue) is
51    begin
52      if Q.Free = 0 then
53        raise Underflow;
54      end if;
55      I := Q.Data(0);
56      Q.Free := Q.Free-1;
57      Q.Data(0..Q.Free-1) := Q.Data(1..Q.Free);
58    end Get;
59
60  end Priority_Queue;
```

Note carefully the differences between a package implementing a data type and one implementing a data object. A package implementing a data object contains the declaration of a variable:

```
Q: Queue(100);
```

In other words, the package has a *state*, meaning that data is associated with the package; calls to the subprograms modify the data, changing the state. Since the state is associated with the package, the exported subprograms need not carry it in a parameter:

```
procedure Put(I: in Integer);
```

On the other hand, a package declaring a type must not have a state,[4] because the subprograms of the package are used to process many different objects declared to be of the type. The subprograms must include a parameter specifying which object the subprogram is to process ‡15:

```
procedure Put(I: in Integer; Q: in out Queue);
```

Note that any subprogram that modifies an object must declare the object parameter to be of mode **in out**.

In the following test program, the data type Queue is used. An object of the type is declared ‡66 and the imported operations of the package are called with the object as the actual parameter ‡74, 78, 79, 84. Of course, any number of objects could have been declared.

```
61  with Priority_Queue;
62  with Ada.Text_IO; use Ada.Text_IO;
63  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
64  procedure PQAV2 is
65
66    Q: Priority_Queue.Queue(10);          -- Create queue of size 10
67    I: Integer;                           -- Element of the queue
```

[4] There may be some common state associated with the ensemble of objects of the type—what other languages call *class variables*. This is easily implemented in Ada by declaring variables in the package body.

```
68    Test_Data: array(Positive range <>) of Integer :=
69      (10, 5, 0, 25, 15, 30, 15, 20, -6, 40);
70
71  begin
72    for N in Test_Data'Range loop
73      Put(Test_Data(N), Width => 5);
74      Priority_Queue.Put(Test_Data(N), Q);
75    end loop;
76    New_Line;
77
78    while not Priority_Queue.Empty(Q) loop
79      Priority_Queue.Get(I, Q);
80      Put(I, Width => 5);
81    end loop;
82    New_Line;
83
84    Priority_Queue.Get(I,Q);                    -- Test underflow
85
86  exception
87    when Priority_Queue.Underflow => Put_Line("Underflow from queue");
88    when Priority_Queue.Overflow => Put_Line("Overflow from queue");
89  end PQAV2;
```

Analyze this program yourself before continuing!

Version 3

The package declared a data *type*, but we have lost all abstraction by declaring the type in the package specification! Any modification of the specification potentially invalidates all the clients. If the representation of the type in the package specification is modified, all uses of resources of the package by clients must be checked to see if they need to be modified (and tested).

Furthermore, since the implementation of the type is not hidden, unintended manipulation of objects of the type can be done:

```
Q.Free := 0;
```

This creates unnecessary coupling between the data type and its clients, and Murphy's Law ensures that such dependencies are discovered after the programmer has left the company!

Now we have a dilemma: if the type is declared in the package specification, it is not abstract, but if it is declared in the package body, clients cannot declare objects. Why don't we try the solution used for subprograms: declare the interface to the type in the specification ‡7 and the implementation in the body ‡19–24?

```
1   --                                                          -- File: PQAV3
2   -- Priority queue abstract data type implemented as an array.
3   -- Third attempt.
4   --
5  package Priority_Queue is
6
7    type Queue(Size: Positive);
8
9    function Empty(Q: in Queue) return Boolean;
10   procedure Put(I: in Integer; Q: in out Queue);
11   procedure Get(I: out Integer; Q: in out Queue);
12
13   Overflow, Underflow: exception;
14
15 end Priority_Queue;
16
17 package body Priority_Queue is
18
19   type Vector is array(Natural range <>) of Integer;
20   type Queue(Size: Positive) is
21     record
22       Data: Vector(0..Size);                    -- Extra slot for sentinel
23       Free: Natural := 0;
24     end record;
25
26   ...
27 end Priority_Queue;
```

Analyze this program yourself before continuing!

5.3 Private types

Version 4

This is an excellent package: the clients see the existence of the type, but its implementation is hidden in the package body. There is only one minor problem: clients of this package cannot be compiled!

To see why this must be true, consider what must be done when compiling:

```
Q1, Q2: Priority_Queue(100);
```

The compiler must allocate memory for the variables Q1 and Q2. From the specification alone, the compiler cannot know that a value of type Priority_Queue must be allocated Size+3 integers. Furthermore, compiling an assignment statement such as:

```
Q1 := Q2;
```

also requires that the size of these variables be known. In other words, the compiler needs certain information about the representation of a type that the programmer does not.

Given a choice between programming at a high level of abstraction and being able to compile a program, it is not surprising that the solution is to weaken abstraction so that the program can be compiled.[5]

A package specification is divided into two parts: the visible part and the private part. The reserved word **private** indicates the boundary between the two parts. Declarations in the visible part of a specification are accessible to clients; declarations in the private part of a specification are not accessible to clients.

> 6 The first list of declarative_items of a package_specification of a package ... is called the *visible part* of the package. The optional list of declarative_items after the reserved word **private** (of any package_specification) is called the *private part* of the package. ... **§7.1**
>
> 7 An entity declared in the private part of a package is visible only within the declarative region of the package itself .... In contrast, expanded names denoting entities declared in the visible part can be used even outside the package; ...

Within the visible part, you can declare a *private type*. The *completion* of a private type by a *full type declaration* must be given in the private part of the same package. The predefined operations on a private type are assignment, equality and inequality; other operations must be explicitly declared. Of course within the package body, whatever operations permitted by the full type declaration are available.

The following specification shows the declaration of Queue as a private type in the visible part ‡7, and its completion as a record in the private part ‡17–21.

```
1   --                                                    -- File: PQAV4
2   -- Priority queue abstract data type implemented as an array.
3   -- Fourth attempt.
4   --
5  package Priority_Queue is
6
7    type Queue(Size: Positive) is private;
8
9    function Empty(Q: in Queue) return Boolean;
10   procedure Put(I: in Integer; Q: in out Queue);
11   procedure Get(I: out Integer; Q: in out Queue);
12
13   Overflow, Underflow: exception;
14
15 private
16   type Vector is array(Natural range <>) of Integer;
17   type Queue(Size: Positive) is
```

[5] An alternate solution is to use indirect allocation as is done in languages that use reference semantics. This is discussed further in Section 5.4 below.

```
18      record
19        Data: Vector(0..Size);                    -- Extra slot for sentinel
20        Free: Natural := 0;
21      end record;
22  end Priority_Queue;
```

The private type declares a *partial view* of the type; the *full view* of the type is declared in the private part and is accessible within the package body only. If the programmer of a client writes:

```
Q.Free := 0;
```

a compilation error would result, because the source code of the client can only access the partial view—from which we cannot conclude that Queue is a record, much less that it contains a component Free.

The designer of an Ada package can place a declaration in three places. The implications of each choice are as follows (where by 'modification' we mean a change of implementation that is semantically equivalent to the original one):

Visible part of specification The declaration is accessible to clients. A modification potentially affects all clients, which must be recompiled and verified.

Private part of specification The declaration is not accessible to any client, but the object code of a client depends on the declaration. A modification never affects the correctness of a client;[6] however, all clients must be recompiled.

Body The declaration cannot be accessed outside the body, so the clients do not depend on the declaration. If the body is modified, clients need only be relinked, not recompiled.

Placing a declaration as low as possible in this hierarchy reduces the coupling of the program modules and makes it easier to verify and maintain the system. Even avoiding recompilation can be important. A large system may take hours to compile—time that may not be available when the system is being integrated and installed.

To demonstrate that a change of type representation need not affect a client, we show a modification of the package specification that implements the queue using a binary tree. Since the visible part is not modified, the client need only be recompiled, not modified or checked. In order to export the same interface, the visible part is the same for both implementations, even though the discriminant Size ‡7 is not used in the tree implementation.

```
1   --                                                    -- File: PQTV4
2   -- Priority queue abstract data type implemented as a tree.
3   -- Fourth attempt.
4   --
5  package Priority_Queue is
6
7    type Queue(Size: Positive) is private;
8
```

[6] Well, hardly ever! See below.

```
 9    function Empty(Q: in Queue) return Boolean;
10    procedure Put(I: in Integer; Q: in out Queue);
11    procedure Get(I: out Integer; Q: in out Queue);
12
13    Overflow, Underflow: exception;
14
15  private
16    type Node;
17    type Link is access Node;
18    type Node is
19      record
20        Data: Integer;
21        Left, Right: Link;
22      end record;
23
24    type Queue(Size: Positive) is                    -- Size ignored!
25      record
26        Root: Link;
27      end record;
28  end Priority_Queue;
```

5.4 Limited types

Version 5

Suppose that we have declared two queues and then assign one to another:

```
Q1, Q2: Priority_Queue.Queue(100);

Q1 := Q2;
```

The assignment operation copies the block of memory allocated to Q2 to the memory cells allocated to Q1. If an array implementation is used, the assignment correctly makes a copy of the queue; however, if a tree implementation is used, the assignment merely copies the access values, and the two variables point to the same queue (Figure 5.1). We promised that a semantically-equivalent modification of the private part cannot affect the correctness of a client, yet here we have a case where an assignment statement in the client is not longer correct. Furthermore, predefined equality is not meaningful for *either* implementation of a priority queue (why?).

One solution to this problem is to claim that predefined assignment and equality are meaningless operations on a data structure such as a queue. The only meaningful operations are those explicitly declared: Empty, Get and Put. By declaring a private type to be *limited*,[7] we restrict the allowable operations to those explicitly declared:

[7]Limited types need not be private: task and protected types are automatically limited and any record may be declared limited.

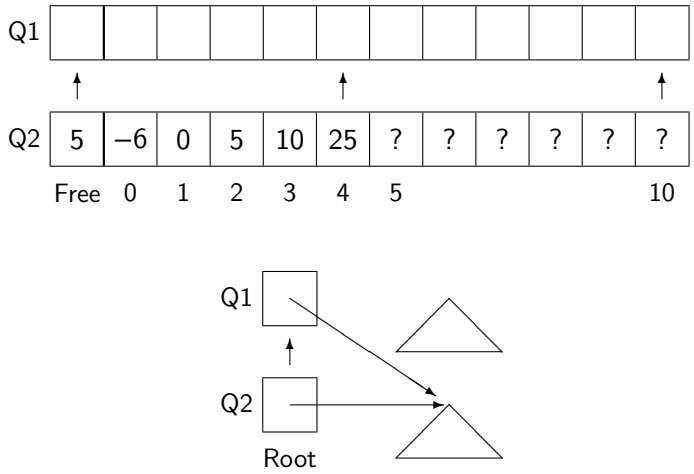

Figure 5.1: Assignment of a queue

```
1 package Priority_Queue is                                    -- File: PQTV5,PQA
2
3   type Queue(Size: Positive) is limited private;
4
5   ...
6 end Priority_Queue;
```

1	A limited type is (a view of) a type for which the assignment operation is not allowed. A nonlimited type is a (view of a) type for which the assignment operation is allowed.	**§7.5**
8	There are no predefined equality operators for a limited type.	

Limited types cannot be used in contexts such as initialization of objects §3.3.1(5), which perform an assignment operation. See §7.5(9–15) for a summary of the properties of limited types.

Like any operator, equality can be overloaded for any type,[8] and this is particularly useful for limited types that have no predefined equality. However, the assignment symbol is not an operator in Ada and cannot be overloaded, though you can declare a normal procedure:

```
type Queue(Size: Positive) is limited private;
function "="(Left, Right: Queue) return Boolean;
procedure Assign(Target: out Queue; Source: in Queue);
```

The semantics of assignment can also be changed by using controlled types; see Section 9.7.

[8] See Quizzes 13 and 14 for the exact details.

Version 6

Consider the package specification:

```
1   --                                                          -- File: PQT
2   -- Priority queue abstract data type implemented as a tree.
3   -- Queue is limited private, representation of nodes in body.
4   --
5  package Priority_Queue is
6
7     type Queue(Size: Positive) is limited private;
8
9     function Empty(Q: in Queue) return Boolean;
10    procedure Put(I: in Integer; Q: in out Queue);
11    procedure Get(I: out Integer; Q: in out Queue);
12
13    Overflow, Underflow: exception;
14
15 private
16    type Node;                                  -- Completion in body
17    type Link is access Node;
18    type Queue(Size: Positive) is               -- Size ignored!
19       record
20          Root: Link;
21       end record;
22 end Priority_Queue;
23
24 package body Priority_Queue is
25
26    type Node is                                -- Completion of type declaration
27       record
28          Data: Integer;
29          Left, Right: Link;
30       end record;
31
32    ...
33 end Priority_Queue;
```

Note that an incomplete type declaration is given for Node ‡16, but its completion has not been declared in the specification; instead, the declaration of its completion is in the body ‡26–30.

> 3 An incomplete_type_declaration requires a completion, which shall be a full_type_declaration. If the incomplete_type_declaration occurs immediately within either the visible part of a package_specification or a declarative_part, then the full_type_declaration shall occur later and immediately within this visible part or declarative_part. If the incomplete_type_declaration occurs immediately within the private part of a given package_specification, then the full_type_declaration shall occur later and immediately within either the private part itself, or the declarative_part of the corresponding package_body. **§3.10.1**
>
> 5 The only allowed uses of a name that denotes an incomplete_type_declaration are as follows:
>
> 7 as the subtype_mark in the subtype_indication of an access_to_object_definition; ...

The reason this works is that objects of an access type such as Link have the same representation regardless of what they point to.[9] We can freely change the representation of Node in the package body without even recompiling the clients. All the client can do is allocate a queue that is a pointer to the root, and pass the queue as a parameter.

This final version of the package gives the highest level of abstraction for a data type, similar to the abstract data object of the first version. Representing an object as a pointer to the actual data is called *indirect allocation*. Many object-oriented languages such as Java, Eiffel and Smalltalk use indirect allocation for all abstract data types. Languages of this kind are said to use *reference semantics*, as opposed to *value semantics* where objects directly contain values. References semantics can significantly simplify programming since explicit pointer manipulation is no longer needed, but other complications are introduced because value semantics are still used for elementary data types like integer and character.

The philosophy of Ada is that access types are explicitly used, but can be encapsulated in the package body. In the next chapter, we will see a similar situation, where access types must be used to implement a data structure but the clients need not be aware of the pointers.

Limited and nonlimited private types

Since assignment and equality are so troublesome, why would you want to declare a type just private and not limited private?

The answer is that private types are used not just for complex data structures, but also for simple information hiding of the components of a record. In these common cases, there is no reason to forbid predefined assignment and equality. The standard example is a complex number, which you will want to declare as private so that you can change the implementation from a Cartesian representation:

[9] Different access types can have different representations (Section 9.6), but once an access type has been declared, its representation is fixed regardless of changes in the designated type.

```
1   --                                                    -- File: COMPLEX1
2   -- Complex numbers: Cartesian representation.
3   --
4  package Complex_Numbers is
5    type Complex is private;
6    I: constant Complex;
7  private
8    type Complex is
9      record
10       Real_Part, Im_Part: Float;
11     end record;
12   I: constant Complex := (Real_Part => 0.0, Im_Part => 1.0);
13 end Complex_Numbers;
```

to a polar representation:

```
1   --                                                    -- File: COMPLEX2
2   -- Complex numbers: polar representation.
3   --
4  package Complex_Numbers is
5    type Complex is private;
6    I: constant Complex;
7  private
8    type Complex is
9      record
10       Rho: Float;
11       Theta: Float range 0.0 .. 360.0;
12     end record;
13   I: constant Complex := (Rho => 1.0, Theta => 90.0);
14 end Complex_Numbers;
```

without modifying the clients. By declaring the type to be nonlimited, you can still use predefined assignment and equality, which are clearly meaningful for both representations of the type. (This example is for demonstration only; libraries for complex numbers are predefined in Annex §G.1. See Section 10.8.)

This example also demonstrates the use of *deferred constants*. When a private type is declared, it is often useful to compare values of the type to a constant, in this case the imaginary value $\imath$ ‡6. Clients can use the deferred constant without knowing its actual value, which is specified in the full constant declaration given in the private part ‡13.

> 3 A deferred constant declaration that is completed by a full constant declaration shall occur immediately within the visible part of a package_specification. For this case, the following additional rules apply to the corresponding full declaration: **§7.4**
> 4 The full declaration shall occur immediately within the private part of the same package;
> 5 The deferred and full constants shall have the same type;
> 6 ... The constant itself will be constrained, like all constants; ...

6 Type Extension and Inheritance

6.1 Case study: discrete event simulation

Simulations are invariably used during the design of large embedded systems. You cannot 'debug' the design of a rocket by launching rocket after rocket! It takes months of time and tens of millions of dollars to build each rocket, so extensive simulation is the only way to develop enough confidence in the design to build and launch one with a reasonable chance of success.

The case study in this chapter is a framework for a simulation of a rocket. Needless to say, we will omit all the physical calculations that would require domain-specific knowledge. The choice of a rocket is arbitrary; the framework can be used for any simulation.

The method used is discrete event simulation. In this method, events are generated and placed on a queue (Figure 6.1).

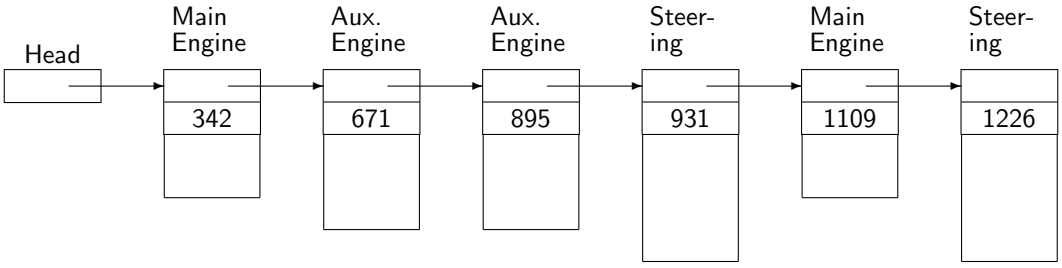

Figure 6.1: Event queue

Each event is time-stamped with the time at which it is to 'occur'. The program does not attempt to maintain a physical clock.[1] Instead, the events are ordered by time, and the program simply removes the event whose occurrence is 'soonest' in the future, sets the simulated clock to the event's time and performs the simulation of the event. This is easily done by maintaining a priority queue, where higher priority is given to earlier events.

The events themselves would normally be generated by additional tasks in a multitasking implementation. For now, we simplify the program and specify that all events are generated and placed on the queue before the simulation is commenced. A multitasking simulation will be developed in Section 14.8.

[1] In an actual system, you might have an operator's console that would display the progress of the simulation in real time, rather than trying to simulate as many events as possible in as short a time as possible.

Conveniently, we have already implemented a priority queue in the previous case study. The outline of the simulation is given by the following program fragment:

```
for All_Events loop
  Put(Create_Event, Q);
end loop;

while Queue_Not_Empty loop
  Simulate(Get(Q));
end loop;
```

We have a major problem to solve: how do we represent an event? If all events were identical, a simple record would suffice. But, as shown in Figure 6.1, different events will have different components associating with them. Some components are, of course, identical in all events: the link and the simulation time. However, the data actually required to conduct the simulation depends on the specific event type. For example, an engine event will need the fuel flow rate, while a steering event will need the deflection angles of the fins or nozzles.

The simplest solution to the problem is to include all possible components in a record. This is clearly impractical, because the records will be too large and confusing. A modification of this solution is to use variant records (Section 8.3), which may be familiar to you from languages like Pascal and C.[2] Each record will contain only the components required by the specific event type; when accessing the record, you must explicitly select the correct variant using a case statement. This solution can make maintenance difficult, because if you add an event, you must modify every case statement that selects according to the record variant.

A better solution is to use inheritance. We will define a general event type and then specialize the type for each specific event. To use the correct terminology, we will *derive* the specific event types from the *parent* type. The parent type will contain the components that are common to all events, and these will be inherited by the derived types, which will be *extended* with event-specific components.

6.2 Tagged types

We start by declaring in package Root_Event a record type Event containing a single component Time. Event is an abstract data type, because it is declared to be private ‡5 with the completion in the private part of the specification ‡17–20.

```
1  package Root_Event is                                    -- File: ROCKET
2     --
3     -- Declaration of abstract event at root of event class.
4     --
5     type Event is abstract tagged private;
6
7     -- Declare (abstract) primitive operations of an Event.
```

[2] They are called unions in C.

```
 8    function Create return Event is abstract;
 9    procedure Simulate(E: in Event) is abstract;
10
11    -- Comparison of events is common to all events in the class.
12    function "<"(Left, Right: Event'Class) return Boolean;
13
14 private
15
16    subtype Simulation_Time is Integer range 0..10_000;
17    type Event is abstract tagged
18       record
19          Time: Simulation_Time;                -- Common component of all events
20       end record;
21
22 end Root_Event;
```

The type is declared to be *abstract*. (Do not confuse this technical term to be discussed in Section 6.10 with the general concept of abstract data type.) You cannot declare an object of an abstract type §3.9.3(8). This is reasonable because a record with just the Time component doesn't actually represent an event that can be simulated. The type will serve only as the root of a class of types, one type for each event. Similarly, the operations declared for the type ‡8–9 are abstract, which means that no body will be defined for them. If you can't declare an object of the type, you don't have to have an operation for it.

The word **tagged**[3] indicates that the type can be *extended*. We now extend the abstract type Event for each concrete event type that is needed in the simulation. Package Root_Event.Engine declares three types: Engine_Event ‡25 derived from Event and two events Main_Engine_Event ‡31 and Aux_Engine_Event ‡34, which are in turn derived from Engine_Event.

```
23 package Root_Event.Engine is
24
25    type Engine_Event is new Event with private;
26
27    -- Override primitive operations of Event.
28    function Create return Engine_Event;
29    procedure Simulate(E: in Engine_Event);
30
31    type Main_Engine_Event is new Engine_Event with private;
32    function Create return Main_Engine_Event;
33
34    type Aux_Engine_Event is new Engine_Event with private;
35    function Create return Aux_Engine_Event;
36    procedure Simulate(E: in Aux_Engine_Event);
37
38 private
```

[3]The choice of the word **tagged** will be explained in Section 6.6.

```
39
40    type Engine_Event is new Event with
41      record
42        Fuel, Oxygen: Natural;
43      end record;
44
45    type Main_Engine_Event is new Engine_Event with
46      null record;
47
48    type Aux_Engine_ID is (Left, Right);
49
50    type Aux_Engine_Event is new Engine_Event with
51      record
52        Side: Aux_Engine_ID;
53      end record;
54
55  end Root_Event.Engine;
```

Packages Root_Event.Steering and Root_Event.Telemetry each declare a single type: Steering_Event ‡58 and Telemetry_Event ‡78, respectively.

```
56  package Root_Event.Steering is
57
58    type Steering_Event is new Event with private;
59
60    -- Override primitive operations of Event.
61    function Create return Steering_Event;
62    procedure Simulate(E: in Steering_Event);
63
64  private
65
66    type Commands is (Roll, Pitch, Yaw);
67    subtype Degrees is Integer range -90 .. 90;
68    type Steering_Event is new Event with
69      record
70        Command: Commands;
71        Degree: Degrees;
72      end record;
73
74  end Root_Event.Steering;
75
76  package Root_Event.Telemetry is
77
78    type Telemetry_Event is new Event with private;
79
80    -- Override primitive operations of Event.
```

```
81    function Create return Telemetry_Event;
82    procedure Simulate(E: in Telemetry_Event);
83
84  private
85
86    type Subsystems is (Engines, Guidance, Communications);
87    type States is (OK, Failed);
88    type Telemetry_Event is new Event with
89      record
90        ID: Subsystems;
91        Status: States;
92      end record;
93
94  end Root_Event.Telemetry;
```

The following syntax chart shows how a derived type is declared as a **new** version of an existing type together **with** a set of components for the derived type. Only a tagged type can be extended §3.4(5). Event is called the *parent* type §3.4 of Engine_Event, which in turn is the parent type of Main_Engine_Event.

§3.4

```
2  derived_type_definition ::=
     [abstract] new parent_subtype_indication
       [record_extension_part]
```

§3.9.1

```
2  record_extension_part ::= with record_definition
```

To maintain abstraction in the data type, the derived types are declared as *private extensions* in the visible part of the specifications, with the actual derivation done as a completion in the private part.

§7.3

```
3  private_extension_declaration ::=
     type defining_identifier [discriminant_part] is
     [abstract] new ancestor_subtype_indication with private;
```

The derivations in this program are done in *child* packages, as indicated by extended names such as Root_Event.Engine ‡23. This allows the child package access to the private part of the parent package (see Section 6.7).

The types derived from Event can be displayed in a tree (Figure 6.2). The components of a derived type consist of the components *inherited* §3.4(12) from the parent type, as well as any additional components added in the extension. Since there is no provision for removing components upon derivation, we can be sure that *every* component of a parent type, for example Oxygen in Engine_Event, is also contained in each descendant of the parent: Main_Engine_Event and Aux_Engine_Event. This fact will be of crucial importance in the discussion of type conversion (Section 6.8).

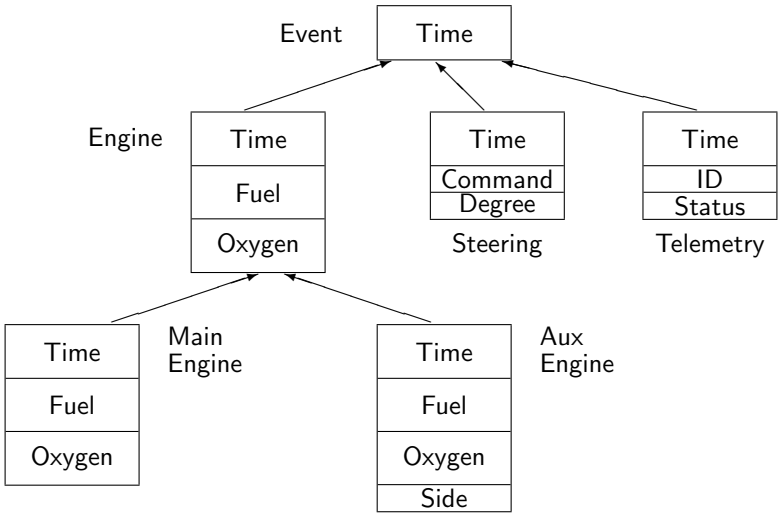

Figure 6.2: Derivation class

A tree of derived types is called a *derivation class*.

> 2 A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. The derivation class of types for a type T (also called the class *rooted* at T) is the set consisting of T (the *root type* of the class) and all types derived from T (directly or indirectly) ... **§3.4.1**

Note that the full view of the private extension can be indirectly derived from the ancestor §7.3(8); in the following example, the partial view of Main_Engine_Event is derived from Event, while the full view is derived directly from Engine_Event and only indirectly from Event.

```
package Root_Event.Engine is
   type Engine_Event is new Event with private;
   type Main_Engine_Event is new Event with private;
private
   type Engine_Event is new Event with ...
   type Main_Engine_Event is new Engine_Event with...;
end Root_Event.Engine;
```

6.3 Primitive operations

Primitive operations of a type are operations that are so closely associated with the type that they are 'carried along' with type.[4] The following section of the *ARM* defining primitive operations is extremely important, so we quote it in full for future reference though you may not understand all the concepts just yet.

[4]Primitive operations are like *methods* or *messages* in other languages.

1 An operation *operates on a type T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter (see 6.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms. **§3.2.3**
2 The *primitive subprograms* of a specific type are defined as follows:
3 The predefined operators of the type (see 4.5);
4 For a derived type, the inherited (see 3.4) user-defined subprograms;
5 For an enumeration type, the enumeration literals (which are considered parameterless functions—see 3.5.1);
6 For a specific type declared immediately within a package_specification, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same package_specification and that operate on the type;

7 Any subprograms not covered above that are explicitly declared immediately within the same declarative region as the type and that override (see 8.3) other implicitly declared primitive subprograms of the type. **§3.2.3**
8 A primitive subprogram whose designator is an operator_symbol is called a *primitive operator*.

Predefined operators are always primitive; for example, "+" is a primitive operation of the type Integer and "=" is a primitive operation of the type Event. You can declare additional primitive operations. These are subprograms which are declared *immediately in the same package specification* as the type §3.2.3(6) and that have a parameter or result of the type §3.2.3(1). A primitive operation may be declared in the private part, but it will not be accessible outside the package body and the children of the package. The restriction to the package specification is critical: it means that once an abstraction is defined in a package specification, all clients of the abstraction see exactly the same primitive operations. Subprograms with a parameter of the type can be declared elsewhere, such as in clients or in the package body, but they are not primitive.

The importance of primitive operations is that they are inherited §3.4(17) by derived types, just as record components are inherited. Thus the subprogram Simulate which is a primitive operation of Engine_Event ‡29 is inherited by Main_Engine_Event. It is as if the declaration

```
procedure Simulate(E: in Main_Engine_Event);
```

appeared just after the declaration of the type ‡31. The compiler considers a call to Simulate with an object of type Main_Engine_Event to be a call to the inherited (primitive) subprogram:

```
M: Main_Engine_Event;

Simulate(M);                          -- Calls Simulate of Engine_Event
```

6.4 Overriding an operation

All the components of the parent type are inherited by the derived type. You can add components upon derivation, but not modify or remove them. Primitive operations are inherited, but they can also be *overridden* §8.3(10). You declare an operation with the same name, replacing parameters of the parent type with parameters of the derived type. The new operation is used when called with parameters of the derived type, while the original operation is retained for calls with parameters of the parent type. For example, the declaration of Aux_Engine_Event ‡34 is followed by declarations of Create and Simulate ‡35–36 that override the inherited subprograms:

```
function Create return Aux_Engine_Event;
procedure Simulate(E: in Aux_Engine_Event);

M: Main_Engine_Event;
A: Aux_Engine_Event;

Simulate(M);                    -- Inherited Simulate of Engine_Event
Simulate(A);                    -- Overridden Simulate of
                                -- Aux_Engine_Event
```

Note that overriding subprograms are themselves primitive §3.2.3(7), and can be overridden upon subsequent derivation. As with components, an operation cannot be removed, so that, given an object of any type descended from Event, the subprogram Simulate is defined on the object: either it was overridden when the type was declared, or the last declaration of Simulate in the chain of ancestors is inherited.

We now give the package bodies for the event packages, starting with the body of the root package:

```
95  package body Root_Event is
96
97     -- Implement class wide operation.
98     function "<"(Left, Right: Event'Class) return Boolean is
99     begin
100       return Left.Time < Right.Time;
101    end "<";
102
103 end Root_Event;
```

Abstract operations cannot be called, so they have no bodies. We will discuss class-wide types in the next section.

It is in the bodies for the derived types that our simulation becomes totally artificial, as it is precisely in the creation and simulation of events that the real work takes place. Create simply constructs an aggregate with random components and Simulate prints the data contained in the event record. The body for the engine types is:

```
104 with Ada.Text_IO; use Ada.Text_IO;
105 with Root_Event.Random_Time;
106 package body Root_Event.Engine is
107
108    G: Random_Time.Generator;
109
110    function Create return Engine_Event is
111    begin
112       return ( Time => Random_Time.Random(G),
113                Fuel => Random_Time.Random(G) mod 100,
114                Oxygen => Random_Time.Random(G) mod 500);
115    end Create;
116
117    function Create return Main_Engine_Event is
118    begin
119       return Main_Engine_Event'(Engine_Event'(Create) with null record);
120    end Create;
121
122    function Create return Aux_Engine_Event is
123    begin
124       return (Engine_Event'(Create) with
125          Aux_Engine_ID'Val(Random_Time.Random(G) mod 2));
126    end Create;
127
128    procedure Put_Data(
129       E: Engine_Event; Engine_ID: String; Thrust: Integer) is
130    begin
131       Put("Time " & Integer'Image(E.Time) & ": ");
132       Put(Engine_ID);
133       Put_Line(    " engine fuel " & Integer'Image(E.Fuel) &
134                    " L, oxygen " & Integer'Image(E.Oxygen) &
135                    " L, produced " & Integer'Image(Thrust) &
136                    " KG thrust");
137    end;
138
139    procedure Simulate(E: in Engine_Event) is
140    begin
141       Put_Data(E, "Main", E.Fuel * E.Oxygen);
142    end Simulate;
143
144    procedure Simulate(E: in Aux_Engine_Event) is
145    begin
146       Put_Data(  Engine_Event(E),
147                  Aux_Engine_ID'Image(E.Side),
```

```
148                     E.Fuel * E.Oxygen / 2);
149    end Simulate;
150
151  begin
152    Random_Time.Reset(G);
153  end Root_Event.Engine;
```

and the bodies for the steering and telemetry systems are:

```
154  with Ada.Text_IO; use Ada.Text_IO;
155  with Root_Event.Random_Time;
156  package body Root_Event.Steering is
157
158    G: Random_Time.Generator;
159
160    function Create return Steering_Event is
161      use Random_Time;
162    begin
163      return (
164        Time      => Random(G),
165        Command => Commands'Val(
166          Random(G) mod (Commands'Pos(Commands'Last)+1)),
167        Degree    => (Random(G) mod 181) - 90
168        );
169    end Create;
170
171    procedure Simulate(E: in Steering_Event) is
172    begin
173      Put("Time " & Integer'Image(E.Time) & ": ");
174      Put_Line(   "Steering command " &
175                  Commands'Image(E.Command) & " to " &
176                  Integer'Image(E.Degree) & " degrees");
177    end Simulate;
178
179  begin
180    Random_Time.Reset(G);
181  end Root_Event.Steering;
182
```

```
183 with Ada.Text_IO; use Ada.Text_IO;
184 with Root_Event.Random_Time;
185 package body Root_Event.Telemetry is
186
187    G: Random_Time.Generator;
188
189    function Create return Telemetry_Event is
190       use Random_Time;
191    begin
192       return (
193          Time  => Random(G),
194          ID    => Subsystems'Val(
195             Random(G) mod (Subsystems'Pos(Subsystems'Last)+1)),
196          Status=> States'Val(Random(G) mod (States'Pos(States'Last)+1))
197          );
198    end Create;
199
200    procedure Simulate(E: in Telemetry_Event) is
201    begin
202       Put("Time " & Integer'Image(E.Time) & ": ");
203       Put_Line(   "Telemetry message " &
204                   Subsystems'Image(E.ID) & " " &
205                   States'Image(E.Status));
206    end Simulate;
207
208 begin
209    Random_Time.Reset(G);
210 end Root_Event.Telemetry;
```

The random number generator is created by instantiating the generic package Ada.Numerics.-Discrete_Random §A.5.2.

```
211    --
212    -- Instantiate random number generator for all events
213    -- Private package so it can instantiate with private Simulation_Time
214    --
215 with Ada.Numerics.Discrete_Random;
216 private package Root_Event.Random_Time is
217    new Ada.Numerics.Discrete_Random(Simulation_Time);
```

Once the package Random_Time has been declared, we can declare a Generator ‡108,158,187 and call the function Random ‡112–114,... to return a random value of type Simulation_Time. For simplicity, we use only a single generator and convert the returned random number to other numeric types.

The procedure Reset that initializes the generator is put in the sequence of statements of the package body ‡152,180,209, which is executed when the package is elaborated.

§7.2

2 package_body ::=
package body defining_program_unit_name **is**
declarative_part
[**begin**
handled_sequence_of_statements]
end [[parent_unit_name.]identifier];

6 For the elaboration of a nongeneric package_body, its declarative_part is first elaborated, and its handled_sequence_of_statements is then executed.

6.5 Class-wide types

Now that we have declared types for the events in the simulation, we can return to the problem of constructing a queue that can store events of different types. A data structure that contains items of more than one type is called *heterogeneous*, as opposed to a *homogeneous* data structure whose elements are all of one type. We cannot create a heterogenous data structure containing items of arbitrary type; this would not be compatible with the strong type checking of Ada. Instead, the design of Ada chooses a flexible intermediate approach: objects of all types derived from a parent type can be stored in a data structure. The specific type of an object is checked, if necessary, when it is used at run-time.

§3.4.1

4 Class-wide types—Class-wide types are defined for (and belong to) each derivation class rooted at a tagged type (see 3.9). Given a subtype S of a tagged type *T*, S'Class is the subtype_mark for a corresponding subtype of the tagged class-wide type *T*'Class. Such types are called "class-wide" because when a formal parameter is defined to be of a class-wide type *T*'Class, an actual parameter of any type in the derivation class rooted at *T* is acceptable (see 8.6).

Given the derivation class of types shown in Figure 6.2, the values of the type Event'Class are the union of the values of all the derived types in the class. As noted in the last sentence of §3.4.1(4) above, if a formal parameter is of class-wide type, the actual parameter can be of any type in the class. For example, the function "<" in package Root_Event ‡12 takes two parameters of the class-wide type Event'Class so it can be called with actual parameters of any event type. Of course, the body of the function ‡98–101 can only reference components common to all types in the class.

However, you can't simply declare an object of a class-wide type:

```
EC: Event'Class;                                    -- Error!
```

Recall the analogous situation with unconstrained array types such as String. It is not possible to declare an uninitialized variable or a record component of the type. However, an object can be of unconstrained type provided that it will be supplied with a constraint upon elaboration:

```
S1: String;                                         -- Error!
S2: String := "Hello world";                        -- OK
```

```
type String_Pointer is access String;
-- Pointer to string of any length
P: String_Pointer := new String(1..80);                    -- OK, see §4.8

type String_Record is
  record
    F1: String;                                            -- Error!
    F2: String(1..80);                                     -- OK
    F3: String_Pointer;                                    -- OK
  end record;

function Palindrome(S: String) return String;
-- Create a palindrome from a string of any length
```

Unconstrained array types and class-wide types are *indefinite types*.

> 23 ... A subtype is an indefinite subtype if it is an unconstrained array subtype, or ...; otherwise the subtype is a *definite* subtype (all elementary subtypes are definite subtypes). A class-wide subtype is ... an indefinite subtype. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary (see 3.3.1). A component cannot have an indefinite nominal subtype. **§3.3**

The implications for the design of a heterogeneous priority queue are as follows. The Put procedure ‡225 is declared with a parameter of class-wide type so that an event of any type in the class can be inserted into the queue, and the Get function ‡226 returns a result of class-wide type so that an event of any type can be removed from the queue. Here is the specification of the package for a heterogeneous priority queue of events:

```
218 with Root_Event; use Root_Event;
219 package Event_Queue is
220
221   type Queue is limited private;
222   type Queue_Ptr is access Queue;
223
224   function Empty(Q: in Queue) return Boolean;
225   procedure Put(E: in Event'Class; Q: in out Queue);
226   function Get(Q: Queue_Ptr) return Event'Class;
227
228 private
229   --
230   -- Implement as tree.
231   --
232   type Node;
233   type Link is access Node;
```

```
234    type Queue is
235      record
236        Root: Link;
237      end record;
238
239  end Event_Queue;
```

A node of the queue cannot directly contain an item of class-wide type; instead, Node contains ‡248 a pointer to the event ‡244 (Figure 6.3).

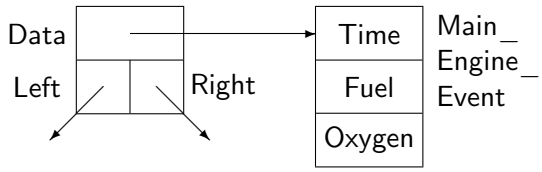

Figure 6.3: Node of a heterogeneous tree

The data structure is encapsulated in the body of the package, which is adapted from the final version of the priority queue package.

```
240  package body Event_Queue is
241
242    -- Heterogeneous data cannot be directly stored in Node.
243    -- Store a pointer to the data.
244    type Event_Class_Ptr is access Event'Class;
245
246    type Node is
247      record
248        Data: Event_Class_Ptr;
249        Left, Right: Link;
250      end record;
251
252    procedure Put(E: in Event'Class; Node_Ptr: in out Link) is
253    begin
254      if Node_Ptr = null then
255        Node_Ptr := new Node'(new Event'Class'(E), null, null);
256      elsif E < Node_Ptr.Data.all then
257        Put(E, Node_Ptr.Left);
258      else
259        Put(E, Node_Ptr.Right);
260      end if;
261    end Put;
262
```

```
263    procedure Put(E: in Event'Class; Q: in out Queue) is
264    begin
265      Put(E, Q.Root);
266    end Put;
267
268    function Empty(Q: in Queue) return Boolean is
269    begin
270      return Q.Root = null;
271    end Empty;
272
273    procedure Get(Node_Ptr: in out Link; Found: out Link) is
274    begin
275      if Node_Ptr.Left = null then
276        Found := Node_Ptr;
277        Node_Ptr := Node_Ptr.Right;
278      else
279        Get(Node_Ptr.Left, Found);
280      end if;
281    end Get;
282
283    function Get(Q: Queue_Ptr) return Event'Class is
284      Found: Link;
285    begin
286      Get(Q.Root, Found);
287      return Found.Data.all;
288    end Get;
289
290  end Event_Queue;
```

An object of type Event'Class is allocated and initialized by copying the data from E. The access to the object is used to initialize the Data component of the allocated node ‡255. Node_Ptr.Data must be explicitly dereferenced ‡256 in order to compare it with inserted event E. Similarly, to return a value of type Event'Class, Found.Data must be dereferenced ‡287.

How to avoid indirect allocation*

An alternative approach is to derive the entire class from the data structure for the node:[5]

```
type Node is tagged
  record
    Left, Right: Link;
  end record;

type Event is new Node with null record;
```

[5]We have ignored encapsulation and abstraction in these declarations.

Now, each descendant of Event simply extends Node with new components and there is no need to use indirect allocation. The problem with this technique is that you can't 'add it on' to an existing abstraction. Furthermore, if the technique is used naively with multiple queues, it is possible to put elements of distinct classes on the same queue which might cause an exception when the elements are removed. (See Section 4.4.1 of the *Rationale* for a more detailed discussion of this technique.)

Self-referential data structures can also be used to avoid indirect allocation (see Section 9.8).

6.6 Dynamic dispatching

We are finally ready to put the pieces together. The main subprogram follows the outline given at the beginning of the chapter.

```
291 with Event_Queue;
292 with Root_Event.Engine, Root_Event.Telemetry, Root_Event.Steering;
293 use Root_Event;
294 procedure Rocket is
295    Q: Event_Queue.Queue_Ptr := new Event_Queue.Queue;
296 begin
297    for I in 1..15 loop
298       Event_Queue.Put(Engine.Main_Engine_Event'(Engine.Create), Q.all);
299       Event_Queue.Put(Engine.Aux_Engine_Event'(Engine.Create), Q.all);
300       Event_Queue.Put(Telemetry.Create, Q.all);
301       Event_Queue.Put(Steering.Create, Q.all);
302    end loop;
303
304    -- Get event from queue and dispatch to Simulate procedure.
305    while not Event_Queue.Empty(Q.all) loop
306       Root_Event.Simulate(Event_Queue.Get(Q));
307    end loop;
308 end Rocket;
```

We allocate a queue object and assign the access value to Q ‡295, Create and Put events—fifteen of each type—on the queue ‡297–302, and then remove the events in order of increasing time, calling Simulate on each event ‡305–307.

There is a technical reason for the dynamic allocation of the queue: since Get must be a function rather than a procedure (as explained below) and since a function cannot have **in out** parameters, we pass an access object as an **in** parameter and modify the designated object. (A better solution is to use access parameters which we will study in Section 9.5.)

Let us follow the processing of the event from creation to simulation, paying particular attention to the types of objects and parameters. We call a function such as Telemetry.Create and then pass the returned event to the procedure Event_Queue.Put.

```
Event_Queue.Put(Telemetry.Create, Q.all);
```

What type of object is returned by the function, and what type is expected for the first formal parameter of the procedure? By examining the declaration of the function Create ‡81 in package Root_Event.Telemetry, we see that the function returns an object of type Telemetry_Event. By examining the declaration of Put ‡225 in package Event_Queue, we see that the first formal parameter is of type Event'Class. As noted in the previous section, §3.4.1(4) allows a class-wide formal parameter to be matched by an actual parameter of *any* specific type in the class. The call is legal because the actual parameter (the object returned by Telemetry.Create) is of type Telemetry_Event, which is in the class rooted at Event. Similarly, the calls to the other Create functions all return objects of types within the class rooted at Event and are acceptable actual parameters in the calls to Put.

Consider now the removal of events from the queue and the call to the procedure Simulate:

```
Root_Event.Simulate(Event_Queue.Get(Q));
```

Again let us examine what type is returned from the function Event_Queue.Get, and what type is expected by Simulate. Get ‡226 returns an object of type Event'Class:

```
function Get(Q: Queue_Ptr) return Event'Class;
```

This is as it should be because the queue is heterogeneous, so before calling Get we cannot know what the type of the returned event will be. By specifying that the return type is Event'Class, we indicate that we are willing to process an event of *any* type in the class.

The value returned by Get is the actual parameter of the call to the procedure Simulate. Obviously, the Simulate in Root_Event is not the appropriate procedure for all events in the class, because we have overridden it four times:

```
procedure Simulate(E: in Engine_Event);                -- ‡29
  -- Also for Main_Engine_Event, by inheritance
procedure Simulate(E: in Aux_Engine_Event);            -- ‡36
procedure Simulate(E: in Steering_Event);              -- ‡62
procedure Simulate(E: in Telemetry_Event);             -- ‡82
```

(In fact, Root_Event.Simulate is abstract and does not even have a body that can be called.) How can the compiler decide which subprogram to call?

The answer is that the compiler doesn't decide! Instead, it emits code that checks the specific type of the object of class-wide type, and jumps to the appropriate procedure for that object. This is called *dynamic dispatching* or simply *dispatching*. See Figure 6.4, where EV_CL denotes an actual parameter of type Event'Class.

A call to Root_Event.Simulate with an object of type Main_Engine_Event is dispatched to the subprogram inherited from Engine_Event. The other event types have explicitly overridden Simulate, and calls are dispatched to the overriding subprograms.

The advantages of dynamic dispatching are clear. The simulation loop is written once, and *never* needs to be changed (or even recompiled!), regardless of how many additional event types are added to the system. Without dispatching, we would have to explicitly include an additional component in the event record identifying the event, and then use a case statement to choose the

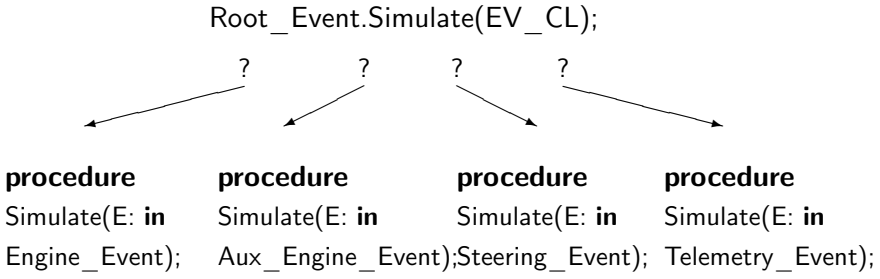

Figure 6.4: Dynamic dispatching

correct subprogram. Every additional extension would require that the loop be recompiled and tested.

You can now understand the meaning of the word **tagged**. For dispatching to work, an object must carry with it at *run-time* an indication of its type. This is conventionally called a *tag*. If a subprogram is called with an actual parameter of a specific type—the parameter is *statically tagged*—then the compiler can decide which subprogram to bind to the call. In the following declarations and statements, it is simply a matter of resolving the overloaded subprograms declared in the package Root_Event.Engine:

```
E: Root_Event.Engine.Engine_Event := Root_Event.Engine.Create;
A: Root_Event.Engine.Aux_Engine_Event := Root_Event.Engine.Create;

Root_Event.Engine.Simulate(E);          -- No question which Simulate to call
Root_Event.Engine.Simulate(A);          -- No question which Simulate to call
```

This is called *early* or *static* binding. However, if the actual parameter is *dynamically tagged*, that is, if it is of class-wide type, the compiler cannot bind the call and the binding is done dynamically at run-time. This is called *late* or *dynamic binding*.

1	The primitive subprograms of a tagged type are called *dispatching operations*. A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time; such a call is termed a *dispatching call*. ...	**§3.9.2**
4	The name or expression is *statically tagged* if it is of a specific tagged type	
5	The name or expression is *dynamically tagged* if it is of a class-wide type, ...	

Note carefully that dispatching is only done on *primitive* subprograms. These subprograms (declared in the same package specification as the tagged type) contain formal parameters of the tagged type called *controlling formal parameters*; the corresponding actual parameters are called *controlling operands*.

> 2 A call on a dispatching operation is a call whose name or prefix denotes the declaration of a primitive subprogram of a tagged type, that is, a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* ...; the corresponding formal parameter is called a *controlling formal parameter*. ... **§3.9.2**
>
> 20 For the execution of a call on a dispatching operation, the body executed is the one for the corresponding primitive subprogram of the specific type identified by the controlling tag value. ...

We wrote the dispatching call Root_Event.Simulate as an *expanded name* §4.1.3(4) rather than depending on the 'use'-clause. This emphasizes that to dispatch with a controlling operand of a class-type Root_Event.Event'Class, you (syntactically) call the primitive operation of the corresponding tagged type Root_Event.Event §3.9.2(2). Similarly, given a controlling operand of type Root_Event.Engine.Engine_Event'Class, the call Root_Event.Engine.Simulate would dispatch according to the specific engine type. Note that the function

```
function "<"(Left, Right: Event'Class) return Boolean;
```

is *not* dispatching. If a *formal* parameter is of class-wide type, the actual parameter may be of any type within the class. The function is not primitive, cannot be overridden and does not dispatch.

To summarize:

- The primitive operations of a tagged type are dispatching.
- Formal parameters of the tagged type are controlling.
- If a dispatching operation is called with controlling operands (actual parameters) of a *specific* type, the call is bound at compile-time.
- If a dispatching operation is called with controlling operands (actual parameters) of a *class-wide* type, the call is dispatched at run-time to the subprogram whose controlling (formal) parameter matches the *tag* of the operand.

6.7 Encapsulation and child packages

There are no particular encapsulation requirements for tagged types (except that primitive operations be declared in the same package specification). The following specification compiles correctly, and the resulting rocket simulation needs only minor modifications that are due to the change in the package name.

```
1  --                                                    -- File: ROCKETC
2  -- Discrete event simulation of a rocket.
3  -- Child packages are not used.
4  --
5  package Event_Package is
6    subtype Simulation_Time is Integer range 0..10_000;
```

```
7   type Event is abstract tagged
8     record
9       Time: Simulation_Time;                -- Common component of all events
10    end record;
11  function Create return Event is abstract;
12  procedure Simulate(E: in Event) is abstract;
13  function "<"(Left, Right: Event'Class) return Boolean;
14
15  type Engine_Event is new Event with
16    record
17      Fuel, Oxygen: Natural;
18    end record;
19  function Create return Engine_Event;
20  procedure Simulate(E: in Engine_Event);
21
22  type Main_Engine_Event is new Engine_Event with
23    null record;
24  function Create return Main_Engine_Event;
25
26  type Aux_Engine_ID is (Left, Right);
27  type Aux_Engine_Event is new Engine_Event with
28    record
29      Side: Aux_Engine_ID;
30    end record;
31  function Create return Aux_Engine_Event;
32  procedure Simulate(E: in Aux_Engine_Event);
33
34  type Commands is (Roll, Pitch, Yaw);
35  subtype Degrees is Integer range -90 .. 90;
36  type Steering_Event is new Event with
37    record
38      Command: Commands;
39      Degree: Degrees;
40    end record;
41  function Create return Steering_Event;
42  procedure Simulate(E: in Steering_Event);
43
44  type Subsystems is (Engines, Guidance, Communications);
45  type States is (OK, Failed);
46  type Telemetry_Event is new Event with
47    record
48      ID: Subsystems;
49      Status: States;
50    end record;
51  function Create return Telemetry_Event;
```

```
52    procedure Simulate(E: in Telemetry_Event);
53 end Event_Package;
```

Recall that in previous simulation program, the dispatching call was to Root_Event.Simulate. Here, all versions of Simulate are declared in the same package Event_Package. The type of the actual parameter in the call to Simulate ‡69 is Event'Class, so the call is dispatched to the appropriate version of Simulate in the derivation class *for the type* Event.

```
54 with Event_Queue;
55 with Event_Package; use Event_Package;
56 procedure RocketC is
57    Q: Event_Queue.Queue_Ptr := new Event_Queue.Queue;
58 begin
59    for I in 1..15 loop
60       Event_Queue.Put(Main_Engine_Event'(Create), Q.all);
61       Event_Queue.Put(Aux_Engine_Event'(Create), Q.all);
62       Event_Queue.Put(Telemetry_Event'(Create), Q.all);
63       Event_Queue.Put(Steering_Event'(Create), Q.all);
64    end loop;
65
66    -- Get event from queue and dispatch to Simulate procedure.
67    while not Event_Queue.Empty(Q.all) loop
68       Simulate(Event_Queue.Get(Q));
69    end loop;
70 end RocketC;
```

This encapsulation of the entire class of types in a single package is legal, though almost certainly not the optimal solution, because too many types are contained in one module and a modification of any one of them will cause too much recompilation and testing. However, it does emphasize that in Ada, decisions relating to encapsulation (packages) are *independent* of decisions relating to derivation classes of types. This point is worth emphasizing because many languages for object-oriented programming identify a type with its encapsulation in a ‘class’.

The limitation that a client can only see the visible part of a package is too inflexible for some applications. Consider the declaration of Root_Event.Event: on one hand we wish to restrict the accessibility of the implementation of the type (the component Time), while on the other hand, derived types should be able to access this component because computations within Simulate may be time-dependent.

The solution is to use *child packages* to form a *subsystem* of packages §10.1(9). (Library *procedures* may have child procedures §6.1(4,7).) Packages within the subsystem share abstractions by granting child packages visibility of the private parts of their specifications. Child packages are denoted syntactically by concatenating the child name to the parent’s name using dotted notation. The hierarchy of descendants may be carried to any depth.

The accessibility rules are determined by assuming that a child package is declared after the parent’s specification, but before its body (Figure 6.5). The figure is intended to show that since the children come between the specification and the body, they are can access the private part but

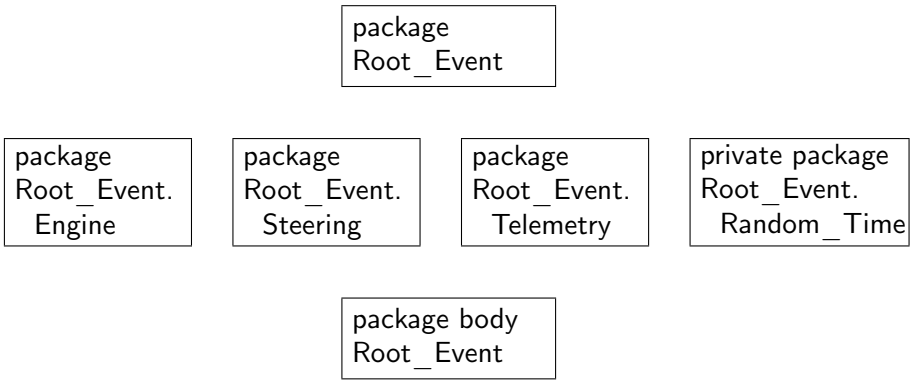

Figure 6.5: Child packages

not the body. Any package can 'with' a child package; for example, sibling packages such as Root_Event.Steering 'with' Root_Event.Random_Time. The parent body has no special privileges and must also 'with' the child if it needs to. Note that 'with'ing a child package automatically 'with's the parent package §10.1.2(6).

7	The declarative region includes the text of the construct together with additional text determined (recursively), as follows:	**§8.1**
9	If the declaration of a library unit ... is included, so are the declarations of any child units .... The child declarations occur after the declaration.	
16	The children of a parent library unit are inside the parent's declarative region, even though they do not occur inside the parent's declaration or body. This implies that one can use (for example) "P.Q" to refer to a child of P whose defining name is Q, and that after "**use** P;" Q can refer (directly) to that child.	

In the main subprogram of the simulation, we 'with' all the child packages:

```
with Event_Queue;
with Root_Event.Engine, Root_Event.Telemetry, Root_Event.Steering;
use Root_Event;
procedure Rocket is
   ...
end Rocket;
```

The context clause contains **use** Root_Event, so we can directly refer to the child package (for example Telemetry.Create ‡300) as noted in §8.1(16) above.

Since child packages have access to the private parts of its ancestors, we must prevent exportation of declarations from the private part:

```
package Root_Event.Export is
   subtype Export_Time is Simulation_Time;                    -- Error!
end Root_Event.Export;
```

There is a special rule that excludes the visible part of a child package specification from accessing the private part of its parent.

> 4 The immediate scope of a declaration in the private part of a library unit does not include the visible part of any public descendant of that library unit. **§8.2**

Consider, however, the random number generator package Root_Event.Random_Time. The package is declared as a generic instantiation of a package from the standard libraries, but in the context of our simulation program, an equivalent specification is as follows:

```
private package Root_Event.Random_Time is
   type Generator is limited private;
   function Random (Gen: Generator) return Simulation_Time;
   procedure Reset (Gen: in Generator);
private
   ...
end Root_Event.Random_Time;
```

The function Random in the visible part of the specification returns a value of type Simulation_Time that is declared in the private part of its parent, in effect exporting the type. This is not normally acceptable, because Simulation_Time was intentionally made private to prevent its exportation from the simulation subsystem.

Ada defines two types of child packages: *public* and *private*. The rule in §8.2(4) holds only for public children; the visible part of a *private* child *is* allowed access to the private part of a parent; however, to prevent unwanted exportation, a client of a private child must be within the family that already has access to the private part.

> 8 If a with_clause of a given compilation_unit mentions a private child of some library unit, then the given compilation_unit shall be either the declaration of a private descendant of that library unit or the body or subunit of a (public or private) descendant of that library unit. **§10.1.2**

In the simulation, Root_Event.Random_Time is declared to be a private child and is used only within the bodies of the packages rooted at Root_Event.

Freezing*

Could we rearrange the declarations in Event_Package so that all the type derivations are declared before the primitive operations?

```
package Event_Package is
   type Event is abstract tagged ...
   type Engine_Event is new Event with ...
   type Main_Engine_Event is new Engine_Event with ...
   type Steering_Event is new Event with ...
   type Telemetry_Event is new Event with ...
   procedure Simulate(E: in Event) is abstract;
   procedure Simulate(E: in Engine_Event);
   procedure Simulate(E: in Aux_Engine_Event);
```

```
   procedure Simulate(E: in Steering_Event);
   procedure Simulate(E: in Telemetry_Event);
end Event_Package;
```

The answer is no. For reasons that will become clear in Section 6.11, implementing extension requires that the entire set of primitive operations for a type be known when the type is extended. Thus the declarations of primitive operations must be 'close to' the declaration of the tagged type or extension. The rule is expressed in terms of a concept called *freezing* §13.14: once an entity is frozen, any declaration that would change its representation is forbidden.

7	The declaration of a record extension causes freezing of the parent subtype.	**§13.14**
16	The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see 3.9.2).	

6.8 Type conversion*

A value of a type in a derivation class can be converted to a value of another type in the class subject to the following rule.

21	... if the target type is tagged, then either:	**§4.6**
22	The operand type shall be covered by or descended from the target type; or	
23	The operand type shall be a class-wide type that covers the target type.	

This rule can be easily understood by examining Figure 6.2. We can convert a value of type Aux_Engine to a value of type Engine or to a value of type Event simply by ignoring the extra components. However, we cannot convert a value of type Engine to Aux_Engine, because it has no Side component. Similarly, a specific type can be converted to a class-wide type that *covers* it §3.4.1(9).

Consider now converting a class-wide type to a specific type:

```
E_CL:Event'Class := ...;
Eng:  Engine := Engine(E_CL);
```

E_CL contains a value of some specific type within Event'Class. If we are 'lucky' and the value of E_CL is in fact of type Engine (or Main_Engine or Aux_Engine), the conversion will succeed; otherwise, the conversion will fail and raise Constraint_Error §4.6(42,57). You would not do such a conversion unless you have reason to believe that the conversion will succeed. Alternatively, you can use a membership test §4.5.2(30) to check the type of the class-wide value at run-time:

```
if E_CL in Engine'Class then
   Eng := Engine(E_CL);                    -- OK
else
   ...                                     -- Do something else
end if;
```

Extension aggregates

Though a value of a parent type cannot be converted to a value of a type derived from it, it is possible to create a value of the derived type by supplying the additional components that are 'missing' from the parent type.

> 1 An extension_aggregate specifies a value for a type that is a record extension by specifying a value or subtype for an ancestor of the type, followed by associations for any components not determined by the ancestor_part. **§4.3.2**
> 2 extension_aggregate ::=
> (ancestor_part **with** record_component_association_list)
> 3 ancestor_part ::= expression | subtype_mark
> 6 For the record_component_association_list of an extension_aggregate, the only components *needed* are those of the composite value defined by the aggregate that are not inherited from the type of the ancestor_part, ...
> 7 For the evaluation of an extension_aggregate, the record_component_association_list is evaluated. If the ancestor_part is an expression, it is also evaluated; if the ancestor_part is a subtype_mark, the components of the value of the aggregate not given by the record_component_association_list are initialized by default as for an object of the ancestor type. ...
> 9 If all components of the value of the extension_aggregate are determined by the ancestor_part, then the record_component_association_list is required to be simply **null record**.
> 10 If the ancestor_part is a subtype_mark, then its type can be abstract. ...

Here are two examples based on expressions from the case study:

```
(Engine_Event'(Create) with Left);
  -- Extend Engine_Event to create Aux_Engine_Event aggregate
(Engine_Event'(Create) with null record);
  -- Extend Engine_Event to create Main_Engine_Event aggregate
```

Even though Main_Engine_Event does not add any components during the extension of Engine_Event, it is still derived from Main_Engine_Event and an extension aggregate must be used with **null record**.

Extension aggregates built from subtype marks are intended to be used when the ancestor is abstract. For example, an aggregate for Engine can be written:

```
return ( Event with
           Fuel   => Random_Time.Random(G) mod 100,
           Oxygen => Random_Time.Random(G) mod 500);
```

Normally, the declaration of Event would contain a meaningful default value for Time.

View conversion and redispatching

Type conversion of tagged types is quite different from what you normally think of as type conversion. A new value is not created; instead, you get a new view of the original value which hides

components that are not part of the target type.

> 5 A type_conversion whose operand is the name of an object is called a *view conversion* if its target type is tagged, ...; other type_conversions are called *value conversions*. **§4.6**
>
> 42 The tag of the result is the tag of the operand. ...
>
> 55 If the target type is tagged, then an assignment to the view assigns to the corresponding part of the object denoted by the operand; ...

Since neither the tag §4.6(42) nor the value of the operand is changed by the conversion, you can always recover the original value and type.

During an assignment, both the source and the target objects retain their tags §5.2(15), and only the relevant components are copied §5.2(13), §4.6(56):

```
E: Engine_Event := ...;                  -- Tagged as Engine_Event
A: Aux_Engine_Event := ...;              -- Tagged as Aux_Engine_Event

E := Engine_Event(A);                    -- Side component ignored
Engine_Event(A) := E;                    -- Side component not assigned to
```

View conversions can be used for *redispatching*. Consider the following tagged type Parent where the derived type Derived inherits the primitive procedure Proc1 but overrides Proc2, and suppose that P_CL is a class-wide object containing a value of type Derived.

```
type Parent is tagged ...;
procedure Proc1(V: in Parent);
procedure Proc2(V: in Parent);
type Derived is new Parent with ...;
procedure Proc2(V: in Derived);

D: Derived := ...;
P_CL: Parent'Class := Parent'Class(D);

Proc1(P_CL);
```

When Proc1 is called, the value of class-wide type will be converted to the specific type Parent of the formal parameter V §4.6(23). However, the conversion is only a view conversion and V remains tagged as Derived. Within Proc1, the following statement will redispatch to the overridden Proc2:

```
Proc2(Parent'Class(V));
```

because the tag of the result is taken from the tag of the operand, namely Derived. This works because tagged types are passed by-reference §6.2(5) so that within Proc1 the tag of its actual parameter exists unchanged.

6.9 Objects of class-wide type*

Block statement

Before discussing objects of class-wide type, we need to make a short digression to study the block statement. A block is like a parameterless procedure written within a sequence of statements.

§5.6

```
block_statement ::=
   [block_statement_identifier:]
     [declare
       declarative_part]
     begin
       handled_sequence_of_statements
     end [block_identifier];
```

Blocks are used to declare objects that depend on a computation:

```
Get(N);
String_Block:
  declare
    S: String(1..N);
  begin
    ...
  end String_Block;
```

Another use for blocks is to retry a computation after an exception:

-- File: COUNTRY4

```
1   --
2   -- Read the manufacturer of a car and write the country
3   -- of origin of the car.
4   -- Exception handler in block.
5   --
6  with Ada.Text_IO; use Ada.Text_IO;
7  procedure Country4 is
8
9     ...
10
11 begin
12    loop
13       begin                                   -- Block begins here ...
14          Put("Enter the make of the car: ");
15          Get_Line(S, Last);
16          Car := Cars'Value(S(1..Last));
17          Put_Line(Cars'Image(Car) & is made in " &
18             Countries'Image(Car_to_Country(Car)));
```

```
19      exception
20        when Constraint_Error =>
21          Put_Line(S(1..Last) & " is not recognized");
22        when Ada.Text_IO.End_Error =>
23          Put_Line("Have a nice day!");
24          exit;
25      end;                                          -- ... and ends here.
26    end loop;
27  end Country4;
```

Since the block contains a handled sequence of statements, completion of the exception handler for Constraint_Error is a successful completion of the statement and the loop statement continues as usual. When end-of-file occurs (CTRL-D or CTRL-Z on a terminal), End_Error will be raised and the exit statement will cause the loop containing the block to be left.

Objects of class-wide type

Why didn't we use a procedure for Event_Queue.Get?

```
procedure Get(E: out Event'Class; Q: in Queue_Ptr);
```

The reason is that a class-wide type is indefinite, but an actual parameter is an object of some specific type and cannot have its type changed when the assignment to the **out** parameter is done:

```
EV_CL: Event'Class := Engine_Event'(100, 4102, 5335);
-- EV_CL contains a value of type Engine_Event ...
Get(EV_CL, Q);
-- ... but a Telemetry_Event might be returned,
-- ... raising Constraint_Error!
```

When a function returns an object, it must allocate (temporary) storage for the object. This object can then be used in an expression, for example as an actual parameter of a dispatching subprogram call.

As with any other indefinite type, one way to create a class-wide *object* that can store objects of different types within the class is to make it a formal parameter of a subprogram. The formal parameter is elaborated anew in each call and the constraint is taken from the actual parameter.

```
procedure Do_Simulation(EV_CL: in Event'Class) is
begin
  Root_Event.Simulate(EV_CL);
  Write_Event_to_Log(EV_CL);
end Do_Simulation;

while not Event_Queue.Empty(Q.all) loop
  Do_Simulation(Event_Queue.Get(Q));
end loop;
```

Alternatively, you can use a block statement:

```
   while not Event_Queue.Empty(Q.all) loop
     declare
       EV_CL: Event'Class := Event_Queue.Get(Q);
     begin
       Root_Event.Simulate(EV_CL);
       Write_Event_to_Log(EV_CL);
     end;
   end loop;
```

In each iteration of the loop, EV_CL is allocated and initialized with the object returned from Event_Queue.Get(Q). This object is discarded when leaving the block, just as local variables are discarded when leaving a subprogram. In this example, the object retrieved from the queue is used in two expressions (actual parameters of procedure calls), so the use of the local variable is essential. See Section 6.13 for another example of this technique.

It is important to understand the paradigms for programming with indefinite types. They enable you to encapsulate pointer-based implementations, so that client programmers can work directly with objects.

6.10 Abstract types*

We will now explain abstract types and subprograms in more detail. The specification of Root_Event is repeated here for convenience:

```
 1 package Root_Event is                                    -- File: ROCKET
 2   --
 3   -- Declaration of abstract event at root of event class.
 4   --
 5   type Event is abstract tagged private;
 6
 7   -- Declare (abstract) primitive operations of an Event.
 8   function Create return Event is abstract;
 9   procedure Simulate(E: in Event) is abstract;
10
11   -- Comparison of events is common to all events in the class.
12   function "<"(Left, Right: Event'Class) return Boolean;
13
14 private
15
16   subtype Simulation_Time is Integer range 0..10_000;
17   type Event is abstract tagged
18     record
19       Time: Simulation_Time;              -- Common component of all events
20     end record;
21
22 end Root_Event;
```

The abstract type Event serves as the ancestor of all the event types. Promoting one of the actual events to be the parent of all others would be arbitrary and inappropriate, so we declare an abstract event even though objects of this type are meaningless. In the simulation program, the abstract type Root_Event.Event is a convenient place to declare the common component Time. More commonly, an abstract type is declared as a null record.

The abstract primitive subprograms serve as ancestors of the real primitive subprograms to be declared upon derivation.

> 1 An *abstract type* is a tagged type intended for use as a parent type for type extensions, but which is not allowed to have objects of its own. An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body. **§3.9.3**

‡306 is an example of the last sentence of the above paragraph.

> 4 For a derived type, if the parent or ancestor type has an abstract primitive subprogram, ... then: **§3.9.3**
> 5 If the derived type is abstract or untagged, the inherited subprogram is abstract.
> 6 Otherwise, the subprogram shall be overridden with a nonabstract subprogram; ...
> 7 A call on an abstract subprogram shall be a dispatching call; nondispatching calls to an abstract subprogram are not allowed.
> 8 The type of an aggregate, or of an object created by an object_declaration or an allocator, ... shall not be abstract. The type of the target of an assignment, operation (see 5.2) shall not be abstract. The type of a component shall not be abstract. ...
> 13 A class-wide type is never abstract. Even if a class is rooted at an abstract type, the class-wide type for the class is not abstract, and an object of the class-wide type can be created; the tag of such an object will identify some nonabstract type in the class.

While a abstract *type* must be tagged §3.9.3(1), an abstract *subprogram* can be primitive for an untagged derived type §3.9.3(5); see Section 8.6. Such a subprogram is never callable and can be used to avoid exporting inherited operations.

6.11 Implementation of dispatching**

This book presents the Ada language as seen by a programmer and is not normally concerned with the implementation techniques used in the compiler and run-time system. Nevertheless, an outline of a possible implementation of dynamic dispatching will enable you to use the technique with the knowledge of the run-time overhead that is incurred.

Figure 6.6 shows a data structure that can be used in the implementation of dispatching for the rocket simulator.[6]

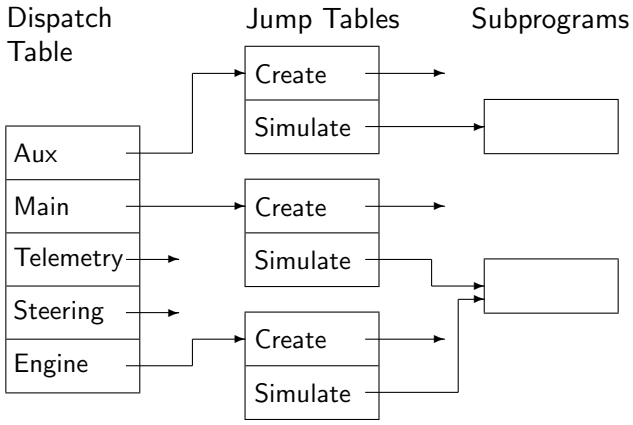

Figure 6.6: Implementation of dynamic dispatching

A *dispatch table* is created by the compiler and loaded at run-time. Tags are represented by offsets into the dispatch table. When a dispatching call is made, the offset is used to obtain the address of the *jump table* corresponding to the specific type of the controlling operands. The jump table contains a pointer to each primitive subprogram; an indirect call on this pointer calls the subprogram. Since Main_Engine_Event inherits Simulate from Engine_Event, a new procedure is not created for the derived type. Instead, the jump table pointer for Main_Engine.Simulate is directed at the procedure already declared for the parent type.

An implication of this implementation is that the run-time overhead is *small* and, more importantly, *fixed*. Two or three machine instructions will suffice for doing the double indirection, and for any given machine and compiler the overhead can be computed or measured. *All* dispatching calls will have exactly this overhead, so there is no uncertainty that would prevent the use of dispatching in real-time systems.

Once a tagged type or extension is declared, additional derived descendants can be declared without recompiling the package specification that declares the parent. Each additional derivation will add an entry to the dispatch table, a new jump table and code for any primitive subprograms overridden or added upon derivation. Existing tags (offsets) and jump tables are not affected.

Primitive subprogram must be declared in the package specification: since no more entries can be made to the jump table for this type, the table can be created during the compilation of the specification. This implementation is possible because derivation can only add primitive operations, not remove them, and any operation not overridden is inherited. Thus if a primitive operation Proc is defined for a tagged type T, then Proc will also be defined for any type in T'Class. Furthermore, the primitive operation can only be called with a controlling operand of the class-wide type T'Class or with the specific type T. This is checked at compile-time, so at run-time the dispatching call can be made without a run-time check.

[6]For lack of space, two jump tables and most subprograms have been omitted.

Even though the run-time overhead is small and fixed regardless of the complexity of the derivation class, deep derivation trees can make it difficult to understand and maintain a program. The reason is that for each specific type, you may have to examine the entire chain of ancestors to locate an inherited subprogram. A good development environment can help by automating the search.

6.12 Multiple controlling operands**

A primitive operation is allowed to have more than one controlling formal parameter. This is particularly useful for dispatching on binary operators:[7]

```
type T is tagged ...;
function "<"(Left: T; Right: T ) return Boolean;
```

Suppose now that a class of types T1, T2, ..., has been derived from T. Given two objects X and Y of some types within T'Class, what is the meaning of X < Y? Clearly, if both X and Y are of the same specific type, the compiler binds to the function declared for the type, either the inherited function or an overriding function. Equally clearly, if X and Y are of different specific types such as T3 and T5, no appropriate function exists. However, what happens if either X or Y, or both, are of the class-wide type T'Class?

8	A call on a dispatching operation shall not have both dynamically tagged and statically tagged controlling operands.	**§3.9.2**
16	... If there is more than one dynamically tagged controlling operand, a check is made that they all have the same tag. If this check fails, Constraint_Error is raised ...	

Given the declarations:

```
W:T3     := T3'(...);
X:T'Class:= T3'(...);
Y:T'Class:= T3'(...);
Z:T'Class:= T5'(...);
```

W < X is illegal by §3.9.2(8), X < Y dispatches to the function for T3 and X < Z raises Constraint_Error. The restrictions are intended to simplify the language implementation.

You can avoid the Constraint_Error promised by §3.9.2(16) by comparing the tags of the class-wide type §3.9(17–18):

```
with Ada.Tags; use Ada.Tags;

C1: T'Class := Get(Q);
C2: T'Class := Get(Q);
```

[7] This "<" operator is a primitive operation since its parameters are of the specific tagged type T, unlike the operator "<" in the case study, which had parameters of type Event'Class.

```
if C1'Tag = C2'Tag then
  if C1 < C2 then ...                    -- Dispatch!
  else ...
  end if;
else ...                                 -- Different types - do something else
end if;
```

Note that we are only comparing tags for equality, not asking if a value of class-wide type has a specific tag, so that the statement need not be modified if additional derivations are done.

There is a special rule concerning the inheritance of the equality operator; see Quiz 12.

6.13 Dispatching on the function result**

A primitive function such as Create is said to have a controlling result if it returns a tagged type. So far, we have used expanded names and qualified expressions to indicate to the compiler which version of Create to call; no dispatching is needed:

```
Event_Queue.Put(Engine.Aux_Engine_Event'(Engine.Create), Q.all);
Event_Queue.Put(Telemetry.Create, Q.all);
```

However, a call of a primitive function can also be a controlling *operand* of a dispatching subprogram call. Such a function call is termed *tag indeterminate* §3.9.2(3); a tag-indeterminate operand is legal only if there is sufficient context to determine how to bind it. For example, in the call Simulate(Create), not enough context is supplied to dispatch Create.

Suppose we wish to modify the rocket simulation so that whenever an event is removed from the queue, a *new* event of the *same* type is inserted into the queue. This can be done by dispatching Create using the technique that we now describe. First, declare a new primitive subprogram Another ‡11 with two controlling parameters:

```
1   --                                                    -- File: ROCKETF
2   -- Discrete event simulation of a rocket.
3   -- Demonstrates dispatching on function result.
4   --
5  package Root_Event is
6
7    type Event is abstract tagged private;
8
9    function Create return Event is abstract;
10   procedure Simulate(E: in Event) is abstract;
11   function Another(Original: Event; Copy: Event) return Event'Class;
12
13   ...
14
15 end Root_Event;
```

Next modify the main loop of the simulation to call Another ‡26 with two parameters—one dynamically tagged of class-wide type and the other a tag-indeterminate function call:

```
16 with Event_Queue;
17 with Root_Event.Engine, Root_Event.Telemetry, Root_Event.Steering;
18 use Root_Event;
19 procedure RocketF is
20
21    ...
22
23    loop
24       declare
25          First: Event'Class := Event_Queue.Get(Q);
26          Second: Event'Class := Another(First, Root_Event.Create);
27       begin
28          Event_Queue.Put(Second, Q.all);
29          Root_Event.Simulate(First);
30       end;
31    end loop;
32 end RocketF;
```

There is now sufficient context to disambiguate the call to Create: since all controlling operands must have the same tag, Create is dynamically dispatched to the version appropriate for the specific type contained in First!

The function Another is just a framework for this dispatching:

```
33 package body Root_Event is
34
35    function Another(Original: Event; Copy: Event) return Event'Class is
36    begin
37       return Event'Class(Copy);
38    end Another;
39
40    -- Implement class wide operation.
41    function "<"(Left, Right: Event'Class) return Boolean is
42    begin
43       return Left.Time < Right.Time;
44    end "<";
45
46 end Root_Event;
```

A tag-indeterminate operand is statically bound if the other parameters are all statically bound:

```
T: Telemetry_Event;
E: Event'Class := Another(T, Create);
  -- Create is statically bound to Telemetry.Create
```

Dispatching on a function call can also occur in the default expression for a controlling formal parameter §3.9.2(11), in an assignment statement §5.2(9) and in enclosing tag-indeterminate calls §3.9.2(6). A program **FUNC.ada** demonstrating these features is included on the CD-ROM.

7 Generics

In Chapter 4 we developed a priority queue abstract data type and showed how to change the implementation of the queue from an array to a tree without changing the client interface supplied by the package specification. Suppose now that we want a priority queue for another type such as floating point numbers. We could simply copy the source code of the existing package and replace all occurrences of Integer by Float. Obviously, this is tedious and error-prone. The modification of the source code could be automated by using a macro processor, either an external program or a preprocessor built into the language implementation. However, the use of a macro processor is also error-prone because the substitutions are done on pure text regardless of syntactic or semantic implications. For example, replacement of Rec by Rec2 will result in the replacement of the reserved word **record** by Rec2ord. While this problem can be resolved by taking delimiters into account, macro processors are always problematical, particularly if you are making several substitutions in succession.

Another solution to the generalization of abstract data types is the use of heterogenous types that we discussed in the previous chapter. In fact, some languages for object-oriented programming use this approach exclusively by defining every type to be an extension of a root type Object. Since every type is derived from Object, a data structure whose elements are of type Object can be used to store elements of any type. There are two drawbacks to this approach: (a) it requires additional overhead because reference semantics is used, and (b) a potentially dangerous type conversion must be done on an item retrieved from the data structure. While Ada allows run-time type checking within the narrow confines of a class of closely-related types, a data structure of unrelated types would not be compatible with strong type checking.

The Ada solution to parameterizing data structures is *generic* units, which are templates that can be used to create instances of a unit at *compile-time*. Since type checking is done at compile-time, the use of generics entails no run-time overhead.[1] The creation of an instance—called *instantiation*—is done by the compiler, which enforces syntactic and semantic rules.

7.1 Generic declaration and instantiation

A generic declaration declares a generic package or subprogram. Syntactically, a generic specification is an ordinary specification preceded by a *generic formal part*:

[1] There are a few exceptions to this statement. For example, since a generic can be instantiated anywhere, accessibility checks (Section 9.4) must be done at run-time.

```
2  generic_declaration ::=
     generic_subprogram_declaration |
     generic_package_declaration
3  generic_subprogram_declaration ::=
     generic_formal_part  subprogram_specification;
4  generic_package_declaration ::=
     generic_formal_part  package_specification;
5 generic_formal_part ::=
     generic
       {generic_formal_parameter_declaration | use_clause}
```

Case study: generic priority queue

Here is a generic version of the package Priority_Queue. The details of the generic formal part are left to the next section. The package is unchanged except for the substitution of the generic formal parameter Item for Integer.

```
1   --                                                    -- File: PQGEN
2   -- Priority queue abstract data type implemented as a tree.
3   -- Queue is limited private; representation of nodes is in body.
4   -- Queue element is generic.
5   --
6  generic
7    type Item is ...
8    with function "<"(Left, Right: Item) return Boolean is <>;
9  package Priority_Queue is
10
11   type Queue(Size: Positive) is limited private;
12
13   function Empty(Q: in Queue) return Boolean;
14   procedure Put(I: in Item; Q: in out Queue);
15   procedure Get(I: out Item; Q: in out Queue);
16
17   Overflow, Underflow: exception;
18
19 private
20   ...
21 end Priority_Queue;
22
```

```
23 package body Priority_Queue is
24
25    type Node is                                  -- Completion of type declaration
26       record
27          Data: Item;
28          Left, Right: Link;
29       end record;
30
31    ...
32
33    procedure Put(I: in Item; Node_Ptr: in out Link) is
34    -- Recursive procedure to insert in queue
35    begin
36       if Node_Ptr = null then
37          Node_Ptr := new Node'(I, null, null);
38       elsif I < Node_Ptr.Data then
39          Put(I, Node_Ptr.Left);
40       else
41          Put(I, Node_Ptr.Right);
42       end if;
43    end Put;
44    ...
45 end Priority_Queue;
```

> 1 The body of a generic unit (a *generic body*) is a template for the instance bodies. The syntax of a generic body is identical to that of a nongeneric body. **§12.2**

> 12 A generic_instantiation declares an instance; it is equivalent to the instance declaration (a package_declaration or subprogram_declaration) immediately followed by the instance body, both at the place of the instantiation. **§12.3**
>
> 13 The instance is a copy of the text of the template. Each use of a formal parameter becomes (in the copy) a use of the actual, as explained below. An instance of a generic package is a package, that of a generic procedure is a procedure, and that of a generic function is a function.

The following program instantiates Priority_Queue twice, once with Integer ‡51 and once with Float ‡52. Note that, like any other library unit, you must 'with' the generic unit ‡46 in order to access it. Once the instances have been created, they are normal packages and their resources accessed using expanded names ‡54, 55, 71, 77, . . . , or a 'use' clause. Procedure Put for floating point output ‡64–66 is obtained by *renaming* the library procedure so as to change the default parameters (see Section 12.5).

```
46 with Priority_Queue;
47 with Ada.Text_IO; with Ada.Integer_Text_IO; with Ada.Float_Text_IO;
48 use Ada; use Text_IO;
49 procedure PQGEN is
50
51    package Integer_Queue is new Priority_Queue(Item => Integer);
52    package Float_Queue is new Priority_Queue(Item => Float);
53
54    QI: Integer_Queue.Queue(10);                    -- Create queue of size 10
55    QF: Float_Queue.Queue(10);                      -- Create queue of size 10
56
57    I: Integer;                                     -- Element of the queue
58    F: Float;                                       -- Element of the queue
59    Integer_Test_Data: array(Positive range <>) of Integer :=
60       (10, 5, 0, 25, 15, 30, 15, 20, -6, 40);
61    Float_Test_Data: array(Positive range <>) of Float :=
62       (10.0, 5.0, 0.0, 25.0, 15.0, 30.0, 15.0, 20.0, -6.0, 40.0);
63
64    procedure Put(F: in Float;
65       Fore: in Field:=3; Aft: in Field:=1; Exp: in Field:=0)
66          renames Float_Text_IO.Put;
67
68 begin
69    for N in Integer_Test_Data'Range loop
70       Integer_Text_IO.Put(Integer_Test_Data(N), Width => 5);
71       Integer_Queue.Put(Integer_Test_Data(N), QI);
72    end loop;
73    New_Line;
74
75    for N in Float_Test_Data'Range loop
76       Put(Float_Test_Data(N));
77       Float_Queue.Put(Float_Test_Data(N), QF);
78    end loop;
79    New_Line;
80
81    while not Integer_Queue.Empty(QI) loop
82       Integer_Queue.Get(I, QI);
83       Integer_Text_IO.Put(I, Width => 5);
84    end loop;
85    New_Line;
86
87    while not Float_Queue.Empty(QF) loop
88       Float_Queue.Get(F, QF);
89       Put(F);
90    end loop;
```

```
91    New_Line;
92
93  exception
94    when Integer_Queue.Underflow | Float_Queue.Underflow =>
95      Put_Line("Underflow from queue");
96    when Integer_Queue.Overflow | Float_Queue.Overflow =>
97      Put_Line("Overflow from queue");
98  end PQGEN;
```

7.2 The contract model

Suppose we try to instantiate the generic package Priority_Queue with a record:

```
type Point is
  record
    X: Float;
    Y: Float;
  end record;

package Point_Queue is new Priority_Queue(Item => Point);
```

Recall that the package body of Point_Queue is a copy of the text of the template; this copy must now be compiled. When the compiler attempts to compile the expression I < Node_Ptr.Data in ‡38, an error will occur since the operator "<" is not defined on the type Point. You need to ensure that Priority_Queue can only be instantiated for types for which the operator is defined and visible.

We are able to diagnose the problem quickly because the text of the generic package is well known to us since it has appeared so often in this book. If you had obtained the package from another employee in your company, you would probably have to seek help in understanding the package. Furthermore, it would be nearly impossible to create a a proprietary generic package if an instantiation by the customer could cause arbitrary compilation errors in the body.

The *contract model* of generics in Ada has been designed to minimize, if not eliminate, these problems. The idea is that the generic formal parameters contain sufficient information so that:[2]

- The generic unit itself can be compiled. Since the generic unit is just a template for which no code is generated, it would be more exact to say that the unit is checked for legality.
- An instantiation may fail if the generic *actual* parameters do not match the *formal* parameters, but no other compilation errors will occur as a result of the instantiation.

Thus the generic parameters form a contract between the programmer writing the generic unit and the programmers using the unit.

The contract model is illustrated in Figure 7.1. The generic *formal* parameter declaration (left side of the figure) specifies a class of types. The generic unit is allowed to use *at most* the operations

[2]The explanation of the contract model here is only approximate. The precise explanation is given in Section 7.11.

common to all types in the class. The generic *actual* parameter must be a type which supplies *at least* those operations.

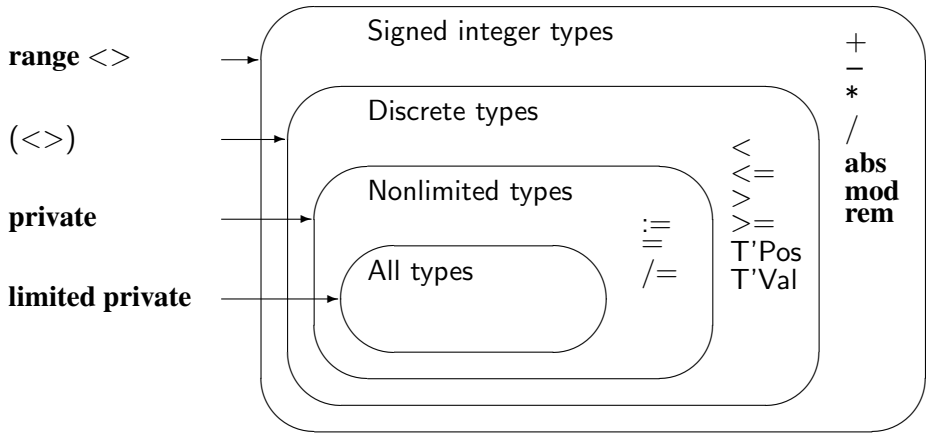

Figure 7.1: Generic types

For example, if the generic formal parameter is **private**, the generic unit is allowed to create objects of the type, to perform assignments and to use the equality and inequality operators. In an instantiation, the generic actual parameter can be any nonlimited type. The actual parameter may, of course, be the type Integer, which also supplies operations like addition, but the generic unit is not allowed to use such operations, only those operations that are common to all types in the class 'nonlimited types'.

Similarly, if the formal parameter is (<>),[3] indicating the class of all discrete types, the generic unit would be allowed to write:

```
for N in Item'Range loop
```

because any discrete type can be used as the type of a loop parameter. We could instantiate with Character or Cars or even Integer, but not with Point.

Note that classes of types do not form always a simple inclusion hierarchy as implied by Figure 7.1; for example, floating point operations are neither a superset nor a subset of integer operations, so they have distinct formal parameter declarations, and a generic floating point formal parameter cannot be associated with an actual parameter of type Integer.

[3]Do not try to read too much into the syntax of generic formal parameters. Familiar reserved words and symbols are reused in a manner that hints at the semantics, but you must learn the exact meaning of each construct.

1 A formal scalar type is one defined by any of the formal_type_definitions in this **§12.5.2** subclause. The class determined for a formal scalar type is discrete, signed integer, modular, floating point, ordinary fixed point, or decimal.

2 formal_discrete_type_definition ::= (<>)
3 formal_signed_integer_type_definition ::= **range** <>
4 formal_modular_type_definition ::= **mod** <>
5 formal_floating_point_definition ::= **digits** <>
6 formal_ordinary_fixed_point_definition ::= **delta** <>
7 formal_decimal_fixed_point_definition ::= **delta** <> **digits** <>

Let us return to the problem of the priority queue. One possibility would be declare the generic formal parameter to be of class discrete:

```
generic
  type Item is (<>);
package Priority_Queue is ...
```

This would allow us to instantiate the package for any discrete type such as Character and Integer. The package would compile correctly, because "<" is predefined for every discrete type. However, it would not allow us to create a priority queue of floating point values, because floating point types are not discrete. Nor would it allow us to create a priority queue of objects of type Point, even assuming that "<" were defined (though not predefined) for the type.

7.3 Generic formal subprograms

A generic formal subprogram parameter declares a subprogram that can be invoked within the generic unit. An instantiation must supply a subprogram which is the one actually called by the instance. A contract model is enforced upon instantiation: the subprogram given as an actual parameter must conform to the formal parameter declaration. A flexible generic priority queue package is obtained by declaring (a) the type Item as private so that a queue can be instantiated for any nonlimited type, and (b) a *generic formal function* for the less-than operator:[4]

```
generic
  type Item is private;
  with function "<"(Left, Right: Item) return Boolean is <>;
package Priority_Queue is ...
```

The instantiation must supply an actual function that is mode-conformant[5] with the formal parameter. The following function uses *short circuit control forms* **and then** and **or else** §4.5.1 for efficient evaluation of the Boolean expression.

[4]The reserved word **with** is reused here a syntactic marker to indicate that the subprogram specification is that of a formal parameter, not of the generic subprogram.

[5]See the Glossary for the definition of this term.

```
function Less_Than(Left, Right: Point) return Boolean is
begin
  return (Left.X < Right.X) or else
    ((Left.X = Right.X) and then (Left.Y < Right.Y));
end Less_Than;

package Point_Queue is
    new Priority_Queue(Item => Point, "<" => Less_Than);
```

Of course, a predefined operator will always be visible and can be used as an actual parameter:

```
package Float_Queue is new Priority_Queue(Float, "<");
```

§12.6

```
2  formal_subprogram_declaration ::=
     with subprogram_specification [is subprogram_default];
3  subprogram_default ::= default_name | <>
4  default_name ::= name
```

10 If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal.

We have declared the generic formal function of the priority queue generic package with a box <>, so that a visible conforming function will be used by default:

```
package Float_Queue is new Priority_Queue(Float);
```

Similarly, if the comparison function for type Point had been declared by overloading the "<" operator, we would not have had to give it explicitly as an actual parameter:

```
function "<"(Left, Right: Point) return Boolean is ...
package Point_Queue is new Priority_Queue(Item => Point);
```

7.4 Dependence of generic formal parameters

The generic formal function "<" has parameters which are of the type of the previous generic formal type Item. Such a dependence of one formal parameter on another is often used in the construction of generic units, in particular when the formal parameter is an array type §12.5.3 or an access type §12.5.4.

Case study: generic sort subprogram

We demonstrate dependence of generic formal parameters in a generic procedure for sorting, where the generic array type Vector depends on previous formal parameters for both its Index and its component type Item. The declaration of the generic procedure is:

```
1   --                                                    -- File: SORT
2   -- Generic procedure for selection sort.
3   --
4  generic
5    type Index is (<>);
6    type Item is private;
7    type Vector is array(Index range <>) of Item;
8    with function "<"(Left, Right: Item) return Boolean is <>;
9  procedure SelectionSort(A: in out Vector);
```

The generic subprogram body is the completion of the generic declaration §12.2(3). (For a subprogram that is not generic, the subprogram body can also serve as the subprogram declaration §6.3(4).) Note the computation of the loop bounds in ‡14: Index is of class (<>) and may be instantiated with any discrete type, not necessarily with an integer type. So we cannot write A'Last-1; instead the attribute Index'Pred, which is defined for any discrete type is used.

```
10 procedure SelectionSort(A: in out Vector) is
11   Min: Index;
12   Temp: Item;
13 begin
14   for I in A'First .. Index'Pred(A'Last) loop
15       Min := I;
16       for J in I .. A'Last loop
17         if A(J) < A(Min) then Min := J; end if;
18       end loop;
19       Temp := A(I); A(I) := A(Min); A(Min) := Temp;
20   end loop;
21 end SelectionSort;
```

The procedure can be instantiated ‡40–41 with any discrete type for its index and any nonlimited type for its component, in this case Character and Point:

```
22 with SelectionSort;
23 with Ada.Text_IO; use Ada.Text_IO;
24 with Ada.Float_Text_IO; use Ada.Float_Text_IO;
25 procedure Sort is
26
27   type Point is
28     record
29       X, Y: Float;
30     end record;
31
32   type Point_Vector is array(Character range <>) of Point;
33
```

```
34    function "<"(Left, Right: Point) return Boolean is
35    begin
36       return (Left.X < Right.X) or else
37              ((Left.X = Right.X) and then (Left.Y < Right.Y));
38    end "<";
39
40    procedure Point_Sort is new SelectionSort(
41       Character, Point, Point_Vector);
42
43    A: Point_Vector :=
44       ((10.0,1.0), (4.0,2.0), (5.0,3.4), (10.0,0.5));
45 begin
46    Point_Sort(A);
47    for I in A'Range loop
48       Put(A(I).X,5,2,0);
49       Put(A(I).Y,5,2,0);
50       New_Line;
51    end loop;
52 end Sort;
```

§12.5.3 gives the rules for matching actual array types with formal array types; in particular, both types must be either unconstrained (as in the example) or constrained. Formal access types whose designated type is a previous formal parameter can be used to build generic units that work on linked structures; see §12.5.4.

7.5 Generic formal tagged private types*

There are two ways that tagged types can be used as generic formal parameters. We begin with generic tagged private types. Generic derived types are discussed in the next section.

A generic formal private type can be declared as tagged:

```
generic
  type Item is tagged private;
```

The class of types that may be used as actual parameters is the class of all nonlimited tagged types. (As with ordinary formal private types, **limited** may be specified in the formal parameter §12.5.1(17), in which case any tagged type may be the actual parameter.) What can you do with *any* tagged type? Obviously, you cannot call any of its primitive operations (other than equality), as these will be different for different tagged types. But any tagged type can be extended!

Case study: mixin inheritance

In the following program,[6] an arbitrary tagged type Item ‡5 is extended to define a new type Displayed_Item ‡8 with an additional component ‡15 and primitive subprograms ‡9–11.

[6]The packages for the simulation are unchanged and omitted here.

```
1   --                                                    -- File: MIXIN1
2   -- 'Mixin' inheritance using tagged generic parameter.
3   --
4  generic
5    type Item is tagged private;
6    with function Init return Item;
7  package Display is
8    type Displayed_Item is new Item with private;
9    procedure Show(D: in Displayed_Item);
10   procedure Set_Size(D: in out Displayed_Item; N: Natural);
11   function Create return Displayed_Item;
12 private
13   type Displayed_Item is new Item with
14     record
15       Size: Integer;
16     end record;
17 end Display;
```

The new subprograms are implemented in the package body. The generic function parameter Init is used in the extension aggregate to create a value of type Displayed_Item ‡34.

```
18 with Ada.Text_IO;
19 package body Display is
20
21   procedure Show(D: in Displayed_Item) is
22   begin
23     Ada.Text_IO.Put_Line("Size = " & Integer'Image(D.Size));
24   end Show;
25
26   procedure Set_Size(D: in out Displayed_Item; N: Natural) is
27   begin
28     D.Size := N;
29   end Set_Size;
30
31   function Create return Displayed_Item is
32     Initial_Size: constant Natural := 300;
33   begin
34     return (Init with Size => Initial_Size);
35   end Create;
36 end Display;
```

The instantiation Displayed_Main_Engine of Display ‡39–41 creates a type that is like Main_-Engine_Event, but extended.

```
37 with Display;
38 with Root_Event.Engine;
39 package Displayed_Main_Engine is
40    new Display(Root_Event.Engine.Main_Engine_Event,
41                Root_Event.Engine.Create);
```

The main subprogram 'with's the instantiated package ‡44. Both the subprogram Simulate ‡67 which is primitive for Event (the generic actual type) and the new subprograms Create ‡53, Show ‡69 and Set_Size ‡55, can be used on objects of type Displayed_Main_Engine.Displayed_Item.

```
42 with Priority_Queue;
43 with Root_Event.Engine, Root_Event.Telemetry, Root_Event.Steering;
44 with Displayed_Main_Engine;
45 use Root_Event;
46 procedure Mixin1 is
47    package Event_Queue is new Priority_Queue(Event'Class);
48    Q: aliased Event_Queue.Queue;
49 begin
50    for I in 1..15 loop
51       declare
52          M: Displayed_Main_Engine.Displayed_Item :=
53               Displayed_Main_Engine.Create;
54       begin
55          Displayed_Main_Engine.Set_Size(M, 500+I*10);
56          Event_Queue.Put(M, Q);
57       end;
58       Event_Queue.Put(Engine.Aux_Engine_Event'(Engine.Create), Q);
59       Event_Queue.Put(Telemetry.Create, Q);
60       Event_Queue.Put(Steering.Create, Q);
61    end loop;
62
63    while not Event_Queue.Empty(Q) loop
64       declare
65          EC: Event'Class := Event_Queue.Get(Q'Access);
66       begin
67          Root_Event.Simulate(EC);
68          if EC in Displayed_Main_Engine.Displayed_Item then
69             Displayed_Main_Engine.Show(
70                Displayed_Main_Engine.Displayed_Item(EC));
71          end if;
72       end;
73    end loop;
74 end Mixin1;
```

This programming paradigm is called *mixin inheritance*. We have no intention of creating objects of type Display.Displayed_Item ‡8; in any case, it is a just a template. Our intention is to 'mix

in' the properties of this type with an existing parent type (here Main_Engine_Event) to derive a new type (here Displayed_Main_Engine.Displayed_Item).

Some languages for object-oriented programming use *multiple inheritance* (MI) in these situations: a derived type can have more than one parent. MI presents technical difficulties in both the language definition and the implementation, so the designers of Ada 95 chose to omit MI from the language. Instead, programming paradigms using single inheritance together with other constructs like generics are used where other languages might use MI. See Section 4.6 of the *Rationale* for additional examples.

7.6 Generic formal derived types*

The second way to use a tagged type as a formal type is to use a generic formal derived type:

```
with Root_Event;
generic
   type Item is new Root_Event.Event with private;
```

The actual parameter can be Event or any type descended from it. According to the contract, within the generic unit any primitive operation of Event can be used, since any descendant of Event is certain to supply that operation, either by inheritance or by overriding. Additional operations that were declared for the descendants upon extension are of course not available, since they are different for each type (Figure 7.2). Create and Simulate are supplied by all descendants, while the other subprograms Emergency_Status and Computer_Heading (assumed to have been added during the extensions) are not available within the generic unit.

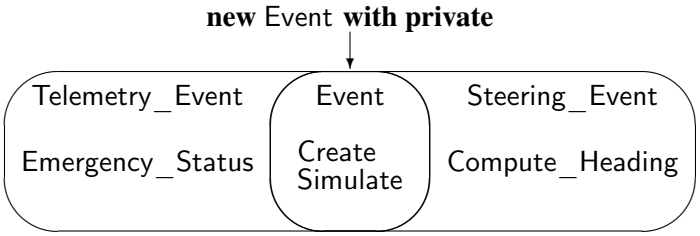

Figure 7.2: Generic tagged types

We now give another version of the mixin program where the generic formal parameter is derived from Event, rather than from an unspecified tagged type. The primitive operations of the tagged type are copied into the instance §12.3(16), so Create can be used directly in the generic package body ‡34, and need not be supplied as an additional generic parameter.

```
1   --                                                    -- File: MIXIN2
2   --  'Mixin' inheritance using generic formal derived type
3   --
4  with Root_Event;
5  generic
6    type Item is new Root_Event.Event with private;
7  package Display is
8    type Displayed_Item is new Item with private;
9    procedure Show(D: in Displayed_Item);
10   procedure Set_Size(D: in out Displayed_Item; N: Natural);
11   function Create return Displayed_Item;
12 private
13   type Displayed_Item is new Item with
14     record
15       Size: Integer;
16     end record;
17 end Display;
18
19 with Ada.Text_IO;
20 package body Display is
21   procedure Show(D: in Displayed_Item) is
22   begin
23     Ada.Text_IO.Put_Line("Size = " & Integer'Image(D.Size));
24   end Show;
25
26   procedure Set_Size(D: in out Displayed_Item; N: Natural) is
27   begin
28     D.Size := N;
29   end Set_Size;
30
31   function Create return Displayed_Item is
32     Initial_Size: constant Natural := 300;
33   begin
34     return Displayed_Item'(Item'(Create) with Size => Initial_Size);
35   end Create;
36 end Display;
37
38 with Display;
39 with Root_Event.Engine;
40 package Displayed_Main_Engine is
41   new Display(Root_Event.Engine.Main_Engine_Event);
```

A generic formal derived type is more specialized than a generic formal private type, but it does give direct access to the primitive operations of the actual type.

Mixing at the root

Rather than mixing the display into each event, it might be better to mix it once into the root event; then, all events would automatically be Displayed_Items. The generic package Display is unchanged from Section 7.5. Package Root_Event is modified so that Event ‡2 is no longer abstract. The reason is that Create ‡25 must be given as a generic actual parameter and cannot be abstract §3.9.3(11).

```
1  package Root_Event is                                    -- File: MIXIN3
2     type Event is tagged private;
3     function Create return Event;
4     procedure Simulate(E: in Event);
5     function "<"(Left, Right: Event'Class) return Boolean;
6  private
7     ...
8  end Root_Event;
```

Within the body, we do the processing of the Time component ‡16, since it will no longer be visible after the mixin.

```
9  with Root_Event.Random_Time;
10 with Ada.Text_IO; use Ada.Text_IO;
11 package body Root_Event is
12    G: Random_Time.Generator;
13
14    function Create return Event is
15    begin
16       return (Time => Random_Time.Random(G));
17    end Create;
18
19    ...
20
21 end Root_Event;
```

Next comes the instantiation to perform the mixin:

```
22 with Display;
23 with Root_Event;
24 package Displayed_Event is
25    new Display(Root_Event.Event, Root_Event.Create);
```

The overridden operations in the extensions for the various events can call the operations from the mixin ‡34,38.

```
26 with Ada.Text_IO; use Ada.Text_IO;
27 with Root_Event.Random_Time;
28 package body Root_Event.Engine is
29
30    G: Random_Time.Generator;
31
32    function Create return Engine_Event is
33       E: Engine_Event :=
34             ( Displayed_Event.Create with
35               Fuel => Random_Time.Random(G) mod 100,
36               Oxygen => Random_Time.Random(G) mod 500);
37    begin
38       Displayed_Event.Set_Size(Displayed_Event.Displayed_Item(E), 500);
39       return E;
40    end Create;
41
42    ...
43
44 end Root_Event.Engine;
```

Finally, within the main subprogram, primitive operations of both 'parents' may be called. Show ‡70 is a primitive operation added to the extension Displayed_Item in the generic package specification, and Simulate ‡71 is an implicit declaration in the instance Displayed_Event §12.3(16–17) of a primitive operation *inherited* from the generic actual parameter Root_Event.Event.

```
45 with Priority_Queue;
46 with Root_Event.Engine, Root_Event.Telemetry, Root_Event.Steering;
47 with Displayed_Event;
48 procedure Mixin3 is
49    package Event_Queue is
50       new Priority_Queue(Root_Event.Event'Class, Root_Event."<");
51    Q: aliased Event_Queue.Queue;
52 begin
53    for I in 1..15 loop
54       declare
55          use Root_Event;
56       begin
57          Event_Queue.Put(Engine.Main_Engine_Event'(Engine.Create), Q);
58          Event_Queue.Put(Engine.Aux_Engine_Event'(Engine.Create), Q);
59          Event_Queue.Put(Telemetry.Create, Q);
60          Event_Queue.Put(Steering.Create, Q);
61       end;
62    end loop;
63
```

```
64    while not Event_Queue.Empty(Q) loop
65       declare
66          use Displayed_Event;
67          DIC: Displayed_Item'Class :=
68                Displayed_Item'Class(Event_Queue.Get(Q'Access));
69       begin
70          Simulate(DIC);
71          Show(DIC);
72       end;
73    end loop;
74 end Mixin3;
```

There is a special rule that requires that any extension to a generic formal tagged type be done in the generic package specification (visible or private part), not in the body (Section 7.11).

7.7 Generic formal objects*

1 A generic formal object can be used to pass a value or variable to a generic unit. **§12.4**

2 formal_object_declaration ::=
    defining_identifier_list : mode subtype_mark
      [:= default_expression];

7 For a generic formal object of mode **in**, the actual shall be an expression. For a generic formal object of mode **in out**, the actual shall be a name that denotes a variable for which renaming is allowed (see 8.5.1).

10 In an instance, a formal_object_declaration of mode **in** declares a new stand-alone constant object whose initialization expression is the actual, whereas a formal_object_declaration of mode **in out** declares a view whose properties are identical to those of the actual.

A formal object of mode **in** is typically used to configure a generic unit. A formal object of mode **in out** can be used to give the generic unit access to a variable in the unit enclosing the instantiation.

```
generic
   Size: in Integer := 100;
   Current: in out State;                    -- State must be visible here
package P is
   subtype Name is String(1..Size);
end P;
```

Note the difference between the use of Size as a generic formal object and the use of Size as a discriminant of the type Priority_Queue. All objects of type Name from a single instantiation will have the same Size, while we can declare Queue's of different sizes from a single instantiation of Priority_Queue.

7.8 Indefinite and abstract parameters**

Can we instantiate the priority queue package with type Event'Class?

```
package Event_Queue is new Priority_Queue(Event'Class);
```

The answer is no. In ‡37 of the generic priority queue package (Section 7.1), an aggregate is used to allocate a new node; an aggregate is an 'anonymous object' §4.3 which stores the components. But you cannot create a record object with components of indefinite type such as a class-wide type. To preserve the contract model, it is illegal to use an indefinite type as the actual parameter for a definite generic formal private parameter §12.5.1(6).

You can write a generic priority queue package that will accept Event'Class items by changing the body to use indirect allocation as we did in Chapter 6, and by declaring the formal parameter to have unknown discriminants:

```
1 generic                                                    -- File: ROCKETQ
2   type Item(<>) is private;                  -- Object of type Item never declared
3   with function "<"(Left, Right: Item) return Boolean is <>;
4 package Heterogeneous_Priority_Queue is ...
```

The contract is now valid: you cannot declare objects of type Item within the generic package, so there is no reason to forbid instantiation with indefinite types.

Formal tagged types can also be declared as abstract.

§12.5.1

```
2  formal_private_type_definition ::=
     [[abstract] tagged] [limited] private
3  formal_derived_type_definition ::=
     [abstract] new subtype_mark [with private]
```

18 The presence of the reserved word **abstract** determines whether the actual type may be abstract.

Note the difference between an abstract type and an indefinite type: an object can never be declared for an abstract type, but an object can be declared for an indefinite type provided that an initial value is given. This might be done by passing the generic unit an initializing function as a formal parameter. If the formal is abstract, since you cannot declare objects within the generic unit, it might as well have unknown discriminants, allowing indefinite actual parameters.

7.9 Formal package parameters**

> 1 Formal packages can be used to pass packages to a generic unit. The formal_package_declaration declares that the formal package is an instance of a given generic package. Upon instantiation, the actual package has to be an instance of that generic package. **§12.7**
>
> 2 formal_package_declaration ::=
> **with** package defining_identifier **is**
> **new** *generic_package*_name formal_package_actual_part;
> 3 formal_package_actual_part ::= (<>) | [generic_actual_part]
> 4 The *generic_package*_name shall denote a generic package (the *template* for the formal package); the formal package is an instance of the template.
> 5 The actual shall be an instance of the template. If the formal_package_actual_part is (<>), then the actual may be any instance of the template; otherwise, each actual parameter of the actual instance shall match the corresponding actual parameter of the formal package ...

Composing abstractions is an important programming technique. With generic package parameters, you can supply one abstraction—a generic unit—with a second abstraction, without listing all the types and operations of the second abstraction as separate parameters. Section 10.8 contains an example of the direct composition of generics. Here we demonstrate the use of an empty generic package called a *signature* to specify an abstraction needed by a generic unit.

Case study: generic simulation

We generalize the rocket simulation by declaring a generic simulation package that can be instantiated for any event type and for any priority queue implementation.

We start by declaring a Root_Event package. Note that the event type is no longer abstract, because generic events will now be put on the queue, not just events derived from the root event. The package body (omitted) contains dummy bodies for the primitive operations.

```
1   --                                                    -- File: ROCKETG
2   -- Root event
3   --
4  package Root_Event is
5    type Event is tagged private;
6    function Create return Event;
7    procedure Simulate(E: in Event);
8    function "<"(Left, Right: Event'Class) return Boolean;
9  private
10   subtype Simulation_Time is Integer range 0..10_000;
```

```
11    type Event is tagged
12      record
13        Time: Simulation_Time;
14      end record;
15  end Root_Event;
```

The next stage is to declare a signature for a generic event priority queue. The meaning of the signature is that an event priority queue is *any* package that supplies the type and subprograms declared as generic formal parameters.

```
16    --
17    -- Generic event priority queue signature
18    --
19  with Root_Event;
20  generic
21    type Queue(Size: Positive) is limited private;
22    with function Empty(Q: access Queue) return Boolean is <>;
23    with procedure Put(E: in Root_Event.Event'Class; Q: access Queue) is <>;
24    with function Get(Q: access Queue) return Root_Event.Event'Class is <>;
25  package Generic_Event_Priority_Queue is
26  end Generic_Event_Priority_Queue;
```

The signature is used as a generic formal package parameter ‡33 for the package Generic_Simulator; the generic body uses subprograms Get ‡50, Empty ‡49 and Put ‡45 from the specification of the formal parameter. In terms of the contract model, the formal parameter specifies the types and subprograms from the package available for use within the generic unit; the actual parameter must be a package that provides *at least* those types and subprograms.

```
27    --
28    -- Generic simulator
29    --
30  with Root_Event;
31  with Generic_Event_Priority_Queue;
32  generic
33    with package Event_Queue is new Generic_Event_Priority_Queue(<>);
34  package Generic_Simulator is
35    procedure Add_Event(E: in Root_Event.Event'Class);
36    procedure Run;
37  end Generic_Simulator;
```

```
38    --
39    -- Generic simulator body
40    --
41  package body Generic_Simulator is
42    Q: aliased Event_Queue.Queue(100);
43    procedure Add_Event(E: in Root_Event.Event'Class) is
44    begin
45      Event_Queue.Put(E, Q'Access);
46    end Add_Event;
47    procedure Run is
48    begin
49      while not Event_Queue.Empty(Q'Access) loop
50        Root_Event.Simulate(Event_Queue.Get(Q'Access));
51      end loop;
52    end Run;
53  end Generic_Simulator;
```

Generic_Simulator is instantiated in three stages! Firstly, the packages Event_Tree_Queue and Event_Array_Queue 'conveniently' supply all the items promised by the signature. They are obtained by instantiating our familiar generic priority queue packages.

```
54    --
55    -- Instantiation of event queue implemented by trees
56    --
57  with Tree_HPQ;
58  with Root_Event; use Root_Event;
59  package Event_Tree_Queue is new Tree_HPQ(Event'Class);
60    --
61    -- Instantiation of event queue implemented by arrays
62    --
63  with Array_HPQ;
64  with Root_Event; use Root_Event;
65  package Event_Array_Queue is new Array_HPQ(Event'Class);
```

Secondly, these packages are used to instantiate the signature to obtain Event_Queue_1 and Event_Queue_2. The instantiation is simple because the actual subprograms are supplied by default.

```
66    --
67    -- Instantiation of the signature with tree queue
68    --
69  with Event_Tree_Queue; use Event_Tree_Queue;
70  with Generic_Event_Priority_Queue;
71  package Event_Queue_1 is new Generic_Event_Priority_Queue(Queue);
72    --
73    -- Instantiation of the signature with array queue
```

```
74    --
75  with Event_Array_Queue; use Event_Array_Queue;
76  with Generic_Event_Priority_Queue;
77  package Event_Queue_2 is new Generic_Event_Priority_Queue(Queue);
```

Finally, we instantiate the generic simulator package to obtain two simulators: Simulator_1 and Simulator_2. As required by §12.7(4–5), the actual package parameters are themselves instantiations of the generic formal package parameters.

```
78    --
79    -- Instantiation of the simulator with tree queue
80    --
81  with Generic_Simulator;
82  with Event_Queue_1;
83  package Simulator_1 is new Generic_Simulator(Event_Queue_1);
84    --
85    -- Instantiation of the simulator with array queue
86    --
87  with Generic_Simulator;
88  with Event_Queue_2;
89  package Simulator_2 is new Generic_Simulator(Event_Queue_2);
90
```

Note that so far we have not said anything about the rocket! The hierarchy of events for the rocket simulation is defined in a child of Root_Event. Except for the package names, the source code for the rest of the rocket packages is unchanged and is omitted here.

```
91    --
92    -- Rocket root event
93    --
94  package Root_Event.Rocket_Event is
95    type Event is abstract new Root_Event.Event with null record;
96  end Root_Event.Rocket_Event;
```

The main subprogram can use both simulators. In a more realistic program, tasking would be used to run the two simulators concurrently.

```
97    --
98    -- Run two simulators
99    --
100 with Simulator_1;
101 with Simulator_2;
102 with Root_Event.Rocket_Event; use Root_Event.Rocket_Event;
103 with Root_Event.Rocket_Event.Engine;
104 with Root_Event.Rocket_Event.Telemetry;
105 with Root_Event.Rocket_Event.Steering;
```

```
106 procedure RocketG is
107 begin
108    for I in 1..15 loop
109       Simulator_1.Add_Event(Engine.Main_Engine_Event'(Engine.Create));
110       Simulator_1.Add_Event(Engine.Aux_Engine_Event'(Engine.Create));
111       Simulator_1.Add_Event(Telemetry.Create);
112       Simulator_1.Add_Event(Steering.Create);
113    end loop;
114    for I in 1..15 loop
115       Simulator_2.Add_Event(Engine.Main_Engine_Event'(Engine.Create));
116       Simulator_2.Add_Event(Engine.Aux_Engine_Event'(Engine.Create));
117       Simulator_2.Add_Event(Telemetry.Create);
118       Simulator_2.Add_Event(Steering.Create);
119    end loop;
120
121    Simulator_1.Run;
122    Simulator_2.Run;
123 end RocketG;
```

7.10 Generic children*

A non-generic package may have generic children §10.1.1; for example, Ada.Numerics is not generic but its children are. However, the child of a generic unit must itself be generic §10.1.1(16–19). If the child package were not generic, the child would have to be compiled for every existing instantiation of the parent, whether it is needed or not. By requiring that the child be generic, instances are created only when explicitly requested.

The following example demonstrates the rules; a realistic application will be given in Section 10.8. Note that the instantiation of the child ‡20 requires two ‘with’ clauses. The first is the usual ‘with’ clause for the generic package being instantiated. However, what is actually being instantiated is the generic child of the *instance* of the parent, so it must also be ‘with’ed. This is what enables the instance to access the generic actual parameter of the instance, so that Child_Instance.V2 is of type Integer.

```
1    --                                                    -- File: GENCHILD
2    --  Generic child of a generic package
3    --
4  generic
5    type T is private;
6  package Parent is
7    V1: T;
8  end Parent;
9
```

```
10 generic
11 package Parent.Child is
12    V2: T;
13 end Parent.Child;
14
15 with Parent;
16 package Parent_Instance is new Parent(Integer);
17
18 with Parent.Child;
19 with Parent_Instance;
20 package Child_Instance is new Parent_Instance.Child;
```

7.11 Limitations of the contract model**

The explanation of the contract model in Section 7.2 was only approximate. Occasionally, you will have to understand the precise details of the model, as discussed in this section.

Consider the following rule:

> 3 ... If the parent type is nonlimited, then each of the components of the record_extension_part shall be nonlimited. ... **§3.9.1**

Does it apply to the declaration of T1 in the following generic package?

```
1 generic                                                    -- File: LEGAL
2    type Parent is tagged limited private;
3    type Component_Type is limited private;
4 package P is
5    type T1 is new Parent with
6       record
7          X: Component_Type;
8       end record;
9 private
10   type T2 is new Parent with
11      record
12         X: Component_Type;
13      end record;
14 end P;
```

The formal type Parent is limited, so the component X in the extension can be of the formal limited type Component_Type. This generic package specification compiles successfully. The technical term is 'assume-the-best': since the formal parameters are declared to be limited, the compiler assumes that the actual parameters will also be limited.

Consider, however, an instantiation of P where the actual parameter associated with Parent is *unlimited* and the actual parameter associated with Component_Type is *limited*:

```
15 with P;
16 package Legal is
17    type Un_Lim is tagged null record;
18    type Lim is tagged limited null record;
19    package Instance is new P(Un_Lim, Lim);
20 end Legal;
```

The type Instance.T1 is an extension of the unlimited type Un_Lim with a limited component of type Lim—clearly, an illegal situation.

> 11 In a generic unit Legality Rules are enforced at compile-time of the generic_declaration and generic body, given the properties of the formals. In the visible part and formal part of an instance, Legality Rules are enforced at compile-time of the generic_instantiation, given the properties of the actuals. In other parts of an instance, Legality Rules are not enforced; this rule does not apply when a given rule explicitly specifies otherwise. **§12.3**

According to the second sentence of this rule, legality rules such as §3.9.1(3) are enforced during the compilation of the of the *instantiation*; the declaration of T1 in the *instance* is an error. This will be somewhat surprising, since there is nothing wrong with the instantiation (given the generic formal part) and the generic specification itself compiles correctly.

The contract model is not between the *generic specification* and the instantiation, but between the *generic body* and the instantiation. An instantiation will never cause the generic body to become illegal; conversely, a modification of the body cannot make an instantiation illegal. Almost all violations of the contract will be caught as conflicts between the generic actual parameters and the generic formal parameters, but occasionally—as shown here—the conflict may be with the specification.

What about the declaration of T2 ‡10–13? This is rejected during the compilation of the instantiation by the third sentence of §12.3(11), together with the following sentence that appears later in the paragraph quoted above:

> 3 ... In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. **§3.9.1**

A list of the legality rules that apply in the private part of a generic instance can be found in the index of the *ARM* under the entry ‘generic contract issues’.

Finally, what about the extension T3 in the following body of P?[7]

```
21 package body P is
22    type T3 is new Parent with
23       record
24          X: Component_Type;
25       end record;
26 end P;
```

It appears that the instantiation will cause the body to be illegal—precisely the situation that the contract model was intended to avoid. The solution is simply to forbid this construct so that the compilation of the *body* is already illegal, regardless of the actual parameters supplied during the instantiation.

4 A type extension shall not be declared in a generic body if the parent type is declared outside that body.	**§3.9.1**

This is known as 'assume-the-worst'. Though it is unlikely that an extension of a formal type in the body will cause a problem, it is in fact possible, so all such extensions are forbidden. The workaround in this case is simply to move the extension to the private part of the specification.

The flexibility of 'assume-the-best' when compiling the specification should not be a problem because the generic specification is the interface that is visible to users of the package.

[7] Strictly speaking P is not allowed to have a body, but that can be easily remedied.

8 Types Revisited

8.1 Characters and strings

Thirty pages of Annex §A of the *ARM* are devoted to packages for character and string handling!

Ada.Characters.Latin_1 §A.3.3 supplies names for all characters (except for the digits and the upper-case letters 'A' through 'Z'). Ada.Characters.Handling §A.3.2 contains functions such as Is_Alphanumeric for classifying characters, as well as conversion functions such as To_Upper and To_Wide_Character. Note that the category *letters* includes international characters such as the letter ç (LC_C_Cedilla) used in French; the predefined upper/lower case conversion functions take account of these characters.

Ada.Strings §A.4.1 has child packages that provide extensive operations for three string types (see Figure 8.1):[1]

- Fixed—This is the predefined type String.
- Bounded—A bounded string object must be declared with a maximal length; the current length is automatically maintained by the library subprograms.
- Unbounded—An unbounded string has no maximal length.

Figure 8.1 also shows null-terminated strings that are defined in Interfaces.C.Strings §B.3.1 and are used to manipulate strings passed to or received from external subprograms in written in C.[2]

Obviously, the more flexible the string type, the more overhead is required. For bounded strings, you will declare a maximum size that is at least as large as the longest string of the type. For unbounded strings, heap allocation and deallocation may be needed for each operation. The subprogram libraries for all three types are very similar so your choice can be based on the requirements of your application.

Package Ada.Strings.Maps §A.4.2 implements operations on type Character_Set, which will be familiar if you have used a string processing language such as SNOBOL and Icon. These sets are used as patterns in search operations such as 'find the first occurrence of any upper case character'. The package also declares the type Character_Mapping, which can be used to translate one set of characters to another. For example, the following statement will do a case-insensitive search for a lower-case Pattern within Source by mapping Source to lower case using the predefined map Lower_Case_Map §A.4.6(5):

[1] An implementation need not use these data structures; furthermore, information on the index bounds is not shown in the figure.

[2] More generally, any software that uses the C convention.

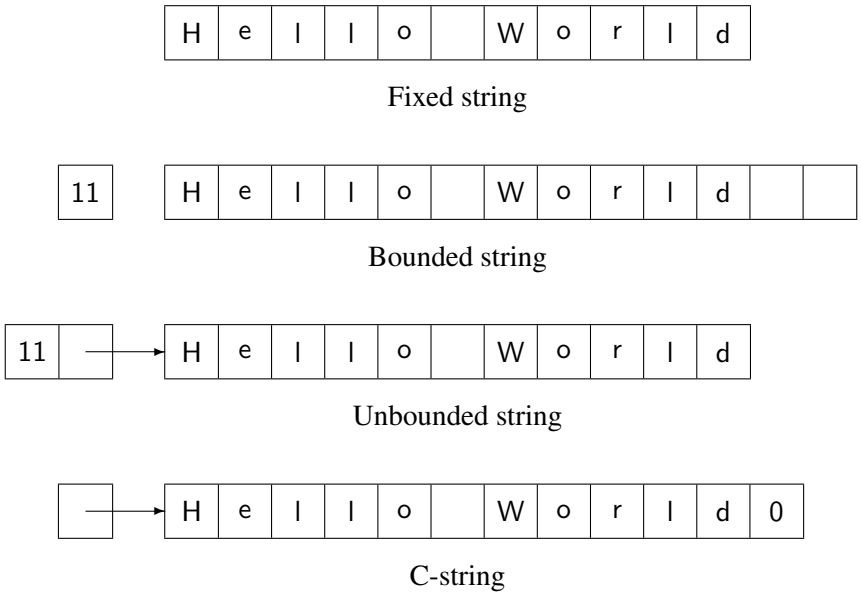

Figure 8.1: Strings in Ada

```
Ada.Strings.Fixed.Index(
  Source, Pattern, Ada.Strings.Forward,
  Ada.Strings.Maps.Constants.Lower_Case_Map);
```

You can read detailed descriptions of these packages in the *ARM*. We now present a case study showing how they can be used.

Case study: Ada to LaTeX

LaTeX (Lamport 1986, Diller 1993) is the software I use to format and typeset my books. The text of the book is written in ordinary ASCII characters together with formatting commands. The LaTeX software formats the text, creating a device-independent file that can be previewed on a screen and then printed. I have written a program that takes an Ada program and inserts formatting commands so that the program can be used in a LaTeX document without manually keying in the text or the commands.

The transformations performed on the text are as follows:

- The characters #, & and _ that are used in the Ada syntax are preceded by the escape character \.
- The reserved words are enclosed in the command for boldface font `{\bf }`.[3]
- The program is enclosed in a tabbing environment, where \> denotes a tab stop and \\ denotes the end of line.

[3]The program will incorrectly perform this transformation on a reserved word that occurs within a string literal! We leave it to the reader to correct the program.

- Comments are formatted using macros that I defined: `\cml{}` for a comment that takes up an entire line and `\cm{}` for a comment that trails a source line.

Before looking at the program, we must discuss the choice of string types. Strings are needed for the program lines to be processed and for a table of reserved words. Since the size of these strings can be bounded, there is no reason to use dynamic strings. Bounded strings are obviously appropriate, but fixed strings can be used as well, since the library subprograms automatically justify and pad their results. For example, the Move procedure allows you to choose the justification direction, padding character and the action to be taken if there are too many characters in the source string:

```
procedure Move(
   Source:  in String;
   Target:  out String;
   Drop:    in Truncation := Error;
   Justify: in Alignment := Left;
   Pad:     in Character := Space);
```

I have written several versions of this program. the one displayed here uses fixed strings for reserved words and bounded strings for the program lines; other versions are on the CD-ROM. (The file names are **TOLATEXB, TOLATEXF, TOLATEXP**, respectively.

The package Ada.Strings.Bounded consists entirely of a generic package that must be instantiated with the maximum length of the bounded string type.[4] Types resulting from different instantiations cannot be directly converted to each other, though you can convert indirectly by converting to and from String. The instantiation Line of the bounded string package is done at the library level to save compilation time when debugging the program.

```
1 with Ada.Strings.Bounded;                                    -- File: TOLATEX
2 package Line is new Ada.Strings.Bounded.Generic_Bounded_Length(120);
```

The reserved words are stored in a constant table of fixed strings ‡23–37. The components of this array are of a string subtype constrained to the length of the longest reserved word. The table could be initialized by manually padding each word (`"if␣␣␣␣␣␣␣␣"`), but it is easier to write a function to do this automatically ‡16–21. The table is sorted so that it can be efficiently searched using binary search ‡47–64. Formatting commands are declared as constants ‡39–45.

```
3  --
4  -- Format Ada program with LaTeX commands.
5  --
6  with Line;
7  with Ada.Strings.Maps.Constants;
8  with Ada.Strings.Fixed;
9  with Ada.Text_IO;
10 with Ada.Command_Line;
```

[4]This is a good test to see if your implementation does code sharing among instantiations. If not, the size of your executable code will increase with each instantiation. You may want to 'round up' the size of some strings to save code memory at the expense of data memory.

```
11 use Line, Ada.Strings;
12 procedure ToLaTeX is
13
14    subtype Keyword is String(1..9);
15
16    function P(Source: in String) return Keyword is
17       Target: Keyword;
18    begin
19       Fixed.Move(Source, Target);
20       return Target;
21    end P;
22
23    Words: constant array(Natural range <>) of Keyword := (
24       P("abort"),     P("abs"),       P("abstract"),  P("accept"),    P("access"),
25       P("aliased"),   P("all"),       P("and"),       P("array"),     P("at"),
26       P("begin"),     P("body"),      P("case"),      P("constant"),  P("declare"),
27       P("delay"),     P("delta"),     P("digits"),    P("do"),        P("else"),
28       P("elsif"),     P("end"),       P("entry"),     P("exception"), P("exit"),
29       P("for"),       P("function"),  P("generic"),   P("goto"),      P("if"),
30       P("in"),        P("is"),        P("limited"),   P("loop"),      P("mod"),
31       P("new"),       P("not"),       P("null"),      P("of"),        P("or"),
32       P("others"),    P("out"),       P("package"),   P("pragma"),    P("private"),
33       P("procedure"), P("protected"), P("raise"),     P("range"),     P("record"),
34       P("rem"),       P("renames"),   P("requeue"),   P("return"),    P("reverse"),
35       P("select"),    P("separate"),  P("subtype"),   P("tagged"),    P("task"),
36       P("terminate"), P("then"),      P("type"),      P("until"),     P("use"),
37       P("when"),      P("while"),     P("with"),      P("xor") );
38
39    Tab_Size:       constant Natural := 2;                          -- At least 2
40    Tab_String:     constant String := "\>";
41    Bold:           constant String := "{\bf ";
42    EOL:            constant String := "\\";
43    Trail_Comment:  constant String := "\cm{";
44    Line_Comment:   constant String := "\cml{";
45    Escapes:        constant Maps.Character_Set := Maps.To_Set("_&#");
46
47    function Search(S: String) return Boolean is
48       Mid: Natural;
49       Low: Natural := Words'First;
50       High: Integer := Words'Last;
51       K: Keyword;
52    begin
53       if S'Length > Keyword'Length then return False;
54       else Fixed.Move(S, K);
55       end if;
```

```
56      loop
57        if Low > High then return False; end if;
58        Mid := (Low + High) / 2;
59        if K = Words(Mid) then return True;
60        elsif K < Words(Mid) then High := Mid - 1;
61        else Low := Mid + 1;
62        end if;
63      end loop;
64    end Search;
```

Procedure Split_Comment searches for the first non-blank character ‡67 and the first comment sequence ‡68. The results are used to split the program line into its source code and comment substrings. Null_Bounded_String is predefined §A.4.4(7). Note that the concatenation operator "&" is overloaded for mixed bounded and fixed string operands §A.4.4(21–25).

```
65    procedure Split_Comment(
66        S: in Bounded_String; Source, Comment: out Bounded_String) is
67      Start: Natural := Index_Non_Blank(S);
68      MM:  Natural := Index(S, "--");
69    begin
70      if Start = 0 or MM = 0 then
71        Source := S;
72        Comment := Null_Bounded_String;
73      elsif Start = MM then
74        Source := Null_Bounded_String;
75        Comment := Line_Comment & Tail(S, Length(S)-MM-1) & "}";
76      else
77        Source := Head(S, MM-1);
78        Comment := Trail_Comment & Tail(S, Length(S)-MM-1) & "}";
79      end if;
80    end Split_Comment;
```

Procedure Tabbing begins by searching for the first non-blank character in a line and computing the number of tabs ‡82. Then it replaces the blanks by the tab string ‡87–88. The function "*" §A.4.3(105) is used to replicate the tab string ‡88:

```
function "*" (Left: in Natural; Right: in String) return String;
```

The ‘use’ clause for Fixed is needed so that operator syntax can be used (see Quiz C53).

```
81    procedure Tabbing(S: in out Bounded_String) is
82      Blank_Count: Natural :=
83        ((Index_Non_Blank(S)-1) / Tab_Size) * Tab_Size;
84      use Fixed;
```

```
85  begin
86    if Blank_Count > 0 then
87      Replace_Slice(
88        S, 1, Blank_Count, (Blank_Count/Tab_Size) * Tab_String);
89    end if;
90  end Tabbing;
```

The procedure Insert_Escapes is a bit more difficult than Tabbing because once a character such as # is replaced by \#, the search for the next character in the set must start after the character already replaced; otherwise an infinite loop will result. This is done by maintaining a current position variable Pos ‡93 and using the function Tail ‡97 §A.4.4(72) to search within the tail of a string. The character set Escapes ‡45 is created using function To_Set which creates a set from a Character_Sequence (a synonym for String). Character sets can also be created from a Character_Range §A.4.2(6) which is defined by two characters bounding a range.

```
91  procedure Insert_Escapes(S: in out Bounded_String) is
92    First: Natural;
93    Pos: Natural := 0;
94  begin
95    loop
96      exit when Pos >= Length(S);
97      First := Index(Tail(S, Length(S)-Pos), Escapes);
98      exit when First = 0;
99      Pos := Pos + First;
100     Insert(S, Pos, "\");
101     Pos := Pos + 2;
102   end loop;
103 end Insert_Escapes;
```

Procedure Emphasize_Keywords also has to keep track of the current position. Function Find_-Token ‡113 §A.4.3(67–68) simplifies programming by directly finding the first substring composed of characters Inside the set Word_Set and delimited by characters not in the set. Note that Word_Set must include the underscore character ‡106–107, otherwise a reserved word that is part of an identifier like My_Type would be incorrectly emphasized.

```
104 procedure Emphasize_Keywords(S: in out Bounded_String) is
105   use Maps;
106   Word_Set: constant Character_Set :=
107     Constants.Alphanumeric_Set or To_Set('_');
108   First, Last: Natural;
109   Pos: Natural := 0;
110 begin
111   loop
112     exit when Pos >= Length(S);
113     Find_Token(Tail(S, Length(S)-Pos), Word_Set, Inside, First, Last);
```

```
114         exit when Last = 0;
115         First := First + Pos; Last := Last + Pos; Pos := Last + 1;
116         if Search(Slice(S, First, Last)) then
117           Replace_Slice(S, First, Last, Bold & Slice(S, First, Last) & '}');
118           Pos := Pos + Bold'Length + 1;
119         end if;
120       end loop;
121     end Emphasize_Keywords;
```

In the main subprogram ‡143–171, Ada.Command_Line §A.15 is used to obtain the input file name from the command line ‡150, which is then used to create the name of the output file ‡151. The files are opened and after writing LaTeX preamble commands, Main_Loop is called ‡161 to process the data. Upon completion, LaTeX commands are written before closing the files.

The main loop is split off into a separate procedure ‡122–141. Input and output are not defined for bounded strings; instead, ordinary Ada.Text_IO subprograms are used ‡128, 137 together with conversions between fixed and bounded strings. Split_Comment is called ‡130 to divide the line into a string with the source code and a string with the comment. Tabbing and reserved words (called 'keywords' in the program) are processed in the source code only ‡132, 133, while escape characters must also be processed in the comments ‡134, 136.

```
122   procedure Main_Loop(Input, Output: in out Ada.Text_IO.File_Type) is
123     Buffer: String(1..Line.Max_Length);
124     Last: Natural;
125     S, Source, Comment: Bounded_String;
126   begin
127     loop
128       Ada.Text_IO.Get_Line(Input, Buffer, Last);
129       S := To_Bounded_String(Buffer(1..Last));
130       Split_Comment(S, Source, Comment);
131       if Length(Source) > 0 then
132         Tabbing(Source);
133         Emphasize_Keywords(Source);
134         Insert_Escapes(Source);
135       end if;
136       Insert_Escapes(Comment);
137       Ada.Text_IO.Put_Line(Output, To_String(Source&Comment&EOL));
138     end loop;
139   exception
140     when Ada.Text_IO.End_Error => null;
141   end Main_Loop;
142
```

```
143 begin
144   if Ada.Command_Line.Argument_Count /= 1 or else
145     Fixed.Index(Ada.Command_Line.Argument(1), ".") = 0 then
146       Ada.Text_IO.Put_Line("Usage: ToLaTex FileName (with extension)");
147   else
148     declare
149       use Fixed, Ada.Text_IO;
150       Input_Name: String := Ada.Command_Line.Argument(1);
151       Output_Name: String := Input_Name(1..Index(Input_Name, ".")) & "tex";
152       Input: File_Type;
153       Output: File_Type;
154     begin
155       Open(Input, In_File, Input_Name);
156       Create(Output, Out_File, Output_Name);
157       Put_Line(Output, "\documentstyle{article}");
158       Put_Line(Output, "\begin{document}");
159       Put_Line(Output, "\begin{tabbing}");
160       Put_Line(Output, 10*(Tab_Size*"x" & "\=") & "\kill");
161       Main_Loop(Input, Output);
162       Put_Line(Output, "\end{tabbing}");
163       Put_Line(Output, "\end{document}");
164       Close(Input);
165       Close(Output);
166       Put_Line("Created file "& Output_Name);
167     exception
168       when Name_Error => Put_Line("No such file");
169     end;
170   end if;
171 end ToLaTeX;
```

8.2 Discriminants

1 A composite type (other than an array type) can have discriminants, which parameterize the type. A known_discriminant_part specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An unknown_discriminant_part in the declaration of a partial view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a partial view are indefinite subtypes. **§3.7**

2 discriminant_part ::=
  unknown_discriminant_part | known_discriminant_part

3 unknown_discriminant_part ::= (<>)

4 known_discriminant_part ::=
  (discriminant_specification {; discriminant_specification})

5 discriminant_specification ::=
  defining_identifier_list : subtype_mark
    [:= default_expression] |
  defining_identifier_list : access_definition
    [:= default_expression]

6 default_expression ::= expression

We have met known discriminants in the implementation of a queue by an array, and unknown discriminants in generic formal parameters:

```
type Queue(Size: Positive) is
  record
    Data: Vector(0..Size);
    Free: Natural := 0;
  end record;

generic
  type Item is (<>);
package Priority_Queue is ...
```

Discriminants are primarily used to parameterize record types, but they can also parameterize tasks and protected objects (Sections 13.2, 14.8). A discriminant in a record type declaration declares a *constant* component of the record §3.3(18), §3.8(9). When an object of a discriminated type is created, a value must be given for each discriminant either by a discriminant constraint §3.7.1 (as we did in the priority queue programs) or by an initial value. As with unconstrained arrays, a formal parameter with a discriminant takes its constraint from an actual parameter.

Changing the value of a discriminant would break type checking:

```
Q: Queue := (Size => 10, Free => 0, Data => (others => 0));

Q.Size := 100;                    -- Error, otherwise ...
Q.Data(62) := 35;                 -- ... this would be legal
```

While it is not possible to change the discriminant alone, the record itself can be assigned, subject to a check that the discriminants match. Both the assignment statement and parameter passing are defined in terms of type conversion §5.2(11), §6.4.1(10,11,14), which includes a constraint check §4.6(51):

```
subtype Queue10 is Queue(10);
Q, R: Q10;
S:    Queue(20);

Q := R;                                  -- OK
Q := S;                                  -- Raises Constraint_Error

procedure Unconstrained(A: in out Queue);
procedure Constrained(A: in out Queue10);

Constrained(R);                          -- OK
Constrained(S);                          -- Raises Constraint_Error
Unconstrained(S);                        -- OK
```

Within a record declaration, a discriminant can be used in a default expression for another component, but if it is used to constrain a component, it must appear directly and not part of an expression §3.8(12).

```
type Queue(Size: Positive) is
  record
    Data: Vector(0..Size+1);                    -- Error
    Free: Natural := 0;
  end record;
```

8.3 Variant records

Discriminants can be used to create variant records, where the existence of some of the record components *depends on* §3.7(22) the discriminants. Programming with variants used to be extremely important, but in many cases it is better to use type extension. The problem with variants is that you typically need large case statements to select the computation that is appropriate for the components that exist for each variant. These case statements are elegant if you are just changing or adding operations because the source code for all variants is located in a single place. However, if you expect to modify or add variants, the program will be difficult to maintain because all these case statements will need to be changed. It is much easier to use type extension, since most subprograms can be inherited and only a few will need to be modified or added.

Furthermore, all components of a record must have distinct identifiers §3.8(9); if you want to factor out common components, you will have to use nested variants and nested case statements. Factoring is trivial to do using repeated extensions in a class of derived types: first you extend with the common fields and then extend again for each ‘variant’.

An important use for variant records is to convert unstructured data to a structured record. For example, a buffer of bytes from a communications link or an operating system service can be converted to any one of a set of structures depending upon a discriminant that functions as an explicit tag.

Case study: message conversion

The following program shows how to use variant records for structuring a sequence of bytes.[5] Type Structured_Message ‡17–28, is a variant record whose discriminant is an enumerated type Codes ‡14, while type Raw_Message ‡33 is just an array of bytes. The record has four common components: the discriminant Code and the components Addressee, Sender and Sequence_Number; the other components depend on the discriminant Code.

```
1   --                                                    -- File: MESSAGE
2   -- Message representation conversion with variant records
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
6  with System; with Unchecked_Conversion;
7  procedure Message is
8
9    type Byte is mod 2**System.Storage_Unit;
10   for Byte'Size use System.Storage_Unit;
11   package Byte_IO is new Ada.Text_IO.Modular_IO(Byte);
12   use Byte_IO;
13
14   type Codes is (M0, M1, M2, M3);
15   for Codes'Size use System.Storage_Unit;
16
17   type Structured_Message(Code: Codes) is
18     record
19       Addressee: Byte;
20       Sender: Byte;
21       Sequence_Number: Byte;
22       case Code is
23         when M0 => null;
24         when M1 => A: Integer;
25         when M2 => B: Character;
26         when M3 => C: Integer; D: Integer;
27       end case;
28     end record;
29   pragma Pack(Structured_Message);
30
```

[5] Modular types are discussed in Section 10.4 and representation items such as Size and Pack in Section 8.4.

```
31    Max_Bytes: constant Integer :=
32      Structured_Message'Size/System.Storage_Unit;
33    type Raw_Message is array(1..Max_Bytes) of Byte;
34    pragma Pack(Raw_Message);
```

The function Send_Message converts a message to an array of bytes. The procedure Process_-Message converts a raw message to a variant record and 'processes' it according to its code. These conversions are done using the functions To_Raw ‡35–36 and To_Structured ‡38–39 obtained by instantiating the generic function Unchecked_Conversion §13.9. This violates type checking, so we must carefully check that the conversions are meaningful. The size of Raw_Message is computed ‡31–32 from the size of Structured_Message, so that we are not in danger of smearing memory, but there is no assurance that the value of the byte for the message code corresponds to the value of the discriminant for the message structure.

```
35    function To_Raw is new Unchecked_Conversion(
36      Source => Structured_Message, Target => Raw_Message);
37
38    function To_Structured is new Unchecked_Conversion(
39      Source => Raw_Message, Target => Structured_Message);
40
41    function Send_Message(S: Structured_Message) return Raw_Message is
42    begin
43      return To_Raw(S);
44    end Send_Message;
45
46    procedure Process_Message(R: Raw_Message) is
47      S: Structured_Message := To_Structured(R);
48    begin
49      Put("Message number "); Put(S.Sequence_Number);
50      Put(" of type " & Codes'Image(S.Code) & " ");
51      Put(" sent by "); Put(S.Sender);
52      Put(" to "); Put(S.Addressee); New_Line;
53      Put("Text of message is ");
54      case S.Code is
55        when M0 => null;
56        when M1 => Put(S.A);
57        when M2 => Put(S.B);
58        when M3 => Put(S.C); Put(S.D);
59      end case;
60      New_Line; New_Line;
61    end Process_Message;
```

To test the program, aggregates for message are created, sent and then processed. In an aggregate, the discriminant is just another component, with the restriction that it must be static since it determines which components are needed §4.3.1(17).

```
62 begin
63    Process_Message(Send_Message((M1, 11, 32, 1, 54)));
64    Process_Message(Send_Message((M2, 32, 11, 2, 'X')));
65    Process_Message(Send_Message((M3, 32, 11, 3, 45, 68)));
66 end Message;
```

Note that the addition of a code, or modification of the components for a code, will require changes to the case-statement. A better approach would be to use tagged types and dispatching. Streams (Section 11.3) can be used for input–output of the tagged types.

8.4 Representation items

Ideally, an Ada program will be completely portable and will execute without modification on any computer. In practice, computer hardware varies so much that perfect portability is impossible, especially if you are writing an embedded system that needs to access hardware interfaces.

The Ada solution to the conflicting requirements of portability and hardware interfacing is to have the language standard specify the syntactic and semantic framework for the hardware interface, but to leave the implementation details to each compiler. This framework is discussed in §13 'Representation Issues'. The section is full of paragraphs entitled 'Implementation Advice' and 'Implementation Permissions', indicating that even a validated compiler need not fully support the specifications. Before choosing an Ada compiler for an embedded system, you must carefully study the compiler's documentation to see it satisfies your requirements. Since the constructs are standardized, it is easy to study and compare different implementations.

We now survey representation issues, using the program in the previous section as an example.

Basic information about the implementation is given in package System §13.7; the most important is the constant Storage_Unit ‡9–10, which gives the numbers of bits in the smallest addressable Storage_Element, usually (but not necessarily) an 8-bit 'byte'. The package also defines the type Address used for representing machine addresses as opposed to language-defined access values. Storage_Element itself is defined in System.Storage_Elements §13.7.1 together with operations for address arithmetic. You can obtain the address of an object or subprogram by using the attribute Address §13.3(10–12). Conversion between access values and machine addressed is supported by package System.Address_To_Access_Conversions §13.7.2.

Most representation characteristics are given as attributes §13.3. Thus Structured_Message'Size ‡32 gives the number of bits required to store a value of this type. Attribute definition clauses §13.3(2) can be used to specify these characteristics. The values of the enumeration type Codes ‡14 might be stored by default in a 32-bit word, but the attribute definition clause ‡15 specifies that it should be stored in a single storage unit. Of course, if there are more values in the type than there is space in a storage unit (256 for an 8-bit byte), the clause would be illegal §13.1(12) and the program would not compile.

Pragma Pack §13.2 is a representation pragma used to specify that storage for Structured_Message ‡29 and Raw_Message ‡34 should be minimized. The meaning of 'minimized' is implementation-defined, so this pragma is less portable than other representation items such as record layout

clauses §13.5 discussed in Section 8.6.

An enumeration representation clause §13.4 can be used to specify the representation of enumeration literals in place of the default representation—the position numbers §13.4(8). The following clause specifies that each message code is assigned a separate bit:

```
for Codes use
  (M1 => 2#0001#, M2 => 2#0010#,
  M3 => 2#0100#, M4 => 2#1000#);
```

8.5 Deeper into discriminants

Unconstrained variables

> 1 If a discriminated type has default_expressions for its discriminants, then unconstrained variables of the type are permitted, and the discriminants of such a variable can be changed by assignment to the variable. ... **§3.7.2**

The default expression serves two purposes: it gives the initial value of the discriminant and it is a syntactic marker that unconstrained records of this type can be created. Unconstrained records can be used to implement bounded strings (Ada.Strings.Bounded §A.4.4).

```
subtype Index is Natural range 0..255;
type Bounded_String(Length: Index := 80) is
    record
      S: String(1..Length);
    end record;

B1: Bounded_String(90) := (90, (others => ' '));
B2: Bounded_String(50) := (50, (others => ' '));
B3: Bounded_String;
```

B1 and B2 are declared as constrained variables; assignment of B1 to B2 or conversely will always raise Constraint_Error. B3 is an *unconstrained variable* whose discriminant is 80 as specified by the default expression. (Note that aliased §3.10(9) and allocated §4.8(4) objects cannot be unconstrained.) Either of the other variables can be assigned to B3. Note that it is still illegal to assign to the discriminant alone; by assigning to the entire record, we ensure the consistency of the discriminant with the components that depend upon it.

There is no difficulty in understanding how B2 with 50 characters can be assigned to B3, which has room for 80 characters. But how, you will certainly ask, can we assign B1, which contains a string of 90 characters, to B3. There are two answers. The first is simply that the assignment is legal in the Ada language and must be implemented by whatever means available, even if this requires implicit allocation of new memory for the modified variable B3. The second answer is that you can implement an unconstrained record by allocating the *maximum* amount of memory needed to contain any value of the type (Figure 8.2). The discriminant exists solely for the purposes of type-checking the value of Length against the size of the string. Note the importance of the default

expression for the discriminant: this ensures that unconstrained records such as B3 are created with *some* discriminant. Implicit allocation and deallocation of memory is difficult to implement, so most—if not all—Ada implementations allocate the maximum amount of memory needed for any value.

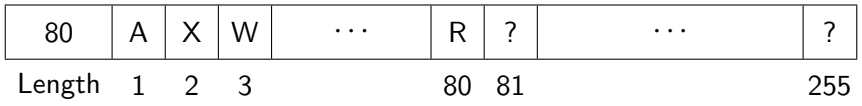

Figure 8.2: Unconstrained record

Sharp-eyed readers will have noted that we defined the subtype of the discriminant to be Index, which is constrained to 256 values, rather than, say, Positive. Declaring the discriminant subtype to be Positive would not be a good idea, because your implementation will probably try to allocate Positive'Last bytes to each unconstrained variable of this type, raising Storage_Error!

Discriminants of private types*

> 9 If the declaration of a partial view includes a known_discriminant_part, then the full_type_declaration shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1, "Conformance Rules"). ... **§7.3**
>
> 11 If a partial view has unknown discriminants, then the full_type_declaration may define a definite or an indefinite subtype, with or without discriminants.
>
> 12 If a partial view has neither known nor unknown discriminants, then the full_type_declaration shall define a definite subtype.

§7.3(9) is demonstrated by our priority queue package. Since the client can constrain an object of the private type, the full type must be fully conformant.

```
package Priority_Queue is
   type Queue(Size: Positive) is private;
private
   type Vector is array(Natural range <>) of Integer;
   type Queue(Size: Positive) is
      record
         Data: Vector(0..Size);
         Free: Natural := 0;
      end record;
end Priority_Queue;
```

If there was no discriminant, the client could obviously create an object of the type, so the full type must be definite, as required by §7.3(12).

```
package Priority_Queue is
   type Queue is private;
private
   type Vector is array(Natural range <>) of Integer;
   Max: constant := 1000;
   type Queue is
      record
         Data: Vector(0..Max);
         Free: Natural := 0;
      end record;
end Priority_Queue;
```

Alternatively, we could have declared the partial view to have unknown discriminants. This means that the type is indefinite §3.3(23), so the client cannot declare uninitialized objects of this type. In this case, there need be no restrictions on the full type, as noted in §7.3(11). You can use this form to force the client to call an explicit initialization function.

```
package Priority_Queue is
   type Queue(<>) is private;
   function Init return Queue;
private
   type Queue is array(Natural range <>) of Integer;
end Priority_Queue;

Q1: Priority_Queue.Queue;                    -- Error
Q2: Priority_Queue.Queue := Init;            -- OK
```

Inheriting discriminants**

> 18 For a type defined by a derived_type_definition, each discriminant of the parent type is either inherited, constrained to equal some new discriminant of the derived type, or constrained to the value of an expression. … **§3.7**

The discriminants are replaced if the derived type has a known discriminant part; otherwise, they are inherited §3.4(10–11).

Case study: simulation with discriminants**

In this version of the simulation, each object of type Event can represent multiple real events, where the number of events is given by the discriminant Number ‡6. Additional data is associated with the *i*'th event of the object; for simplicity, the data is just the *i*'th character within a string ‡9. Examples of the three alternatives of §3.7(18) are given: Engine_Event has replaced the discriminants ‡12, one of which is used to constrain the parent type ‡13; Steering_Event inherits the the discriminant Number ‡18; Telemetry_Event has no discriminants, because it constrains the parent with an expression ‡24.

```
1   --                                                    -- File: ROCKETD
2   -- Discrete event simulation of a rocket.
3   -- Inherited discriminants.
4   --
5  package Event_Package is
6    type Event(Number: Positive) is abstract tagged
7      record
8        Time: Simulation_Time;
9        Name: String(1..Number);
10     end record;
11
12   type Engine_Event(Count: Positive; Engines: Positive) is
13     new Event(Count) with
14     record
15       Fuel, Oxygen: Natural;
16     end record;
17
18   type Steering_Event is new Event with
19     record
20       Command: Commands;
21       Degree: Degrees;
22     end record;
23
24   type Telemetry_Event is new Event(2) with
25     record
26       ID: Subsystems;
27       Status: States;
28     end record;
29 end Event_Package;
30
```

The following declaration is illegal §3.7(13), because there is no way to know how much memory to allocate for the components of the parent type Event:

```
type Engine_Event(Count: Positive) is new Event with ...
```

8.6 Untagged derived types*

A new type can be derived from any type, not just from a tagged type. The concepts of primitive operations, inheritance and overriding are the same; however, untagged types cannot be extended, class-wide types cannot be declared, and there is no dynamic polymorphism.

Derived types are used to define numeric types, as we shall see in Chapter 10. Derived types can also be used to declare a new type with the same structure as an existing type. Consider the definition of the type Queue in the tree implementation of a priority queue:

```
type Queue is
  record
    Root: Link;
  end record;
```

The purpose of the definition is to ensure that Queue and Link are different types, even though Queue is implemented as a single Link. This improves type checking and makes it possible to overload a subprogram name on both types. The same effect could have been achieved by deriving Queue from Link:

```
type Queue is new Link;
```

It is always possible to convert within a derivation class §4.6(21,24):

```
procedure Get(I: out Integer; Q: in out Queue) is
begin
  if Q = null then raise Underflow; end if;
  Get(I, Link(Queue));                    -- Convert Queue to Link
end Get;
```

Case study: representation conversion

Another application of derived types is to convert between two different representations for a type §13.6. Given a record type, you can derived a new type and give a representation clause for the derived type. Type conversion between the two records will convert the representation. In the following example, an 8-bit instruction ‡5–11 is composed of two components: Op_Code of three bits and Operand of five bits. It will be more efficient to work with the unpacked representation and use the packed representation only for input–output.

```
1   --                                                    -- File: REP
2   -- Derived type for change of representation.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure Rep is
6    type Operators is (Op0, Op1, Op2, Op3, Op4, Op5, Op6, Op7);
7    type Byte is mod 256;
8    type Instruction is
9      record
10       Op_Code: Operators;
11       Operand: Byte range 0..31;
12     end record;
13
14   type Packed_Instruction is new Instruction;
15   for Packed_Instruction use
16     record
17       Op_Code at 0 range 0..2;
18       Operand at 0 range 3..7;
19     end record;
```

```
20    for Packed_Instruction'Size use 8;
21
22    PI: Packed_Instruction := (Op3, 26);
23    I: Instruction := Instruction(PI);
24 begin
25    Put(Operators'Image(PI.Op_Code));
26    Put(Byte'Image(PI.Operand));
27    Put(Operators'Image(I.Op_Code));
28    Put(Byte'Image(I.Operand));
29 end Rep;
```

Type Instruction uses the default representation, which will probably allocate a full word for each component, while the derived type Packed_Instruction ‡13 has a record representation clause ‡14–18 §13.5.1 that specifies the byte offset and bit positions of each component. The Size attribute definition clause ‡19 specifies that the entire record be packed into 8 bits.

The rules for untagged derived types are often different from those for tagged types. One such difference is discussed in the following subsection; others are left for the quizzes.

Derived types and discriminants**

Since no components can be added upon derivation of an untagged type, new discriminants must use the same memory allocated to the old discriminants.

> 12 For a type defined by a derived_type_definition, **§3.7**
> if a known_discriminant_part is provided in its declaration, then:
> 13 The parent subtype shall be constrained;
> 14 If the parent type is not a tagged type, then each discriminant of the derived type shall be used in the constraint defining the parent subtype;

If the event hierarchy were untagged, we could not declare Main_Engine_Event as shown below, because there is nowhere to store the new discriminant Engines. The other derivations are legal.

```
type Event(Number: Positive) is
  record
    Time: Integer;
    Name: String(1..Number);
  end record;

type Main_Engine_Event(Count: Positive; Engines: Positive) is
  new Event(Count);                                    -- Error!

type Aux_Engine_Event(Count: Positive) is new Event(Count);
type Steering_Event is new Event;
type Telemetry_Event is new Event(2);
```

9 Access Types

9.1 General access types

The access types we have been using are called *pool-specific* access types, because every access value points to a designated object that is allocated in a *storage pool* on the 'heap'. *General* access types §3.10(8) can be used to create pointers to declared objects, in addition to objects created by allocators:

```
type Ptr is access all Integer;
P1: Ptr := new Integer;

N: aliased Integer;
P2: Ptr := N'Access;
```

The reserved word **all** indicates that Ptr is a general *access-to-variable* type. P1 contains a pointer to an object allocated in a pool, while P2 contains a pointer to a declared object N. The pointer is created by applying the attribute Access[1] §3.10.2(24) to the object.

A general access type can be an *access-to-constant* type §3.10(10). Such types cannot be used to modify the designated type.

```
type Ptr is access constant Integer;
N: aliased Integer := 4;
P3: Ptr := N'Access;

P3.all := 5;                          -- Error !
```

Aliasing

The attribute Access can only be used on objects which are *aliased*, meaning that the object might have more than one access path: N and P2.**all**. In most cases, you explicitly declare an object to be aliased §3.3.1(2) as shown above. The explicit declaration is important, both as a warning to the programmer and as an indication to the compiler that optimization techniques such as storing a value in a register may not be appropriate for this object.

The exact rules for determining if an object is aliased (and hence if you can apply the Access attribute to it) are contained in §3.10(9). The dereference of an access-to-object value is aliased, so if X is a parameter of an access type or an access parameter (Section 9.5) X.**all**'Access is legal. Note that a formal parameter of any tagged type is aliased.

[1] Do not confuse this attribute with Address (Section 8.4), which obtains the actual address for hardware interface.

9.2 Access-to-subprogram types

An *access-to-subprogram type* §3.10(11) specifies the profile of a designated subprogram. An object of this type can be assigned an access value obtained by applying the attribute Access to any subtype-conformant subprogram §3.10.2(32):

```
type Func_Ptr is access function(L, R: Float) return Boolean;
function Compare(Left, Right: Float) return Boolean is ...
F: Func_Ptr := Compare'Access;
```

For applications of these types see the case study below and the one in Section 10.6.

9.3 Case study: callback

The following program demonstrates two applications of general access types: *ragged arrays* and *callbacks*. A ragged array is an array whose components are of different sizes. A callback is a programming technique used in event-driven software such as graphical user interfaces. A subprogram is associated with an object such as a 'button', and when the button is activated by a mouse click the subprogram is called.

Recall that you cannot create an array of strings, because the component of an array or record must be definite:

```
String_Array: constant array(Positive range <>) of String :=
                 ("Hello", "World");                          -- Error
```

To create an array of strings of arbitrary length, you would have to declare an array of pointers to strings:

```
type String_Ptr is access String;
String_Array: constant array(Positive range <>) of String_Ptr :=
                 (new String'("Hello"), new String'("World"));
```

But now every string would be stored twice: once as constant data and once on the heap when the aggregate is created.[2] This is unacceptable in embedded systems for two reasons: there may not be enough memory, and copying the strings from ROM to the heap may take too much time upon start or restart of the program.

In the following program, clicking is simulated by typing the names of the mouse button and the screen button. The event is processed by simply echoing the click data and displaying a message. Enumeration types are used for the screen buttons ‡7 and the mouse clicks ‡11 so that instantiations of Ada.Text_IO.Enumeration_IO can be used to read simulated clicks from the keyboard. Type Message_Ptr ‡15 is a access-to-constant general access type. Type Procedure_Ptr ‡16–17 is an access-to-subprogram type whose designated profile is a procedure that takes two parameters: one of type Clicks and the other of type String. These general access types are definite and can be components of the record type Callbacks ‡19–23, which contains the data structure associated with each callback.

[2] Implementations are encouraged to be more efficient in this case (Section 9.6).

```
1   --                                                          -- File: CALLB
2   -- General access types for ragged arrays and callbacks.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure CallB is
6
7     type Buttons is (OK, Apply, Help, Cancel);
8     package Buttons_IO is new Enumeration_IO(Buttons);
9     use Buttons_IO;
10
11    type Clicks is (Left, Middle, Right);
12    package Clicks_IO is new Enumeration_IO(Clicks);
13    use Clicks_IO;
14
15    type Message_Ptr is access constant String;
16    type Procedure_Ptr is
17       access procedure(C: in Clicks; S: in String);
18
19    type Callbacks is
20       record
21          Message: Message_Ptr;
22          Action: Procedure_Ptr;
23       end record;
24
```

We declare the string of each message as an aliased constant ‡25–28, which can be stored in ROM if need be. (Even if the strings themselves were not declared constant, the Access attribute could still be applied, yielding a constant view §3.10.2(25) of the string variables.) Next we declare a generic procedure Proc ‡30–38 and instantiate it for each screen button ‡40–43. Accesses to these strings and procedures are stored in the table Callback ‡45–49.

```
25    M_OK:     aliased constant String := "You have won the lottery.";
26    M_Apply:  aliased constant String := "Spread on evenly and rub in.";
27    M_Help:   aliased constant String := "Please help yourself.";
28    M_Cancel: aliased constant String := "Your credit card is cancelled.";
29
30    generic
31       B: in Buttons;
32    procedure Proc(C: in Clicks; S: in String);
33    procedure Proc(C: in Clicks; S: in String) is
34    begin
35       Put("Clicked " & Clicks'Image(C) & " on ");
36       Put(B); New_Line;
37       Put_Line(S);
38    end Proc;
39
```

```
40    procedure Proc_OK     is new Proc(OK);
41    procedure Proc_Apply  is new Proc(Apply);
42    procedure Proc_Help   is new Proc(Help);
43    procedure Proc_Cancel is new Proc(Cancel);
44
45    Callback: constant array(Buttons) of Callbacks := (
46      (M_OK'Access,       Proc_OK'Access),
47      (M_Apply'Access,    Proc_Apply'Access),
48      (M_Help'Access,     Proc_Help'Access),
49      (M_Cancel'Access,   Proc_Cancel'Access) );
```

When a button is entered ‡57, it is used as an index to select an action to be called ‡59. The mouse click C ‡56 and the string associated with the button are the actual parameters in the call.[3]

```
50  begin
51    loop
52      declare
53        B: Buttons;
54        C: Clicks;
55      begin
56        Put("Click mouse: "); Get(C);
57        Put("on button: "); Get(B);
58        Skip_Line;
59        Callback(B).Action(C, Callback(B).Message.all);
60      exception
61        when Data_Error => Put_Line("Invalid button pressed");
62        when End_Error => exit;
63      end;
64    end loop;
65  end CallB;
```

9.4 Accessibility rules

If you create a pointer to a non-heap object, you risk creating a ‘dangling pointer’:

```
1  procedure Level is                                          -- File: LEVEL
2    type Ptr is access all Integer;
3    function F return Ptr is
4      N: aliased Integer;
5    begin
6      return N'Access;                          -- Error!
7    end F;
8    P: Ptr := F;
```

[3] To call a parameterless subprogram pointed to by an access value, explicit dereferencing with **all** must be used.

```
 9 begin
10    null;
11 end Level;
```

The variable N is deallocated at the end of the function, but can still be accessed as P.**all** within the program, seriously compromising type-checking.

> 3 The accessibility rules, which prevent dangling references, are written in terms of **§3.10.2** *accessibility levels*, which reflect the run-time nesting of *masters*. ... a master is the execution of a ... block_statement, a subprogram_body .... An accessibility level is *deeper than* another if it is more deeply nested at run-time. For example, an object declared local to a called subprogram has a deeper accessibility level than an object declared local to the calling subprogram. The accessibility rules for access types require that the accessibility level of an object designated by an access value be no deeper than that of the access type. This ensures that the object will live at least as long as the access type, which in turn ensures that the access value cannot later designate an object that no longer exists. The attribute Unchecked_Access may be used to circumvent the accessibility rules.

(Unchecked_Access is defined in §13.10.)

In the above program, the main subprogram is a master and the access type Ptr is declared at its level. The called function F is at a deeper accessibility level. Therefore, there is a compilation error at ‡6 because the level of the object N is deeper than that of the type.

§3.10.2 goes on to define the accessibility level of each construct. The general principle is that a level represents a change of lifetime at run-time, not a compile-time change such as embedding within a package or renaming. Violations of the accessibility rules can generally be determined at compile-time §3.10.2(4). In a few cases (generics and access parameters), the check is made at run-time and a violation will cause the exception Program_Error to be raised §3.10.2(29).

A value of any access-to-type-T can be converted to any general access-to-type-T. (The converse is not possible: you cannot convert to a pool-specific access type.) The rules are given in §4.6(13–17): the designated type must be convertible to the target type, and accessibility levels and constantness must be respected.

9.5 Access parameters*

Formal access parameters allow a subprogram to be called with actual parameters of more than one access type.

§6.1

```
15  parameter_specification ::=
      defining_identifier_list : mode subtype_mark
        [:= default_expression]
      | defining_identifier_list : access_definition
        [:= default_expression]
```

§6.4.1

6 The type of the actual parameter associated with an access parameter shall be convertible (see 4.6) to its anonymous access type.

Case study: simulation with access parameter

The following version of the simulation uses access parameters to pass the queue to the Empty, Put and Get subprograms ‡7–9 of the priority queue package. The queue itself need not be dynamically allocated; as long as it is aliased ‡18, the attribute Access can be used to create an access value to pass to the subprograms ‡21–26, 29–30. Of course, an access parameter is also allowed to have an allocated object as its formal parameter. The advantage of using an access parameter in this program is that the *function* Get can modify its parameter without using explicit pointers.

-- **File: ROCKETA**

```
1   --
2   -- Discrete event simulation of a rocket.
3   -- Access parameters used in priority queue.
4   --
5  package Event_Queue is
6    type Queue is limited private;
7    function Empty(Q: access Queue) return Boolean;
8    procedure Put(E: in Event'Class; Q: access Queue);
9    function Get(Q: access Queue) return Event'Class;
10 private
11   ...
12 end Event_Queue;
13
14 with Event_Queue;
15 with Root_Event.Engine, Root_Event.Telemetry, Root_Event.Steering;
16 use Root_Event;
```

```
17 procedure RocketA is
18    Q: aliased Event_Queue.Queue;
19 begin
20    for I in 1..15 loop
21       Event_Queue.Put(
22          Engine.Main_Engine_Event'(Engine.Create), Q'Access);
23       Event_Queue.Put(
24          Engine.Aux_Engine_Event'(Engine.Create), Q'Access);
25       Event_Queue.Put(Telemetry.Create, Q'Access);
26       Event_Queue.Put(Steering.Create, Q'Access);
27    end loop;
28
29    while not Event_Queue.Empty(Q'Access) loop
30       Root_Event.Simulate(Event_Queue.Get(Q'Access));
31    end loop;
32 end RocketA;
```

Be sure to distinguish between an access parameter and a parameter that just happens to be of an access type. Access parameters have special characteristics not shared with parameters of an access type:

- An access parameter is of an anonymous access-to-variable type §6.1(24). Since it is of an anonymous type you cannot create new objects of the type. However, since it is of an access-to-variable type, you can dereference it and get an (aliased) designated object, just like the dereference of a parameter of an access type.
- An access parameter cannot be assigned a null value §4.6(49). Once you have successfully called the subprogram, checks for null no longer have to be done.
- The accessibility level is passed along with the access parameter §3.10.2(7) and dynamically checked.
- You can dispatch on an access parameter §6.1(24). In the following declarations, Proc1, but not Proc2, is a primitive subprogram for the tagged type Parent and is overridden by the declaration for Derived. If the actual parameter is any access to a type in Parent'Class, dispatching will be done. Proc2 for Derived simply overloads Proc2 for Parent.

```
type Parent tagged null record;
type Parent_Ptr is access Parent;
procedure Proc1(X: access Parent);
procedure Proc2(X: in Parent_Ptr);

type Derived is new Parent with null record;
type Derived_Ptr is access Derived;
procedure Proc1(X: access Derived);
procedure Proc2(X: in Derived_Ptr);
```

9.6 Storage pools*

Allocation of dynamic memory need not be done from a single heap.

1	Each access-to-object type has an associated storage pool. The storage allocated by an allocator comes from the pool; instances of Unchecked_Deallocation return storage to the pool. Several access types can share the same pool.	**§13.11**

The ability to define multiple storage pools is important in embedded systems where the amount of storage is limited. For each access type, the attribute Storage_Size §13.11(14) enables you to define the size of the pool for objects of the designated type. Package System.Storage_Pools §13.11(5–10) defines an abstract type Root_Storage_Pool that you can override to define your own storage allocation scheme. The attribute Storage_Pool §13.11(13) can then be used to assign different storage pools to different types. Note that a derived access type shares the same storage pool as its parent access type §3.4(31).

24	A default (implementation-provided) storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects.	**§13.11**

If the implementation follows this advice, then the example in Section 9.3 will not have run-time overhead, provided that String_Ptr is declared as access-to-constant.

9.7 Controlled types*

1	Three kinds of actions are fundamental to the manipulation of objects: initialization, finalization, and assignment. Every object is initialized, either explicitly or by default, after being created (for example, by an object_declaration or allocator). Every object is finalized before being destroyed (for example, by leaving a subprogram_body containing an object_declaration, or by a call to an instance of Unchecked_Deallocation). An assignment operation is used as part of assignment_statements, explicit initialization, parameter passing, and other operations.	**§7.6**
2	Default definitions for these three fundamental operations are provided by the language, but a *controlled* type gives the user additional control over parts of these operations. In particular, the user can define, for a controlled type, an Initialize procedure which is invoked immediately after the normal default initialization of a controlled object, a Finalize procedure which is invoked immediately before finalization of any of the components of a controlled object, and an Adjust procedure which is invoked as the last step of an assignment to a (nonlimited) controlled object.	

These operations are primitive operations of the abstract tagged type Controlled defined in package Ada.Finalization §7.6(4–8). You can derive from this type and override one or more of these operations. There is also a type Limited_Controlled without the Adjust operation, since limited types cannot be assigned.

Case study: priority queue with controlled type

The following version of the priority queue package demonstrates the use of controlled types. Type Node is derived from Controlled ‡15–19, and Initialize, Adjust and Finalize are overridden ‡20–22, 35–50 to print messages when they are invoked.

The queue itself is also controlled. We have made the type Queue limited by deriving it from Limited_Controlled ‡24–28. Only Finalize is overridden ‡29, 64–70 to recursively free all the nodes in the tree ‡52–62.

```
1   --                                                          -- File: PQTCT
2   -- Priority queue abstract data type implemented as a tree.
3   -- Nodes are controlled
4   --
5  with Ada.Finalization;
6  package Priority_Queue is
7    type Queue(Size: Positive) is limited private;
8    function Empty(Q: in Queue) return Boolean;
9    procedure Put(I: in Integer; Q: in out Queue);
10   procedure Get(I: out Integer; Q: in out Queue);
11   Overflow, Underflow: exception;
12 private
13   type Node;
14   type Link is access Node;
15   type Node is new Ada.Finalization.Controlled with
16     record
17       Data: Integer;
18       Left, Right: Link;
19     end record;
20   procedure Initialize(Object: in out Node);
21   procedure Adjust(Object: in out Node);
22   procedure Finalize(Object: in out Node);
23
24   type Queue(Size: Positive) is
25       new Ada.Finalization.Limited_Controlled with
26     record
27       Root: Link;
28     end record;
29   procedure Finalize(Object: in out Queue);
30 end Priority_Queue;
```

To create an aggregate of type Node ‡81, an extension aggregate must be used with a *subtype mark* for the ancestor part §4.3.2(3): since Controlled is abstract §7.6(5), no values exist which can be extended. Initialize is *not* called for the allocated object, because it is explicitly initialized with the aggregate §7.6(10). However, when the aggregate is assigned to the designated object created by the allocator, the value will be adjusted §7.6(15), in this case converting the Data component to an even number ‡43. Finalize is then called for the aggregate object which is no longer needed.

Unchecked_Deallocation is instantiated ‡33 to create a subprogram Free_Node, which is called when removing a node from the tree ‡61, 103. When the node is deallocated, Finalize is called.

```
31 with Ada.Text_IO; use Ada.Text_IO; with Ada.Unchecked_Deallocation;
32 package body Priority_Queue is
33    procedure Free_Node is new Ada.Unchecked_Deallocation(Node, Link);
34
35    procedure Initialize(Object: in out Node) is
36    begin
37       Object.Data := Object.Data + 1;
38       Put_Line("Initialize->" & Integer'Image(Object.Data));
39    end Initialize;
40
41    procedure Adjust(Object: in out Node) is
42    begin
43       Object.Data := (Object.Data / 2) * 2;
44       Put_Line("Adjust->" & Integer'Image(Object.Data));
45    end Adjust;
46
47    procedure Finalize(Object: in out Node) is
48    begin
49       Put_Line("Finalize->" & Integer'Image(Object.Data));
50    end Finalize;
51
52    procedure Free_All_Nodes(Node_Ptr: in out Link) is
53    begin
54       if Node_Ptr.Left /= null then
55          Free_All_Nodes(Node_Ptr.Left);
56       end if;
57       if Node_Ptr.Right /= null then
58          Free_All_Nodes(Node_Ptr.Right);
59       end if;
60       Put_Line("Freeing " & Integer'Image(Node_Ptr.Data));
61       Free_Node(Node_Ptr);                              -- Finalize node
62    end Free_All_Nodes;
63
64    procedure Finalize(Object: in out Queue) is
65    begin
66       Put_Line("Finalize Queue");
67       if Object.Root /= null then
68          Free_All_Nodes(Object.Root);
69       end if;
70    end Finalize;
71
```

```
72    function Empty(Q: in Queue) return Boolean is
73    begin
74      return Q.Root = null;
75    end Empty;
76
77    procedure Put(I: in Integer; Node_Ptr: in out Link) is
78    begin
79      if Node_Ptr = null then
80           Node_Ptr :=
81             new Node'(Ada.Finalization.Controlled with I, null, null);
82    --  Adjust designated object, then Finalize aggregate
83      elsif I < Node_Ptr.Data then
84        Put(I, Node_Ptr.Left);
85      else
86        Put(I, Node_Ptr.Right);
87      end if;
88    end Put;
89
90    procedure Put(I: in Integer; Q: in out Queue) is
91    begin
92      Put(I, Q.Root);
93    exception
94      when Storage_Error => raise Overflow;
95    end Put;
96
97    procedure Get(I: out Integer; Node_Ptr: in out Link) is
98      Save: Link := Node_Ptr;
99    begin
100     if Node_Ptr.Left = null then
101       I := Node_Ptr.Data;
102       Node_Ptr := Node_Ptr.Right;
103       Free_Node(Save);                          --  Finalize node
104     else
105       Get(I, Node_Ptr.Left);
106     end if;
107   end Get;
108
109   procedure Get(I: out Integer; Q: in out Queue) is
110   begin
111     if Q.Root = null then
112       raise Underflow;
113     end if;
114     Get(I, Q.Root);
115   end Get;
116 end Priority_Queue;
```

The main subprogram inserts four elements in the queue and then retrieves them:

```
117 with Priority_Queue;
118 with Ada.Text_IO; use Ada.Text_IO;
119 procedure PQTCT is
120   Q: Priority_Queue.Queue(10);
121   I: Integer;
122   Test_Data: array(Positive range <>) of Integer := (10, 5, 25, 15);
123 begin
124   for N in Test_Data'Range loop
125     Put_Line("Put-> " & Integer'Image(Test_Data(N)));
126     Priority_Queue.Put(Test_Data(N), Q);
127   end loop;
128   while not Priority_Queue.Empty(Q) loop
129     Priority_Queue.Get(I, Q);
130     Put_Line("Get-> " & Integer'Image(I));
131   end loop;
132 exception
133   when Priority_Queue.Underflow => Put_Line("Underflow from queue");
134   when Priority_Queue.Overflow => Put_Line("Overflow from queue");
135 end PQTCT;
```

The following output (reformatted) will be displayed:

```
Put-> 10          Adjust-> 10       Finalize-> 10
Put-> 5           Adjust-> 4        Finalize-> 5
Put-> 25          Adjust-> 24       Finalize-> 25
Put-> 15          Adjust-> 14       Finalize-> 15

Finalize-> 4      Get-> 4
Finalize-> 10     Get-> 10
Finalize-> 14     Get-> 14
Finalize-> 24     Get-> 24
Finalize Queue
```

Finalize is called when Q is deallocated at the completion of the subprogram, but since the queue is empty, only the message is printed. If you remove the loop that gets elements from the queue ‡128–131, the following output will be displayed instead:

```
Finalize Queue
Freeing  4        Finalize-> 4
Freeing  14       Finalize-> 14
Freeing  24       Finalize-> 24
Freeing  10       Finalize-> 10
```

9.8 Access discriminants**

The discriminants previously discussed were of discrete type and were used to constrain indices and to control variants. Discriminants can also be of a named access type or they can be anonymous access discriminants:

```
type Rec(D: access String) is limited null record;
```

Access discriminants can only be given for limited types §3.7(10). In Section 14.9, we will demonstrate the use of access discriminants to pass configuration data to a task. In this section, we show how access discriminants can be used to implement a self-referential data structure. In the absence of multiple inheritance in Ada, this technique can be used to create multiple views of a data structure; see Section 4.6.3 of the *Rationale*.

Case study: simulation with access discriminant

Event ‡17–21 is declared with a component ‡19 of type Node ‡11–14 that has an access discriminant pointing to the enclosing record (Figure 9.1).

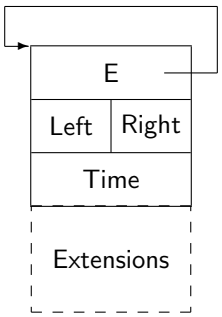

Figure 9.1: Event with embedded Node

The use of the name Event in Event'Access ‡19 refers to the *current instance* §8.6(17) of the type, not the type itself. Thus, when an event object is allocated, the discriminant is set to point to the object itself, as shown in the figure.

```
1   --                                                    -- File: ROCKETAD
2   -- Discrete event simulation of a rocket.
3   -- Self-referential nodes using access discriminants.
4   --
5  package Root_Event is
6    type Event;
7    type Event_Ptr is access Event;
8
9    type Node;
10   type Link is access all Node;
```

```
11    type Node(E: access Event'Class) is limited
12      record
13        Left, Right: Link;
14      end record;
15
16    subtype Simulation_Time is Integer range 0..10_000;
17    type Event is abstract tagged limited
18      record
19        Inner: aliased Node(Event'Access);
20        Time: Simulation_Time;
21      end record;
22
23    function Create return Event_Ptr is abstract;
24    procedure Simulate(E: in Event) is abstract;
25    function "<"(Left, Right: Event'Class) return Boolean;
26  end Root_Event;
```

Node must be limited since it has an access discriminant, and Event must be limited since it has a limited component. Appropriate modifications must be made to the simulations, because aggregates cannot be used for limited types.

The queue package links the events by linking the *contained* nodes. Given a (pointer to a) node, we can access the enclosing event using the discriminant, as shown in the right operand of "<" ‡33. Similarly, when a search of the tree returns the smallest node, the discriminant is used to return the enclosing event ‡46.

```
27  package body Event_Queue is
28
29  procedure Put(E: access Event'Class; Node_Ptr: in out Link) is
30  begin
31    if Node_Ptr = null then
32      Node_Ptr := E.Inner'Access;
33    elsif E.all < Node_Ptr.E.all then
34      Put(E, Node_Ptr.Left);
35    else
36      Put(E, Node_Ptr.Right);
37    end if;
38  end Put;
39
40    ...
41
```

```
42 function Get(Q: access Queue) return Event'Class is
43    Found: Link;
44 begin
45    Get(Q.Root, Found);
46    return Found.E.all;
47 end Get;
48
49 end Event_Queue;
```

10 Numeric Types

In most programming languages, numeric types are not portable: the type Integer may denote a 16-bit number in one implementation and a 64-bit number in another. In Ada, you can declare the intended precision of a numeric type; when the program is compiled, the implementation will choose a representation that is appropriate for the machine. For example, a type declared as an integer type with a range of 1 to 100,000 will be represented as a double word on a 16-bit machine and as a single word on a 32-bit machine.

10.1 Principles of numeric types

Universal types

The numeric types in Ada form derivation hierarchies, similar to classes of tagged types (Figure 10.1). The classes are called *universal* types §3.4.1(6–7) and the specific types at the roots of

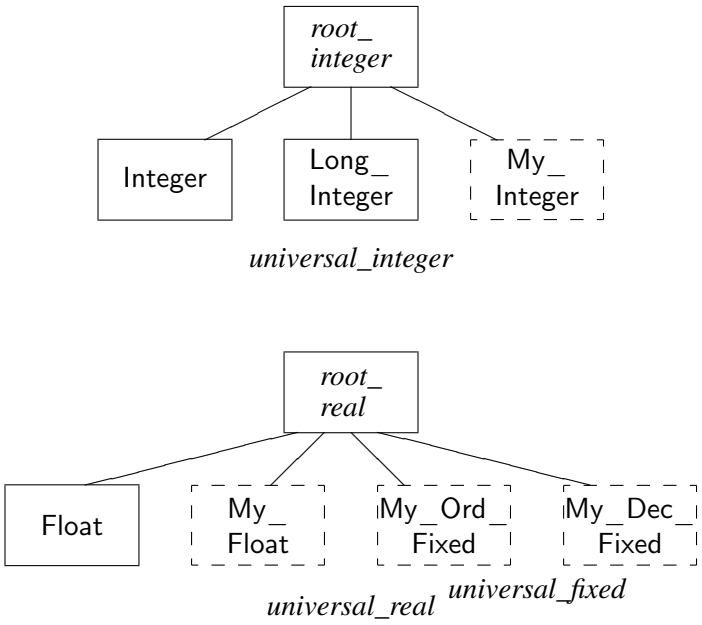

Figure 10.1: Numeric types

the derivation trees are called *root* types §3.4.1(8). These types are conceptual; you cannot explicitly declare an object or parameter to be of type *root_integer* or *universal_integer*. Some specific

types like Integer and Float are predefined, and you can declare new integer, float, ordinary fixed and decimal fixed types as indicated by the dashed boxes in the figure.

> 6 ... a value of a universal type (including an integer or real numeric_literal) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6). **§3.4.1**

> 8 An integer literal is of type *universal_integer*. A real literal is of type *universal_real*. **§4.2**

N+27 is legal for N of any integer type, because the literal 27 of type *universal_integer* is converted in context to the type of N. Note that some attributes have parameters of universal type; for example, Pos returns a value of type *universal_integer* and Val takes a parameter of type *universal_integer* §3.5.5. Character'Pos(C) for a variable C is an example of an expression that is of universal type but not static.

Type conversion

Ada does not usually allow implicit conversion between numeric types. Not only would this defeat type checking, but the rules for implicit type conversion can be very difficult, especially in the presence of overloading. For example, given that N is of type Integer, is Put(N) a call to the procedure Put with a parameter of type Integer, or is it an implicit conversion of N to Long_Integer, followed by a call to the procedure Put with a parameter of type Long_Integer?

Explicit conversion between two numeric types is always allowed §4.6(6), and is not restricted to conversion within a universal class as the analogy with derivation classes would imply.

Named numbers

> 1 A number_declaration declares a named number. **§3.3.2**
> 2 number_declaration ::=
> defining_identifier_list : **constant** := *static*_expression;
> 5 The named number denotes a value of type *universal_integer* if the type of the *static*_expression is an integer type. The named number denotes a value of type *universal_real* if the type of the *static*_expression is a real type.

The following declaration is taken from Ada.Numerics §A.5(3) !

```
Pi : constant :=
 3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
```

The value of a named number is given by a static expression:

```
Two_Pi:          constant := 2.0*Ada.Numerics.Pi;
Bits_per_Word:   constant := 16;
Values_per_Word: constant := 2**Bits_per_Word;
```

Be careful not to confuse named numbers with constant objects:

```
Two_Pi:        constant := 2.0*Ada.Numerics.Pi;
Float_Two_Pi:  constant Float := 2.0*Ada.Numerics.Pi;
```

Two_Pi is of type *universal_real* and will be converted to a specific type in each context in which it appears. Float_Two_Pi is a constant object of type Float; the initial value 2.0*Ada.Numerics.Pi of type *root_real* will be converted to type Float—losing precision—when the object is elaborated.

10.2 Integer types

1 An integer_type_definition defines an integer type; it defines either a *signed* integer type, or a *modular* integer type. ... **§3.5.4**

2 integer_type_definition ::=
signed_integer_type_definition |
modular_type_definition

3 signed_integer_type_definition ::=
range *static*_simple_expression ..
*static*_simple_expression

9 A signed_integer_type_definition defines an integer type whose base range includes at least the values of the simple_expressions and is symmetric about zero, excepting possibly an extra negative value. ...

11 There is a predefined signed integer subtype named Integer, declared in the visible part of package Standard. It is constrained to the base range of its type.

12 Integer has two predefined subtypes, declared in the visible part of package Standard:

13
```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

(Base range is discussed in Section 10.8 and can be ignored for now.)

Given the declaration:

```
type Altitude is range 0 .. 100_000;
```

the compiler can allocate a single word on a 32-bit machine and a double word on a 16-bit machine. There is no need to modify the source code when porting.

The range of predefined type Integer and its subtypes Natural and Positive is implementation-defined, and programs using them are not strictly portable. There is no reason not to use Integer for array indices because their ranges almost invariably fall within the minimum range (16-bits) of Integer §3.4.5(21), but for integer computation you should define your own types. An implementation is permitted §3.5.4(25) to provide additional predefined integer types with names like Long_Integer, Short_Integer and Long_Long_Integer, though they need not be implemented with different precisions. Needless to say, such types are not portable.

10.3 Types versus subtypes

Recall that type is a compile-time concept, while subtype is a run-time concept. Given the following declarations:

```
type Altitude is range 0 .. 100_000;
Num: Integer  := 1000;
Alt:   Altitude := 50_000;
```

Alt of type Altitude cannot be assigned to Num of type Integer because the types are different. The type conversions Num:=Integer(Alt) and Alt:=Altitude(Num) are legal.

Subtypes, however, can always be mixed; at worst, Constraint_Error will be raised. Note the difference between the two assignments to High below: in the first, the exception Constraint_Error will be raised when trying to convert 2000 of type Integer (the result of the addition) to the subtype High_Altitude, while in the second, the type conversion to Altitude will succeed but the assignment to High will raise the exception.

```
subtype Low_Altitude    is Altitude range 0 .. 35_000;
subtype High_Altitude   is Altitude
  range Low_Altitude'Last+1 .. Altitude'Last;

Num:  Integer           := 1000;
Low:  Low_Altitude      := 1000;
High: High_Altitude     := 50_000;

High := High_Altitude(Num + Integer(Low));
High := Altitude(Num + Integer(Low));
```

Newcomers to Ada have a tendency to overuse integer types, resulting in arithmetical expressions that are difficult to understand because they are filled with type conversions. If you are planning to do extensive computation with integer values, subtypes are probably more appropriate, whereas types are more often used for indices, keys or handles that are unlikely to be involved in arithmetical expressions.

10.4 Modular types

§3.5.4

4 modular_type_definition ::= **mod** *static*_expression

10 A modular_type_definition defines a modular type whose base range is from zero to one less than the given modulus. ...

19 For a modular type, if the result of the execution of a predefined operator (see 4.5) is outside the base range of the type, the result is reduced modulo the modulus of the type to a value that is within the base range of the type.

If the modulus is a power of two, the modular type is usually called an unsigned integer type.

Case study: checksum

The following program computes the checksum of an array of bytes. (Interfaces.Unsigned_8 §B.2 could be used instead of Byte.) The addition ‡16 is automatically reduced modulo 256.

```
1   --                                                          -- File: CHECK
2   -- Modular types for checksum.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure Check is
6
7     type Byte is mod 2**8;                          -- 2**8 = 256
8     for Byte'Size use 8;
9     type Byte_Array is array(Natural range <>) of Byte;
10    pragma Pack(Byte_Array);
11
12    function Checksum(A: Byte_Array) return Byte is
13       C: Byte := 0;
14    begin
15       for I in A'Range loop
16          C := C + A(I);
17       end loop;
18       return C;
19    end Checksum;
20
21    Message: Byte_Array := (134, 56, 121, 38, 206, 117);
22 begin
23    Put_Line(Byte'Image(Checksum(Message)));                -- Prints 160
24 end Check;
```

Note that Constraint_Error will never be raised when computing with a modular type, though it may be raised if you try to convert another type to the modular type. For example, if B is of type Byte, then B:=260 will raise Constraint_Error during the conversion of the literal of type *universal_integer*.

The logical operators **and**, **or**, **xor** and **not** can also be used on modular types §4.5.1(2).

Finally, note that the modulus need not be a power of two. A prime modulus can be used as the index of an array implementing a hash table.

10.5 Real types

1 Real types provide approximations to the real numbers, with relative bounds on errors for floating point types, and with absolute bounds for fixed point types. **§3.5.6**

2 real_type_definition ::=
floating_point_definition | fixed_point_definition

3 A type defined by a real_type_definition is implicitly derived from *root_real*, an anonymous predefined (specific) real type. Hence, all real types, whether floating point or fixed point, are in the derivation class rooted at *root_real*.

4 Real literals are all of the type *universal_real*, the universal type (see 3.4.1) for the class rooted at *root_real*, allowing their use with the operations of any real type. Certain multiplying operators have a result type of *universal_fixed* (see 4.5.5), the universal type for the class of fixed point types, allowing the result of the multiplication or division to be used where any specific fixed point type is expected.

1 For floating point types, the error bound is specified as a relative precision by giving the required minimum number of significant decimal digits. **§3.5.7**

1 A fixed point type is either an ordinary fixed point type, or a decimal fixed point type. The error bound of a fixed point type is specified as an absolute value, called the *delta* of the fixed point type. **§3.5.9**

Let us clarify these concepts by giving some examples. Suppose that the precision of a floating point type is six digits. A one-digit error in the least significant digit of 123456.0E9 is an absolute error of one billion, while the same error in 123456.0E1 is an absolute error of only ten. Though the absolute error varies widely, in both cases one digit represents a constant relative error of %0.0001.

Consider now a six-digit fixed point type with a delta of 0.01 and a range of 0.00 to 9999.99. A one-digit error causes an absolute error of 0.01 which is independent of the value of the number. The drawback of fixed point types is that the range of values is limited.

Floating point types are usually used in scientific computation, where we may want to compute the thrust of a rocket to an accuracy of, say, 0.01%. Fixed point types are used in financial calculations, where the absolute error must be limited to 0.01 or 0.0001 of the currency unit. The range limitation is not a problem, since even the largest government debt can be expressed in 12 or 18 digits!

Fixed point types are further divided into ordinary fixed point types whose delta is usually a power of two, and decimal fixed point types whose delta is a power of ten. The former are used for hardware interfacing and the latter for financial calculations.

10.6 Floating point types

§3.5.7

2 floating_point_definition ::=
digits *static*_expression [real_range_specification]
3 real_range_specification ::=
range *static*_simple_expression .. *static*_simple_expression
12 There is a predefined, unconstrained, floating point subtype named Float, declared in the visible part of package Standard.

A floating point type declaration declares a new type that is represented in the machine with *at least* the precision requested. The type is explicitly convertible to all other numeric types, including integer types. There is one predefined type Float, though the implementation may define others such as Long_Float. For serious computational tasks, you should avoid the non-portable predefined types and define your own.

Case study: Euler's method

The following program computes a solution to an elementary differential equation using Euler's method. It solves an equation:

$$\frac{dy}{dx} = f(y)$$

by dividing the range into steps and then starting from an initial value, computing each successive point by extending the tangent of the previous one. The example used is $dy/dx = y$ on the interval 0.0 to 1.0; given an initial condition of 1.0, the answer is $y = e^x$.

The procedure Euler is generic in the floating point type, an array type for returning the result and an access type for passing the function. The computation in the body of the generic procedure ‡13–20 uses attributes to get the indices of Result, whose type is the unconstrained generic formal array type. F is implicitly dereferenced it is called with a parameter ‡18.

-- File: DIFF

```
1   --
2   -- Solving a differential equation.
3   -- Demonstrates generic floating point type.
4   --
5  generic
6     type Float_Type is digits <>;
7     type Vector is array(Integer range <>) of Float_Type;
8     type Function_Ptr is
9           access function (X: Float_Type) return Float_Type;
10 procedure Euler(
11    F: in Function_Ptr; Init, H: in Float_Type; Result: out Vector);
12
```

```
13 procedure Euler(
14   F: in Function_Ptr; Init, H: in Float_Type; Result: out Vector) is
15 begin
16   Result(Result'First) := Init;
17   for N in Result'First+1..Result'Last loop
18     Result(N) := Result(N-1) + H * F(Result(N-1));
19   end loop;
20 end Euler;
```

The procedure is tested with a 6-digit floating point type Real ‡25. After declaring appropriate types for the array and function pointer ‡26-27, the generic procedure is instantiated ‡29, and can then be called ‡42 for any function such as Ident ‡31–34 that matches the profile of the access type.

```
21 with Ada.Text_IO;
22 with Euler;
23 procedure Diff is
24
25   type Real is digits 6;
26   type Vector is array(Integer range <>) of Real;
27   type Ptr is access function (X: Real) return Real;
28
29   procedure Solve is new Euler(Real, Vector, Ptr);
30
31   function Ident(X: Real) return Real is
32   begin
33     return X;
34   end Ident;
35
36   package Real_IO is new Ada.Text_IO.Float_IO(Real);
37   use Real_IO;
38
39   Answer: Vector(1..21);
40
41 begin
42   Solve(Ident'Access, 1.0, 0.05, Answer);
43   for N in Answer'Range loop
44     Put(0.05 * Real(N-1), Exp => 0);
45     Put( Answer(N), Exp => 0);
46     Ada.Text_IO.New_Line;
47   end loop;
48 end Diff;
```

Generic units with formal parameters of floating point type can be used to create numerical libraries. Not only can you change the precision of the type without otherwise modifying the program, but you can also instantiate a library for multiple precisions and use the instantiations in the

same program.

Elementary functions such as the trigonometric functions are predefined in the generic package Ada.Numerics.Generic_Elementary_Functions §A.5.1, which can be instantiated with any floating point type. Ada.Numerics.Elementary_Functions is a predefined instantiation for Float. A package generating random numbers of type Float is defined in §A.5.2.[1]

Annex §G 'Numerics' has two sections. The first defines packages for complex numbers, including elementary functions and IO. The second 'Numeric Performance Requirements' gives a detailed model of computation with real types. The annex is briefly discussed in Section 10.8.

10.7 Fixed point types

Decimal fixed point types

§3.5.9

2 fixed_point_definition ::=
  ordinary_fixed_point_definition |
  decimal_fixed_point_definition
4 decimal_fixed_point_definition ::=
  delta *static*_expression digits *static*_expression
    [real_range_specification]
8 The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. ...
9 For a decimal fixed point type, the *small* equals the *delta*; the *delta* shall be a power of 10. ...

The values of the following type are in the range ±9,999,999.99:

```
type Money is delta 0.01 digits 9;
```

There is a limit on the precision that can be specified: eventually, the implementation will run out of digits to store values of the requested range. In general, the smaller the delta, the smaller the range that can be specified.

Decimal fixed point types can be implemented using *binary coded decimal (BCD)*, which is supported in hardware on some computers. Four bits (a 'nibble') are sufficient to encode the ten values of a single digit, so two digits can be stored in a single byte. Alternatively, the multiple of the delta can be stored as a normal integer and the value scaled with the delta as needed.

Multiplication and division are problematical for fixed point types. Suppose that M1 and M2 are two variables of type Money that both contain the value 0.25. What is the type of M1*M2 which equals 0.0625? This answer is that the type must be given by the context; if the expression M1*M2 is assigned to another variable of type Money, the value will be truncated to 0.06. The rules are given in §4.5.5 and will be demonstrated in the next case study.

[1] There is also a generic package for random numbers that can be instantiated with a *discrete* type; we used this package in the rocket simulation.

Annex F Information Systems

Historically, there has been a large gap between the world of scientific and systems programming, and business programming, where COBOL has been the language of choice. Of course, most of the requirements for a language for business programming are not different from those of other fields: reliability, efficiency, system interfaces and support for software engineering. The primary extension needed is in the area of decimal types.

The decimal fixed point types in Ada provide this basic functionality, though an implementation need not support such types §3.5.9(21). Implementations conforming to Annex §F 'Information Systems' are required to implement decimal fixed point types, as well as three packages: Ada.Decimal, which contains named numbers specifying properties of the decimal types and a generic procedure for arbitrary decimal fixed point division, and Ada.Text_IO.Editing and Ada.Wide_Text_IO.Editing for formatted input–output.

Case study: currency conversion

The following program reads a currency and an amount, and writes the equivalent value in the other eight currencies. A table copied from my daily newspaper gives the conversion rates—to four digits after the decimal point—between the currencies.

Currencies ‡13 is an enumeration type used internally; Signs ‡16–25 is an array of bounded strings for display of the currency symbols. Package Ada.Characters.Latin_1 §A.3.3 contains characters for the British Pound and the Japanese Yen. My computer cannot display them, but the program is still portable.

```
1   --                                                    -- File: CONVERT
2   -- Currency conversion using decimal fixed types.
3   --
4  with Ada.Strings.Bounded;
5  package BS is new Ada.Strings.Bounded.Generic_Bounded_Length(10);
6
7  with BS;
8  with Ada.Characters.Latin_1;
9  with Ada.Text_IO.Editing;
10 use Ada.Text_IO;
11 procedure Convert is
12
13   type Currencies is (US, UK, DM, Y, SF, FF, fl, LIT, BF);
14   package Currency_IO is new Enumeration_IO(Currencies);
15
16   Signs: constant array(Currencies) of BS.Bounded_String := (
17     BS.To_Bounded_String("$"),
18     BS.To_Bounded_String((1=>Ada.Characters.Latin_1.Pound_Sign)),
19     BS.To_Bounded_String("DM"),
20     BS.To_Bounded_String((1=>Ada.Characters.Latin_1.Yen_Sign)),
21     BS.To_Bounded_String("SF"),
```

```
22      BS.To_Bounded_String("FF"),
23      BS.To_Bounded_String("fl"),
24      BS.To_Bounded_String("LIT"),
25      BS.To_Bounded_String("BF"));
```

Money ‡26 is a decimal fixed point type with two digits after the decimal point. To read a currency and an amount, Ada.Text_IO.Enumeration_IO §A.10.10 is instantiated ‡14 with type Currencies, and Ada.Text_IO.Decimal_IO §A.10.9 is instantiated ‡27 with type Money.

```
26    type Money is delta 0.01 digits 9;
27    package Money_IO is new Decimal_IO(Money);
28
```

Conversion ‡31–40 is a table of the exchange rates; the component type Rates ‡29 has four digits. Since the denominations of currencies vary, a table Factors ‡42–44 provides additional scaling; for example, one US dollar is worth 1.8399 *thousand* Italian Lira, or conversely, one *thousand* Lira is worth $0.5435.

```
29    type Rates is delta 0.0001 digits 6;
30
31    Conversion: constant array(Currencies, Currencies) of Rates :=
32  (( 1.0, 0.6265, 1.8807, 1.1877, 1.5315, 6.3741, 2.1191, 1.8399, 3.8840),
33  (1.5961, 1.0, 3.0018, 1.8957, 2.4443, 10.1304, 3.3823, 2.9367, 6.1991),
34  (0.5317, 0.3331, 1.0, 0.6315, 0.8143, 3.3748, 1.1268, 0.9783, 2.0652),
35  (0.8419, 0.5275, 1.5835, 1.0, 1.2894, 5.3439, 1.7842, 1.5491, 3.2701),
36  (0.6530, 0.4091, 1.2280, 0.7755, 1.0, 4.1444, 1.3837, 1.2014, 2.5361),
37  (0.1576, 0.0987, 0.2963, 0.1871, 0.2413, 1.0, 0.3339, 0.2899, 0.6119),
38  (0.4719, 0.2957, 0.8875, 0.5605, 0.7227, 2.9951, 1.0, 0.8682, 1.8328),
39  (0.5435, 0.3405, 1.0222, 0.6455, 0.8324, 3.4496, 1.1517, 1.0, 2.1109),
40  (0.2575, 0.1613, 0.4824, 0.3058, 0.3943, 1.6342, 0.5456, 0.4737, 1.0));
41
42    Factors: constant array(Currencies) of Integer :=
43      (US => 1, UK => 1, DM => 1, Y => 100, SF => 1, FF => 1,
44      FL => 1, LIT => 1000, BF => 10);
```

Function Get_Value ‡45–51 performs the conversion. The expression we want to compute is $(M/F1) \cdot R \cdot F2$, where M is the amount of money, R is the conversion rate, and $F1$ and $F2$ are the factors for the original currency and the new one.

```
45    function Get_Value(M: Money; From, To: Currencies) return Money is
46      type Intermediate is delta 0.000001 digits 13;
47    begin
48      return Money(
49        (Intermediate(M) / Factors(From)) *
50        (Intermediate(Conversion(From, To)) * Factors(To)) );
51    end Get_Value;
```

The operators we have at our disposal are:

13 The following multiplication and division operators, with an operand of the predefined type Integer, are predefined for every specific fixed point type T: **§4.5.5**

14
```
function "*"(Left : T; Right : Integer) return T
function "*"(Left : Integer; Right : T) return T
function "/"(Left : T; Right : Integer) return T
```

18 Multiplication and division between any two fixed point types are provided by the following two predefined operators: **§4.5.5**

19
```
function "*"(Left, Right : universal_fixed)
   return universal_fixed
function "/"(Left, Right : universal_fixed)
   return universal_fixed
```

The division $M/F1$ has a fixed point dividend and an integer divisor and returns a fixed point quotient of the *same* type as the dividend. If we divide 999.99 Italian Lira by 1000, the result will be 0.99 after truncation to two fractional digits. To maintain precision, we have declared a new type Intermediate ‡46 with six fractional digits. The amount of money is converted to this type before division by the factor.

The expression can be computed in one of two orders: $((M/F1) \cdot R) \cdot F2$ or $(M/F1) \cdot (R \cdot F2)$. We have chosen the second order: the result of $R \cdot F2$ is also converted to a value of type Intermediate and the final multiplication between the two fixed point values returns a result of type *universal_fixed*, which is converted to the type Money.

Writing the converted amounts is done using Ada.Text_IO.Editing.Decimal_Output §F.3.3 instantiated with type Money ‡52. This package supplies a private type Picture that is used for format control. A value of type Picture is created by calling To_Picture with a string of format control characters ‡53.

```
52  package Edit is new Editing.Decimal_Output(Money);
53  Money_Picture: Editing.Picture := Editing.To_Picture("###*_***_**9.99");
```

The syntax and semantics of formatting are specified in §F.3.1 and §F.3.2; you can also consult a COBOL textbook, since picture formatting is almost identical in the two languages. The picture we use is "###*_***_**9.99", where the meaning of each character is as follows:

- '9' - Decimal digit.
- '.' - Radix mark (‘decimal point’).
- '_' - Separator character.
- '*' - Fill character.
- '#' - Currency string.

The amount 4156.34 French Francs will be written as ␣FF***4,156.34.

Fill characters are used instead of blanks to prevent forgery. The picture characters are fixed in the language, but the displayed characters are parameters of the procedure Put and can be changed for localization. Alternatively, new default values can be specified when Decimal_IO is instantiated.[2]

The main subprogram performs the interactive dialog and calls Get_Value ‡74 for each target currency. The dialog is in a loop containing a block ‡56–86 with exception handlers ‡81–85. Entering an invalid code for a currency will raise Data_Error §A.13(6). If the value is too large, either Constraint_Error will be raised during the computation or Picture_Error §F.3.3(9) will be raised during formatting. After the exception is handled, the block is left and the next iteration of loop gives the user a chance to fix the error. If End_Error §A.13(12) occurs, the loop is exited.

```
54 begin
55   loop
56     declare
57       Source: Currencies;
58       Amount: Money;
59     begin
60       Put("Currency (");
61       for C in Currencies loop
62         Currency_IO.Put(C);
63         if C /= Currencies'Last then Put(", "); end if;
64       end loop;
65       Put(") and amount: ");
66       Currency_IO.Get(Source);
67       Money_IO.Get(Amount);
68       Skip_Line;
69       Edit.Put(Amount, Money_Picture, BS.To_String(Signs(Source)));
70       Put_Line(" is worth ");
71       for Target in Currencies loop
72         if Source /= Target then
73           Edit.Put(
74             Get_Value(Amount, Source, Target),
75             Money_Picture, BS.To_String(Signs(Target)));
76           New_Line;
77         end if;
78       end loop;
79       New_Line;
80     exception
81       when Data_Error =>
82         Skip_Line; Put_Line("Illegal input");
83       when Editing.Picture_Error | Constraint_Error =>
84         Put_Line("Amount too large");
```

[2]These localization features are not found in COBOL. Furthermore, COBOL does not have the '#' currency string, which allows a fixed-width currency field unlike '$'.

```
85          when End_Error => exit;
86       end;
87    end loop;
88 end Convert;
```

Ordinary fixed point types

> 8 The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. For a type defined by an ordinary_fixed_point_definition (an *ordinary* fixed point type), the *small* may be specified by an attribute_definition_clause (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*. **§3.5.9**

Ordinary fixed point types are similar to decimal fixed point types, except that their *small* can be any number. The small is usually a power of two so that values of the type can be exactly represented in binary. Ordinary fixed point types are extremely useful for programming embedded systems for two reasons: (a) small computers may not have floating point hardware, and (b) external peripherals transfer binary numbers that represent physical quantities.

The following program shows how a 16-bit word received from a sensor ‡7 can be easily converted to an ordinary fixed point value. The assumed representation is that the least significant bit represents 1/16 of a degree of temperature ‡10–12. First a fixed point type Temperature is declared, followed by representation attributes that specify that 16 bits should be used for objects of the type and that the least significant (binary) digit represent 1/16. After the conversion ‡14–15,17, fixed point operators could be used for further computation, though we just print the value.

```
1   --                                                          -- File: TEMP
2   -- Hardware interface using ordinary fixed point types.
3   --
4  with Interfaces; with Unchecked_Conversion;
5  with Ada.Text_IO; use Ada.Text_IO;
6  procedure Temp is
7    Sensor: Interfaces.Integer_16 := 2#0_001_0001_0001_1100#;
8    -- 256 + 16 + 1 + 1/2 + 1/4 = 273.75
9
10   type Temperatures is delta 2.0**(-4) range -2048.0..2048.0;
11   for Temperatures'Size use 16;
12   for Temperatures'Small use 2.0**(-4);
13
14   function To_Temp is new Unchecked_Conversion(
15     Source => Interfaces.Integer_16, Target => Temperatures);
16 begin
17   Put_Line(Temperatures'Image(To_Temp(Sensor)));
18 end Temp;
```

10.8 Advanced concepts*

Base range

Clearly, a computer will not support different hardware formats for each floating point type such as **digits** 6, **digits** 7, **digits** 8. The implementation will represent values of each type in a hardware format that can contain at least the values of the type, but possibly more. Similarly, an enumeration or integer type will be stored in a hardware format whose range of values may be significantly more than the minimum required.

6	The *base range* of a scalar type is the range of finite values of the type that can be represented in every unconstrained object of the type; it is also the range supported at a minimum for intermediate values during the evaluation of expressions involving predefined operators of the type.	**§3.5**
15	S'Base—S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type.	

Of course the base range is implementation-dependent, so S'Base should not normally be used in writing Ada programs. Base ranges are important in defining the language and in optimizing machine code. Predefined arithmetical operators are defined §A.1(15–18) for the unconstrained type Integer'Base, not for the constrained type Integer. Range checks never apply to an unconstrained type. Consider:

```
N1, N2: Integer := 15_000;
N3: Integer := (N1 + N2) / 2;
```

Suppose the implementation defines type Integer to have a 16-bit range, but Integer'Base is defined to have a 32-bit range because all computation is performed in 32-bit registers. If "+" were defined on parameters of type Integer, the addition would be required to raise Constraint_Error. However, since the addition is performed on the unconstrained type Integer'Base, no range check need be done until the assignment to N3 and Constraint_Error will not be raised. You can also declare variables of the unconstrained type to explicitly hold intermediate values of a computation.

Note that predefined Integer is constrained §3.5.4(11) to its base range while Float is unconstrained §3.5.7(12).

Complex numbers

Annex §G 'Numerics' defines support for complex arithmetic including elementary functions and IO.

§G.1.1

```
2  generic
     type Real is digits <>;
   package Ada.Numerics.Generic_Complex_Types is
     pragma Pure(Generic_Complex_Types);
3    type Complex is
       record
         Re, Im : Real'Base;
       end record;
4    type Imaginary is private;
5    i : constant Imaginary;
     j : constant Imaginary;
22 private
     type Imaginary is new Real'Base;
23   i : constant Imaginary := 1.0;
     j : constant Imaginary := 1.0;
24 end Ada.Numerics.Generic_Complex_Types;
```

The reason Imaginary is declared as a separate type is to allow expressions of the form R1 + I1*i. Since the type Complex is visible, you can create values of the type using ordinary aggregates such as (5.0, 6.3). The arithmetical operators are overloaded for all combinations of parameters of types Real'Base, Complex and Imaginary.

Ada.Numerics.Complex_Types is a predefined instantiation of the generic package for predefined Float.

§G.1.2 declares a generic package for complex elementary functions and §G.1.3 declares a generic package for IO. These packages have a single generic formal parameter, which is the package Ada.Numerics.Generic_Complex_Types; the actual parameter can be any instantiation of the package obtained by supplying a floating point type for the formal parameter Real.

Case study: complex vectors

Here is the outline of a package for complex vectors that is generic in the complex type package ‡8–9 and the complex elementary function package ‡10–11. For simplicity, the package contains only one subprogram.

```
1  --                                                    -- File: COMPLEX
2  -- Complex vectors using generic package parameters.
3  --
4  with Ada.Numerics.Generic_Complex_Types;
5  with Ada.Numerics.Generic_Complex_Elementary_Functions;
```

```
 6 generic
 7    use Ada.Numerics;
 8    with package Complex_Types is
 9       new Generic_Complex_Types (<>);
10    with package Complex_Functions is
11       new Generic_Complex_Elementary_Functions(Complex_Types);
12 package Generic_Complex_Vectors is
13    type Vector(<>) is private;
14    function Distance(Left, Right: Vector) return Complex_Types.Real'Base;
15 private
16    type Vector is array(Integer range <>) of Complex_Types.Complex;
17 end Generic_Complex_Vectors;
18
19 package body Generic_Complex_Vectors is
20    use Complex_Types;
21    function Distance(Left, Right: Vector) return Real is
22       Sum: Complex := Compose_From_Cartesian(0.0);
23    begin
24       for N in Left'Range loop
25          Sum := Sum + Left(N) * Right(N);
26       end loop;
27       return abs(Complex_Functions.Sqrt(Sum));
28    end Distance;
29 end Generic_Complex_Vectors;
```

The actual parameters of the instantiation ‡36–37 Complex_Vectors are the predefined packages for Float, which are considered to be equivalent to instantiations of the generic packages.

```
30 with Ada.Numerics.Complex_Types;
31 with Ada.Numerics.Complex_Elementary_Functions;
32 with Generic_Complex_Vectors;
33 package Complex_Vectors is new
34    Generic_Complex_Vectors(
35       Ada.Numerics.Complex_Types,
36       Ada.Numerics.Complex_Elementary_Functions);
```

Alternatively, if we have defined our own floating point type, we can instantiate in sequence the generic packages for complex types, complex elementary functions and complex vectors.

```
37 package Signals is
38    type Real is digits 12;
39 end Signals;
40
41 with Signals;
42 with Ada.Numerics.Generic_Complex_Types;
43 package Signals_Complex is
44    new Ada.Numerics.Generic_Complex_Types(Signals.Real);
```

```
45
46 with Signals;
47 with Signals_Complex;
48 with Ada.Numerics.Generic_Complex_Elementary_Functions;
49 package Signals_Complex_EF is new
50    Ada.Numerics.Generic_Complex_Elementary_Functions(Signals_Complex);
51
52 with Signals_Complex;
53 with Signals_Complex_EF;
54 with Generic_Complex_Vectors;
55 package Signals_Complex_Vectors is new
56    Generic_Complex_Vectors(Signals_Complex, Signals_Complex_EF);
```

In Section 7.10 we noted that a generic package can have a generic child. The following package extends the complex vector abstraction by creating an abstraction of a pair of vectors that is implemented ‡63–66 using the full view of the type Vector that is visible to a child package. Signals_Complex_Vectors.Pair is created by instantiating ‡71–72 the generic child Generic_Pair of the *instance* Signals_Complex_Vectors.

```
57 generic
58 package Generic_Complex_Vectors.Generic_Pair is
59    type Pair is private;
60 private
61    subtype Max is Integer range 0..100;
62    type Pair(Size: Max := 10) is
63       record
64          First, Second: Vector(1..Size);
65       end record;
66 end Generic_Complex_Vectors.Generic_Pair;
67
68 with Signals_Complex_Vectors;
69 with Generic_Complex_Vectors.Generic_Pair;
70 package Signals_Complex_Vectors.Pair is
71    new Signals_Complex_Vectors.Generic_Pair;
```

Preference for root types**

What is the type of N in the following loop statement?

```
for N in 1..100 loop
```

Since the literals are of type *universal_integer*, the type should be ambiguous, because there is no context that can be used to convert the range to a specific type. The following rules are used to resolve the ambiguity:

29	There is a *preference* for the primitive operators (and ranges) of the root numeric types *root_integer* and *root_real*.	**§8.6**

> 18 If the type of the range resolves to *root_integer*, then the discrete_subtype_definition defines a subtype of the predefined type Integer with bounds given by a conversion to Integer of the bounds of the range; **§3.6**

1..100 is resolved by preference to *root_integer* and defines a subtype of type Integer.

For another example, consider the following program, which prints "OK" because of the preference for root numeric types, even though "<" could refer to the 'strange' operator that we have defined for predefined Integer!

```
1 with Ada.Text_IO; use Ada.Text_IO;                    -- File: PREF
2 procedure Pref is
3   function "<"(Left, Right: Integer) return Boolean is
4   begin
5     return Left >= Right;
6   end "<";
7 begin
8   if 5 < 4 then Put("Strange"); else Put("OK"); end if;
9 end Pref;
```

Model numbers**

> 8 The set of values for a floating point type is the (infinite) set of rational numbers. The *machine numbers* of a floating point type are the values of the type that can be represented exactly in every unconstrained variable of the type. ... **§3.5.7**

The accuracy of floating point computation is defined in Annex §G, using an idealization of machine numbers called *model numbers*. Some values of floating point type, such as $0.5_{10} = 0.1_2$, are equal to model numbers; others, such as $0.2_{10} = 0.00110011\ldots_2$ that do not equal model numbers are represented by the *model interval* between the model numbers closest to the value.[3] An arithmetical operation between two model intervals yields another model interval. These concepts are used to define the accuracy of computation with real types.

An implementation may find some accuracy requirements difficult to achieve efficiently. In addition to the required *strict mode* of computation that fully conforms with the standard, an implementation may offer a *relaxed mode* of computation §G.2(1–3).

If you are interested in numeric computation, you will want to read Annex §G of the *Rationale*, which explains both complex arithmetic and model numbers in detail.

[3]This sentence assumes a binary machine; on a decimal machine, both values would be model numbers.

11 Input–output

11.1 Libraries for input–output

Sections §A.6 through §A.14 of Annex §A describe the input–output facilities of Ada. So far we have used Ada.Text_IO §A.10 for input and output of characters and numerals. There is a package Ada.Wide_Text_IO §A.11 of identical functionality for Wide_Character. Subprograms for input–output of characters and strings are declared directly within Ada.Text_IO. For other scalar types, a generic package must be instantiated. These packages are Integer_IO, Modular_IO, Float_IO, Fixed_IO, Decimal_IO and Enumeration_IO. If you are going to use these packages frequently in your system, it is more efficient at compile-time if you instantiate them once as library packages:

```
package P is
  type Countries is (US, UK, France, Germany, Japan, Korea);
end P;

with P; with Ada.Text_IO;
package Countries_IO is new Ada.Text_IO.Enumeration_IO(P.Countries);
```

For each predefined integer type §A.10.8(20–22) and predefined floating point type §A.10.9(32–34), nongeneric packages are predefined which are equivalent to instantiations with these types. For example, Ada.Integer_Text_IO is predefined for type Integer.

Here is an overview of some of the features of Ada.Text_IO:

- Most subprograms are overloaded: one version with an explicit file parameter of type File_Type §A.10.1(3) and the other using a default file. The default files are usually the keyboard and screen, but this can be changed using the subprograms described in §A.10.3.
- Text files are considered to be composed of logical *pages*, *lines* and *columns* §A.10(7–11). Subprograms that query and modify the logical structure of the file are described in §A.10.4 and §A.10.5. For example, function End_Of_File §A.10.5(24) checks if the end of an input file has been reached, and procedure New_Page §A.10.5(15) causes a 'page terminator' (however that is implemented on your system) to be written to an output file. Input–output may be done to internal strings as well as to external files §A.10(3).
- While many languages use a special syntax to specify the format of the text (field width, number base, and so on), Ada uses ordinary parameters to specify format information. A simple subprogram call like Put(N) uses default values for format parameters. Named parameter association can be used to change just one of these values: Put(N, Base=>16).

- There is no special syntax (as in Pascal or C) that allows input or output of more than one value in a single subprogram call.

'Binary' input–output can be done using Ada.Sequential_IO §A.8.1, in which the values are accessed in sequential order, and Ada.Direct_IO §A.8.4, in which you can access a value at any position in the file. Both packages take a generic private parameter which specifies the type of the file elements.

Other sections of Annex §A pertaining to input–output are:

- §A.7 defines the terminology used for files and §A.8.2 describes subprograms that perform file management such as opening and closing a file.
- Package Ada.Storage_IO §A.9 implements input–output to and from a memory buffer rather than an external file. It can be used to create your own input–output package, since it translates from an internal representation which includes dope vectors and tags to a flat buffer of storage elements.
- Exceptions that may be raised upon input–output are declared in Ada.IO_Exceptions §A.13. Other packages rename the exceptions so you won't have to 'with' this package directly.
- §A.14 discusses what happens if an external file is associated with more than one internal file object.
- Stream input–output §A.12 is discussed in Section 11.3.

11.2 Package Exceptions*

Package Ada.Exceptions §11.4.1 declares two types: Exception_ID and Exception_Occurrence, each of which can be converted to and from a string. There is a value of type Exception_ID for each distinct exception—both predefined and declared. E'Identity returns the identity associated with exception E. Raising an exception creates a value of type Exception_Occurrence. Obviously, many occurrences of a single exception may be created.

Ada.Exceptions gives you additional control over exception handling. Procedure Raise_Exception associates a string with a specific occurrence. The subprograms Save_Occurrence and Reraise_Occurrence can be used to create data structures of exception occurrences.

Case study: saving exceptions

In the following program, three subprograms P1, P2, P3 are called, but we save all the exception occurrences in a priority queue (surprise!). The queue elements are of type Exception_Record ‡9–13, which includes a 'priority' and an access to the exception occurrence §11.4.1(3). Function "<" ‡15–18 compares exceptions, and is used by default in the instantiation ‡20 of the generic priority queue. When the execution of a procedure raises an exception ‡29,36,52, the exception occurrence is placed on the queue ‡31,38,45, and the procedure terminates.

```
1   --                                                          -- File: EXCEP
2   -- Saving and reraising exception occurrences.
3   --
4  with Priority_Queue;
5  with Ada.Text_IO; use Ada.Text_IO;
6  with Ada.Exceptions; use Ada.Exceptions;
7  procedure Excep is
8
9     type Exception_Record is
10       record
11          Priority: Positive;
12          Occurrence: Exception_Occurrence_Access;
13       end record;
14
15    function "<"(Left, Right: Exception_Record) return Boolean is
16    begin
17       return Left.Priority < Right.Priority;
18    end "<";
19
20    package Exception_Queue is new Priority_Queue(Exception_Record);
21    use Exception_Queue;
22
23    Q: aliased Queue;
24
25    Ex1, Ex2, Ex3, Ex4: exception;
26
27    procedure P1 is
28    begin
29       Raise_Exception(Ex1'Identity, "P1 " & Integer'Image(13));
30    exception
31       when E: others => Put( (13, Save_Occurrence(E)), Q);
32    end P1;
33
34    procedure P2 is
35    begin
36       Raise_Exception(Ex1'Identity, "P2 " & Integer'Image(6));
37    exception
38       when E: others => Put( (6, Save_Occurrence(E)), Q);
39    end P2;
40
41    procedure P3 is
42    begin
43       Raise_Exception(Ex4'Identity, "P3 " & Integer'Image(8));
44    exception
45       when E: others => Put( (8, Save_Occurrence(E)), Q);
46    end P3;
```

Note that the message associated with each occurrence need not be static as shown, but could be computed dynamically. Upon completion of the execution of the subprograms ‡49, the exception occurrences are retrieved in order of priority and reraised ‡51.

```
47 begin
48    P1; P2; P3;
49    while not Empty(Q) loop
50       begin
51          Reraise_Occurrence(Get(Q'Access).Occurrence.all);
52       exception
53          when E: others => Put_Line(Exception_Information(E));
54       end;
55    end loop;
56 end Excep;
```

Each reraised exception is handled ‡53 within the block of the main program by printing the implementation-defined string §11.4.1(13) associated with the occurrence:

```
EXCEP.EX1    P2      6
EXCEP.EX4    P3      8
EXCEP.EX1    P1     13
```

11.3 Streams**

Ada.Sequential_IO performs input–output on values of a single type. *Streams* are used for input–output of values of more than one type to a single file. The basic idea is that a value of any type is output as a sequence of bytes which will be reconstructed into a value of the same type upon input. A stream file is much more portable than a binary file: while the encoding of elementary types is implementation-dependent, there is a canonical order defined for encoding composite types §13.13.2(9). Streams are used not only to create files, but also to pass data between the partitions of a distributed system (Section 16.5).

Package Ada.Streams §13.13.1 declares type Root_Stream_Type as an abstract tagged type from which all streams are derived. A stream is composed of a sequence of values of the modular type Stream_Element. Figure 11.1 shows how streams work. Two operations are involved in writing to a stream: the attribute S'Write transforms values of a subtype S into stream elements; then the subprogram Write writes the elements onto the stream. Reading does these steps in the opposite direction.

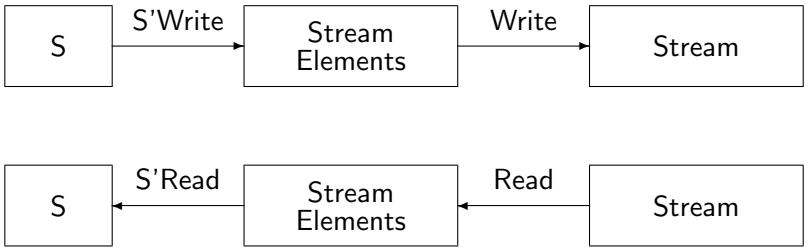

Figure 11.1: Streams

1 The Write, Read, Output, and Input attributes convert values to a stream of elements **§13.13.2**
and reconstruct values from a stream.
2 For every subtype S of a specific type *T*, the following attributes are defined.
3 S'Write—S'Write denotes a procedure with the following specification:
4 **procedure** S'Write(
Stream : **access** Ada.Streams.Root_Stream_Type'Class;
Item : **in** *T*)
5 S'Write writes the value of *Item* to *Stream*.
6 S'Read—S'Read denotes a procedure with the following specification:
7 **procedure** S'Read(
Stream : **access** Ada.Streams.Root_Stream_Type'Class;
Item : **out** *T*)
8 S'Read reads the value of *Item* from *Stream*.

The subprograms Write and Read are not explicitly called; instead, the attributes call them automatically §13.13.1(1). You can override the abstract subprograms Write and Read §13.13.1(5–6) when you define a stream for a new type, and you can supply an attribute definition clause to specify the attributes §13.13.2(36).

The following case study shows how you can use streams without a detailed knowledge of package Ada.Streams. To create a stream file, you do not need to explicitly declare a stream. Instead, you can declare a file of type File_Type from package Ada.Streams.Stream_IO §A.12.1. This package declares a function that returns an access to the stream associated with the file; this access is then used as a parameter of the Read and Write attributes.

§A.12.1

```
 4  type Stream_Access is access all Root_Stream_Type'Class;
13  function Stream (File : in File_Type) return Stream_Access;
       -- Return stream access
       -- for use with T'Input and T'Output
```

29 The Stream function returns a Stream_Access result from a File_Type object, thus **§A.12.1**
allowing the stream-oriented attributes Read, Write, Input, and Output to be used
on the same file for multiple types.

Case study: simulation with streams

This version of the discrete event simulation creates and writes events on a file. The file can then be repeatedly read to rerun the same simulation scenario. (The attributes Input and Output are used instead of Read and Write for reasons explained below.) File management of stream files is no different from that of any other file. The essential difference is the declaration of the variable S ‡12 of type Stream_Access and the assignment to S of the stream obtained from the file ‡15,28. S is then used as a parameter of the attributes Event'Class'Input ‡30 and Event'Class'Output ‡18–23.

```
1   --                                                    -- File: ROCKETST
2   -- Discrete event simulation of a rocket.
3   -- Write events to a stream file and read back.
4   --
5  with Event_Queue;
6  with Root_Event.Engine, Root_Event.Telemetry, Root_Event.Steering;
7  use Root_Event;
8  with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;
9  procedure RocketST is
10   Q: Event_Queue.Queue_Ptr := new Event_Queue.Queue;
11   Event_File: File_Type;
12   S: Stream_Access;
13 begin
14   Create(Event_File, Name=>"Event.Str");
15   S := Stream(Event_File);
16
17   for I in 1..15 loop
18     Event'Class'Output(S,
19       Event'Class(Engine.Main_Engine_Event'(Engine.Create)));
20     Event'Class'Output(S,
21       Event'Class(Engine.Aux_Engine_Event'(Engine.Create)));
22     Event'Class'Output(S, Event'Class(Telemetry.Create));
23     Event'Class'Output(S, Event'Class(Steering.Create));
24   end loop;
25   Close(Event_File);
26
27   Open(Event_File, In_File, Name=>"Event.Str");
28   S := Stream(Event_File);
29   for I in 1..45 loop
30     Event_Queue.Put(Event'Class'Input(S), Q.all);
31   end loop;
32   Close(Event_File);
33
34   while not Event_Queue.Empty(Q.all) loop
35     Root_Event.Simulate(Event_Queue.Get(Q));
36   end loop;
37 end RocketST;
```

The following rule describes how values are transformed into storage elements by Read and Write:

> 9 For elementary types, the representation in terms of stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in a canonical order. The canonical order of components is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if T is an array type. If T is a discriminated type, discriminants are included only if they have defaults. If T is a tagged type, the tag is not included. **§13.13.2**

Thus given:

```
subtype Line is String(1..120);
```

Line'Write will write 120 stream elements (probably bytes), and you must be careful to read it only with the corresponding subprogram Line'Read, which will read the same 120 stream elements. Consider now the following procedure:

```
procedure Write_String(S: in Stream_Access; Str: in String) is
begin
  String'Write(S, Str);
end Write_String;
```

Each call to the procedure will write the stream elements that represent the current value of Str. You would need to explicitly write additional information on the stream in order to read it correctly. This can be done automatically by using the attribute Output §13.13.2(19–21), which is the same as Write except that array bounds and discriminants, if any, are automatically written to the stream. Similarly, Input §13.13.2(22–24) is like Read, except that it can use this information to determine how much data to read and how to arrange it in an object.

For class-wide types, S'Class'Write and S'Class'Read §13.13.2(10–16) dispatch to the Write and Read attributes according to the specific type of the actual parameter. S'Class'Output and S'Class'Input §13.13.2(28–34) are similar except that a representation of the tag is written to the stream and used upon input to reconstruct a value of the corresponding specific type. This is clearly what is needed in the example: the tag of each event is written and restored upon reading. Fortunately, S'Input and S'Class'Input are functions, rather than procedures, so we can use the result to give an initial value to an indefinite type, or as an actual parameter to a subprogram with a formal parameter of indefinite type ‡30. Note the type conversions to Event'Class ‡18–23: the attributes expect a value of class-wide type, while the Create functions return values of one of the specific types in the class.

Finally, package Ada.Text_IO.Text_Streams §A.12.2 enables you to obtain a stream associated with a text file. This can be used to include 'binary' data within the file.

12 Program Structure

12.1 Compilation and execution

This section discusses topics from §10 'Program Structure and Compilation Issues'. The basic definitions are given in §10.1:

§10.1

1 A *program unit* is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

2 The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit. The representation for a compilation is implementation-defined.

3 A library unit is a separately compiled program unit, and is always a subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a *subsystem*.

4 An implementation may impose implementation-defined restrictions on compilations that contain multiple compilation_units.

Note that a context clause is associated with a single compilation_unit §10.1.1(3), §10.1.2; if several units 'with'ing the same package are contained in one compilation, each unit must have a context clause for the package.

Semantic dependencies §10.1.1(23–26) are used to determine both visibility and the order of compilation. Before a unit can be compiled, *consistent* versions of all units upon which the unit depends semantically must have been previously compiled §10.1.4(5). For example, if the program depends on two packages P1 and P2 which both 'with' Q, they must have been compiled in the context of the same version of Q.

Previously compiled units are stored in a conceptual 'program library' called an *environment* §10.1.4(1). The rules concerning the environment are intentionally left vague to allow any implementation that can satisfy the requirements of dependency and consistency.

The GNAT compiler exploits the permission given in §10.1(4) and forbids multiple compilation units in a compilation; instead, a tool is provided for 'chopping' a file containing several units into

files with one unit apiece. The environment is simply the operating system's file system that stores source files; compiling a unit causes compilation of all other units (usually, specifications) that the unit depends on.

Inline subprograms

Pragma Inline §6.3.2 is a recommendation §6.3.2(6) to the compiler that the code for a subprogram be expanded inline at the point of a call to save the overhead of a jump to and return from the subprogram. This trades space for time. Pragma Inline should be used sparingly, or at least its use should be deferred until late in the development of a program. The reason is that recompilation of a package *body* containing an inlined subprogram may require recompilation of every unit that depends on the package *specification*, even if the subprogram body was not modified §10.1.4(7).

12.2 Subunits

If a unit becomes very large, it is possible to 'break off' enclosed bodies into *subunits* §10.1.3, creating *stubs* in place of the bodies and then creating additional compilation_units for the bodies. In the following example, the bodies of Proc1 and Inner have been replaced by stubs and moved into subunits.

```
1  package P is                                          -- File: SUBUNIT
2    procedure Proc1;
3  end P;
4
5  package body P is
6    S: String := "Global variable";
7    package Inner is
8      procedure Proc2;
9    end Inner;
10   procedure Proc1 is separate;                        -- Body stub
11   package body Inner is separate;                     -- Body stub
12   -- Body of Inner is illegal here
13 end P;
14
15   with Ada.Text_IO; use Ada.Text_IO;                  -- Additional context clauses
16   separate(P)                                         -- Subunit
17   procedure Proc1 is
18   begin
19     Put_Line(S & " visible from Proc1");
20     Inner.Proc2;
21   end Proc1;
22
```

```
23    separate(P)                                   -- Subunit
24    package body Inner is
25       procedure Proc2 is separate;               -- Body stub
26    end Inner;
27
28    with Ada.Text_IO; use Ada.Text_IO;            -- Additional context clauses
29    separate(P.Inner)                             -- Subunit
30    procedure Proc2 is
31    begin
32       Put_Line(S & " visible from Proc2");
33    end Proc2;
34
35 with P;
36 procedure Subunit is
37 begin
38    P.Proc1;
39 end Subunit;
```

Subunits are transparent in terms of visibility §10.1.3(16–17). The variable S ‡6 is visible within Proc1 ‡19 and Proc2 ‡32, and Inner.Proc2 is visible within Proc1 ‡20, as if the subunits had textually replaced the stubs. Subunit may have their own context clauses ‡15,28 §10.1.1(3), so they can be used to reduce dependencies of a unit. The body of Inner cannot be placed within P ‡12, because it also contains a stub, and stubs must appear immediately within a compilation unit body §10.1.3(13). A subunit depends semantically on its parent §10.1.1(26), so any change in the parent requires recompilation of all subunits.

To summarize, there are four ways of organizing packages:

- Library packages
- Nested packages
- Child packages
- Subunits

A nested package can be used to reduce name-space clutter by encapsulating declarations, but the package is still part of the unit's compilation. A library package achieves maximum compilation independence, but it can no longer access declarations within the parent unit body. A child package can access declarations in the package specification but not in the body. The compilation of a child is relatively independent from that of its parent, because it only depends on its parent's specification, not its body. A subunit can access its parent's body, but it is tightly coupled to the parent because any change in the parent requires recompilation of the subunit.

12.3 Pragmas*

Pragmas are directives to the compiler §2.8. (A list of language-defined pragmas can be found in §L.) Special rules §10.1.5 apply to the placement of certain pragmas. *Configuration pragmas* apply to an entire program, and the pragma declaration itself is a compilation unit §10.1.5(8–9), which is normally the first unit that is compiled. For example, pragma Restrictions §13.12 is a configuration pragma used to inform the compiler that you intend to restrict the use of the language, perhaps by avoiding allocators. Use of an allocator anywhere within the program would then be diagnosed as an error.

Other pragmas are classified as *program unit pragmas* §10.1.5(2–6) (or *library unit pragmas* if they apply only to library units). Such pragmas can be placed either within the unit they apply to, or after the unit. If the pragma is the first entity within a unit, you can sometimes omit the name of the unit. In the next subsection, you will see two possible placements of the library unit pragma Elaborate_Body.

12.4 Elaboration*

Elaboration order

Elaboration §3.1(11) is the process by which a declaration has its run-time effect. For example, a variable declared within a subprogram is elaborated as part of the execution of the subprogram body §6.3(7). The question is now: When are library units elaborated?

The answer is that they are considered to be declared local to an *environment task* §10.2(8) whose execution is initiated by the operating system. The units are elaborated in any order that is consistent with their semantic dependencies §10.2(9). The environment task that elaborates all library units and calls the main subprogram is usually constructed in a step called 'binding', which occurs after compilation and before linking.

Occasionally, stronger control over the elaboration order is required. Consider the following program, where Func ‡2, 6–9 contains a 'complicated' computation used to initialize a variable N ‡14 in a package specification:

```
 1 package P is                                          -- File: ELAB
 2    function Func return Integer;
 3 end P;
 4
 5 package body P is
 6    function Func return Integer is
 7    begin
 8       return 10;
 9    end Func;
10 end P;
11
```

```
12 with P;
13 package Q is
14    N: Integer := P.Func;
15 end Q;
16
17 with Q;
18 with Ada.Text_IO;
19 procedure Elab is
20 begin
21    Ada.Text_IO.Put_Line(Integer'Image(Q.N));
22 end Elab;
```

The semantic dependencies require that the package specification of P be elaborated before the package specification of Q, but the body of P may be elaborated either before or after the specification of Q. If the specification of Q is elaborated before the body of P, then the call to the initialization function Func will be a call to a subprogram that does not yet exist! If this elaboration order is chosen, Elaboration_Check would fail and Program_Error will be raised §11.5(19–20).

A similar problem is shown in the following program, where the package specification contains a translation table whose initial value is set in a 'complicated' computation in the initialization of the package:

```
1 package Table is                                          -- File: TABLE
2    Translate: array(1..10) of Character;
3 end Table;
4
5 package body Table is
6 begin
7    Translate := (others => 'X');
8 end Table;
```

A library package is not allowed to have a body unless one is actually needed §7.2(4),[1] usually to contain the bodies of subprograms declared in the specification. In the example, the package body containing the initialization would be illegal, because the package specification is legal without the body.

To solve these problems, you can use elaboration pragmas. The most useful is Elaborate_Body §10.2.1(19–26), which requires that the body of the package be elaborated immediately after the specification. Elaborate_Body is a library unit pragma §10.2.1(24) so it can be placed either within the package declaration or immediately following the declaration §10.1.5(4):

```
package P is
   pragma Elaborate_Body;
   function Func return Integer;
end P;
```

[1] This rule simplifies configuration management.

```
package Table is
   Translate: array(1..10) of Character;
end Table;
pragma Elaborate_Body(Table);
```

Alternatively, we could have used pragma Elaborate(P) in the context clause of package Q of the first example to specify that P be completely elaborated before Q. Pragma Elaborate_All(P) is a transitive version of Elaborate(P); all units upon which P depends would be elaborated before Q.

Note that elaboration checks can be very expensive because the code for the check would be executed for *every* call of a subprogram declared in a package specification, even though it is only relevant during the first call. If you are satisfied that you have solved every possible elaboration problem (using elaboration pragmas, if necessary), you may want to suppress the check.

Elaboration control

There are pragmas to specify that a library unit be Pure §10.2.1(13–19) or Preelaborate §10.2.1(2–12). All pure units are elaborated before all preelaborable units, that in turn are elaborated before a unit to which neither of the pragmas applies §10.2(13–17). A pure unit is one which contains no state such as a variable, only constants and subprograms; thus a pure unit can be replicated in a distributed system (Section 16.5). A preelaborable unit contains no executable statements, so the elaboration can be done as part of the construction of the executable program; in particular, the implementation may be able to store the elaborated unit in ROM.

Package Ada §A.2 and some of its children are declared pure; some children are preelaborable, while others, such as Ada.Text_IO, are neither.

12.5 Renamings

Objects, exceptions, packages, subprograms and generic units can be renamed §8.5. Renaming is primarily used to declare an additional, shorter, name for an entity, especially if you do not want to use 'use' clauses:

```
package LEF is new Ada.Numerics.Long_Elementary_Functions;
```

A 'use' clause for part of the hierarchy can often be helpful in shortening names without the potential confusion caused by making everything directly visible. The main subprogram of the simulation has 'use' clause for Root_Event so that the child packages such as Telemetry are directly visible.

Object renamings can be used to improve readability and (depending on the optimizer) run-time efficiency when accessing complex data structures:

```
for N in A1'Range loop
  declare
    V: Integer renames A1[N].Field1.A2[N].Field2.all;
  begin
    V := (V + K1) * (V - K2) / 4*V;
  end;
end loop;
```

Subprogram renamings §8.5.4 come in two flavors: *renaming-as-declaration* and *renaming-as-body*. The former gives a new name (and new parameter names and default expressions if needed) to a subprogram or entry. The latter completes the declaration of a subprogram with an existing subprogram, so you don't have to write a trivial subprogram to call it. Renaming-as-body makes it easy to export a partial view of an abstraction. The following package exports Put and Put_Line from Ada.Text_IO by renaming-as body. Put from Ada.Integer_Text_IO is exported by writing a body that calls it, because we want to change the number of parameters, and renaming requires subtype-conformant profiles §8.5.4(5).

```
 1 package Renam is                                          -- File: RENAM
 2   procedure Writeln(S: in String);
 3   procedure Write(S: in String);
 4   procedure Write(I: in Integer);
 5 end Renam;
 6
 7 with Ada.Text_IO; with Ada.Integer_Text_IO;
 8 package body Renam is
 9   procedure Writeln(S: in String) renames Ada.Text_IO.Put_Line;
10   procedure Write(S: in String) renames Ada.Text_IO.Put;
11   procedure Write(I: in Integer) is
12   begin
13     Ada.Integer_Text_IO.Put(I);                  -- Put has three parameters
14   end Write;
15 end Renam;
```

An instance of a generic package is a package and can be renamed. A generic unit can also be renamed; for example, §J.1 renames children of package Ada for compatibility with Ada 83.

12.6 Use type clause

Is the following program legal?

```
 1 package P is                                          -- File: QUESTION
 2   type T is (A, B, C, D);
 3 end P;
 4
 5 with P;
 6 procedure Question is
 7   V: P.T;
 8 begin
 9   if V = P.A then null; else null; end if;
10 end Question;
```

The answer is no. The problem is with the expression in the condition of the if-statement. V and P.A are both of type P.T, but the operator "=" for the type is not visible. A 'use'-clause for the package P would, of course, make it visible, but many Ada programmers prefer to minimize the number of 'use'-clauses. It would also be possible to use prefix notation: P."="(V,P.A), but this syntax is unnatural. A better solution is furnished by the *use type clause*:

```
use type P.T;
```

The *primitive operators* of the type, but not other entities of the package, become visible §8.4(8). (See Quiz 53.) More exactly, they become potentially use-visible, as described in the next section.

12.7 Visibility rules**

Visibility of entities in Ada has been informally discussed throughout the text. The rules in §8.1–§8.4 are among the most difficult in the *ARM*, but fortunately you can usually write correct programs without understanding them in detail. Here we will survey the terminology used in case you need to study the rules; in addition, we will point out a few details worth knowing.

A *declarative region* §8.1 is a portion of the program text that can contain other declarations. Most declarative regions are declarations: packages, procedures, records and so on, but so is a for-loop statement that declares its loop parameter. The declarative region of a package includes its body, children and subunits.

The *scope* §8.2 of a declaration is the portion of a program text where it is legal to refer to the declared entity. The scope includes the *immediate scope*, which extends from the declaration itself until the end of the *immediately enclosing* declarative region. For example, the immediate scope of a type declared in a package specification comprises the remainder of the package specification, as well as the package's body, children and subunits. The *scope* of the type includes its immediate scope and extends to include the scope of the package itself §8.2(10) and of its semantic dependents §8.2(3).

Certain parts of an entity are *visible* §8.2(5–9); for example, the visible part of a package specification, the profile of a subprogram and the components of a record. Other parts of the entity are *private*.

Within its immediate scope, a declaration is normally *directly visible* so it can be referred to by using its simple ('direct') name alone. A declaration can be directly visible because it is *immediately visible*, or because it is *use-visible*. For example, a type declared in a package specification is immediately visible within the package itself; outside the package, it is directly visible only if a use clause has been given for the package. Within its scope but not its immediate scope, selector syntax such as P.T must be used to access an entity.

A 'use' clause only makes the names in a package specification *potentially use-visible* §8.4(8). Name conflicts—'use' of two packages that both declare the same name, or a local declaration with the same name as one in a 'use'd package—will prevent direct visibility, and you will have to use selector syntax.

> 8 Two declarations are *homographs* if they have the same defining name, and, if both are overloadable, their profiles are type conformant. An inner declaration hides any outer homograph from direct visibility. **§8.3**
>
> 9 Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other ...

Are the following two declarations homographs?

```
procedure Proc(A: Integer);
procedure Proc(B: Positive);
```

The answer is yes because: (a) they have the same defining name Proc, (b) as procedures they are overloadable and (c) they are type-conformant. Since they are homographs, they cannot appear in the same declarative region. If Positive were changed to Float, the second declaration would no longer be type-conformant, and the overloaded declarations could appear in the same declarative region.

A declaration can be *hidden* by an inner homograph; in this case, it is *hidden from direct visibility*, but not *hidden from all visibility*, and can be accessed using selector syntax:

```
procedure P is
  X: Integer;
  procedure Q is
    X: Integer;                    -- Hides outer declaration
  begin
    X := 2;                        -- Local declaration directly visible
    P.X := 3;                      -- Global declaration visible, but not directly
  end Q;
begin
  Q;
end P;
```

Some declarations are *hidden from all visibility* §8.3(4–20); in particular, once an inherited declaration is overridden, there is no way to name it:

```
type T is (A, B, C, D);
procedure P(X: T);
type T1 is new T;
  -- Inherited P is visible
procedure P(X: T1);
  -- Inherited P is hidden from all visibility
```

There are special visibility rules for context clauses; see §10.1.6.

12.8 Overloading**

The rules for overloading are described in §8.6. As with the visibility rules, they are quite complex, but you will rarely need to understand the details. At worst, the compiler will be unable to disambiguate the use of a name, and you can supply additional syntax such as an expanded name or a qualification.

Overloading resolution is done within each *complete context* §8.6(4–6) such as a declaration or a statement. An *interpretation* of the context is determined by first using the syntax and visibility rules to list *possible interpretations*. A possible interpretation is *acceptable* if it obeys the overloading rules, that is, if the interpretation fits the *expected type* or *profile* defined in the 'Name Resolution Rules'.

Consider the following program:

```
 1 procedure Over is                                        -- File: OVER
 2
 3   function F(A: Integer) return Boolean is
 4   begin
 5     return False;
 6   end F;
 7
 8   function F(A: Integer) return Integer is
 9   begin
10     return 1;
11   end F;
12
13   function F(A: Long_Integer) return Boolean is
14   begin
15     return False;
16   end F;
17
18 begin
19   if F(1) then null; else null; end if;
20 end Over;
```

The syntax rules of the if-statement require that the condition be an expression §5.3(3). So all three functions are possible interpretations. However, the Name Resolution Rule §5.3(4) specifies that the condition must be of a boolean type. So only the first and third functions are acceptable

interpretations. Since there must be only one acceptable interpretation §8.6(28, 31), the call to F is ambiguous and the program illegal. If the third function were not declared, overloading resolution would succeed, choosing the first function as the only acceptable interpretation. Alternatively, you could write the parameter as a qualified expression F(Long_Integer'(1)) to disambiguate the call.

The concept of *expected type* must be extended to consider the cases of class-wide, universal and access types; see §8.6(25) for details.

Overloading is also affected by the preference for root numeric types §8.6(29) as discussed in Section 10.8.

13 Concurrency

A program contains one or more *tasks*[1] that execute *concurrently*. We use concurrent in preference to 'parallel' to emphasize that the parallelism is conceptual, not necessarily physical. A correct multitasking Ada program will produce the same result, whether it is run on a multiprocessor system or on a time-shared single processor, though the multiprocessor system will (hopefully) be significantly faster.

For convenience, the material on concurrency is divided into two chapters, with more advanced material in Chapter 14. Multitasking programs are frequently written for embedded computer systems where hardware interface and program performance are critical. This chapter and the next will present the logical aspects of concurrency, leaving the systems aspects to Chapters 15 and 16.

13.1 Concepts

In this section, we survey the basics concepts of concurrency. If you have never studied this topic before, you may want to first read a textbook on the subject (see Appendix G).

A task is like a subprogram except that a *thread of control* is associated with a task. The thread is represented by a data structure containing a pointer to the task's current instruction and to some local memory such as a stack segment. If each task is assigned a processor, the processors will execute the instructions of the tasks simultaneously. If there are more tasks than processors—at worst if there is only one processor and more than one task—a *scheduler* will assign processors to tasks according to some algorithm.

Interleaving

To abstract away from these differences, we look at *interleaved* execution sequences. Given two tasks T1 and T2, the simultaneous execution of the 'next' instruction of each of the tasks is considered to be equivalent to one of two cases: either T1 executes its next instruction before T2, or conversely. It is as if a global scheduler decides at each step which task is allowed to execute an instruction.

> A concurrent program is correct if *all* interleaved execution sequences give the correct result.

[1] Tasks are called *processes* or *threads* in other systems and languages.

If we had to deal with overlapped execution of instructions in a continuous time frame, it would be extremely difficult to reason about concurrent programs, but powerful mathematical tools exist for reasoning about interleavings (Manna & Pnueli 1992).

It is impossible to 'debug' a concurrent program, if by debugging you mean running and rerunning the program, tracing its execution and looking for an error. The number of possible interleavings of even simple programs is astronomical; furthermore, in embedded systems with hardware interfaces, it is impossible to recreate a scenario at will. The only way to validate a concurrent program is to prove (informally, if not formally) that there can be no incorrect scenarios.

Atomic instructions

If several tasks are totally independent of each other, then all interleaved executions will essentially give identical results. Interleaving is significant only in the presence of shared resources that require task synchronization, or for communications between tasks. For interleaving to be well-defined, it is necessary to introduce the concept of *atomic instruction*.

Suppose that T1's next instruction is X:=1 and that T2's next instruction is X:=2, for some shared variable X. It is conceivable that the simultaneous execution of the instructions will leave X holding the value 3. If the store is done by clear'ing the memory cell and then or'ing the bits of the value, simultaneous execution could cause both clear's to be done before the or's. This is know as a *race condition*: a scenario specifying an interleaving that leads to an incorrect result.

On most computers, instructions that load values from memory into a register and store values from a register to memory are atomic, and the above scenario will not occur. Nevertheless, simple race conditions still exist. Consider two tasks executing the assignment statement X:=X+1 on a shared variable X. The assignment statement is compiled into a sequence of machine code instructions such as:

```
Load  X
Add   A,#1
Store X
```

Consider the following scenario, where the initial value of X is zero:

- Task 1 loads 0 from X into its accumulator
- Task 2 loads 0 from X into its accumulator
- Task 1 adds 1 to the value in its accumulator
- Task 2 adds 1 to the value in its accumulator
- Task 1 stores 1 from its accumulator into X
- Task 2 stores 1 from its accumulator into X

The assignment has been executed twice, but X contains the value 1!

Concurrency constructs in Ada

The essence of concurrent programming is to define atomic instructions and then to develop (and prove!) algorithms that use these instructions to solve problems of synchronization and communication between tasks. In Ada, there are three constructs for concurrency:

- Load and store of shared variables. This is usually too low-level and will be treated in Section 15.3.
- Protected objects for asynchronous sharing of resources.
- Rendezvous for direct task-to-task synchronous communication.

The term 'asynchronous' means that tasks need not access the protected object at the same time. In fact, a task can insert data into a protected object and then terminate, while a second task later extracts the data from the object. The rendezvous is 'synchronous' because both tasks must participate in the rendezvous at the same time.

Protected objects are very efficient and are appropriate for solving simple synchronization problems. Rendezvous will be more appropriate when you wish to maximize potential concurrency. In the next two sections, we will solve a simple problem, once using protected objects and once using rendezvous, so that we can compare the constructs.

13.2 Tasks and protected objects

The problem that we will solve is called the *producer–consumer* problem. One or more tasks produce data elements which must be transferred to one or more consumer tasks. An example would be a network interface that 'produces' data downloaded from the net, and a graphical browser program that 'consumes' the data. A *buffer* is used for structure transformation and flow control. The data elements may arrive in large blocks, which must be stored so that the browser can process one element at a time. In addition, the browser must be blocked if no data elements are currently available, and similarly, the interface must not download data elements if the storage area is full.

Case study: producer–consumer (protected object)

The following program solves the producer–consumer problem with integers as the data elements for simplicity.

```
1  --                                                    -- File: PROTECT
2  -- Producer-consumer using protected object.
3  --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure Protect is
6    type Index is mod 8;
7    type Buffer_Array is array(Index) of Integer;
8
```

```
 9  protected Buffer is
10    entry Append(I: in Integer);
11    entry Take (I: out Integer);
12  private
13    B: Buffer_Array;
14    In_Ptr, Out_Ptr, Count: Index := 0;
15  end Buffer;
16
17  protected body Buffer is
18    entry Append(I: in Integer) when Count < Index'Last is
19    begin
20      B(In_Ptr) := I;
21      Count := Count + 1;
22      In_Ptr := In_Ptr + 1;
23    end Append;
24
25    entry Take(I: out Integer) when Count > 0 is
26    begin
27      I := B(Out_Ptr);
28      Count := Count - 1;
29      Out_Ptr := Out_Ptr + 1;
30    end Take;
31  end Buffer;
32
33  task Producer;
34  task body Producer is
35  begin
36    for N in 1..200 loop
37      Put_Line("Producing " & Integer'Image(N));
38      Buffer.Append(N);
39    end loop;
40  end Producer;
41
42  task type Consumer(ID: Integer);
43  task body Consumer is
44    N: Integer;
45  begin
46    loop
47      Buffer.Take(N);
48      Put_Line(Integer'Image(ID) & " consuming " & Integer'Image(N));
49    end loop;
50  end Consumer;
51
52  C1: Consumer(1);
53  C2: Consumer(2);
```

```
54
55 begin
56    null;
57 end Protect;
```

There is one Producer task object ‡33 and two tasks C1 and C2 ‡52–53, which are declared to be of task type Consumer ‡42. The task declarations in this program must be present even though they are empty §9.1(8). The task declaration for Consumer contains a discriminant §9.1(2,16), which is used for configuring a task with an identification number. A task body is syntactically like a procedure body §9.1(6).

The producer creates two hundred data elements, which are appended to the buffer ‡36–39; each consumer is in an infinite loop, taking and 'consuming' the elements ‡46–49. Note that the main subprogram ‡55–57 is empty! The tasks are activated just after the **begin** of the main subprogram, which must wait until the tasks have terminated. The producer will clearly terminate, but the consumer tasks will not. Make sure that you know how to break the execution of an Ada program on your computer (usually CTRL-C) before running this example. Section 14.1 will discuss activation and termination of tasks in detail.

Buffer ‡9–31 is a protected object. Syntactically, a protected unit (which can be either a single object, or a type that can be used to declare objects §9.4(1)) is like a package with a declaration—divided into a visible part and a private part—and a body §9.4(2–9). A protected unit cannot contain type declarations, so the data types used to implement the buffer ‡6–7 have been declared in the enclosing procedure. The visible part of the protected object contains the declaration of two *entries* Append and Take ‡10–11. The private part contains the declaration of components ‡13–14 belonging to the protected object. Components can only be declared in the private part, while operations such as entries can be declared anywhere in the protected unit declaration §9.4(4–6).

Buffer.Append(N) ‡38 and Buffer.Take(N) ‡47 are *entry calls* §9.5.3. An entry call is syntactically a procedure call and causes the body of the entry to be executed, passing parameters to and from the body.

Note that task units and protected units are *not* compilation units; they must be declared within a compilation unit such as a subprogram or a package. However, a task or protected *body* can be separately compiled as a subunit §10.1.3(10).

Protected actions

How is a protected unit different from a package? The subprograms of a package can be called concurrently §6.1(35) from multiple tasks, possibly leading to race conditions. The operations of a protected unit are *mutually exclusive*, meaning that only one will be executed at a time.

> 4 A new protected action is not started on a protected object while another protected action on the same protected object is underway, ... This rule is expressible in terms of the execution resource associated with the protected object: **§9.5.1**

5	Starting a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object ... for exclusive read–write access ...	**§9.5.1**
6	Completing the protected action corresponds to *releasing* the associated execution resource.	

For example, the producer call to Append and a consumer call to Take may try to update Count simultaneously, but no race condition occurs because one of the tasks will acquire the lock ('execution resource') and complete the entry body before the other task is allowed to begin the call.

The protected object Buffer also provides flow control, using *entry queues* §9.4(17) and *barriers* §9.5.2(5,7). We must prevent the producer from calling Append if the buffer is full and a consumer from calling Take if the buffer is empty.

7	An entry of a protected object is open if the condition of the entry_barrier of the corresponding entry_body evaluates to True; otherwise it is closed. ...	**§9.5.3**
8	For the execution of an entry_call_statement, evaluation of the name and of the parameter associations is as for a subprogram call (see 6.4). The entry call is then *issued*: For a call on an entry of a protected object, a new protected action is started on the object (see 9.5.1). The named entry is checked to see if it is open; if open, the entry call is said to be *selected immediately*, and the execution of the call proceeds as follows:	
10	For a call on an open entry of a protected object, the corresponding entry_body is executed (see 9.5.2) as part of the protected action.	
12	If the named entry is closed, the entry call is added to an *entry queue* (as part of the protected action, for a call on a protected entry), and the call remains queued until it is selected or cancelled; there is a separate (logical) entry queue for each entry of a given task or protected object (including each entry of an entry family).	

The barrier **when** Count $>$ 0 ‡25 closes the Take entry when the buffer is empty. A consumer calling Take will be enqueued on the entry queue for Take. Similarly, the barrier **when** Count $<$ Index'Last ‡18 closes the Append entry when the buffer is full.

Suppose the buffer is empty and that one or more calls from consumer tasks are enqueued on the queue for Take. In this state, only the producer will now succeed in passing its barrier and commencing the execution of its protected action Append. During the execution of the entry body, the value of Count will be incremented ‡21.

13	When a queued call is *selected*, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called *servicing* the entry queue. An entry with queued calls can be serviced under the following circumstances:	**§9.5.3**
15	If after performing, as part of a protected action on the associated protected object, an operation on the object, ... the entry is checked and found to be open.	

The completion of a protected operation that could potentially change the value of the barrier causes the system to reevaluate the barrier. When Append is completed Count>0 is now true, the entry queue is *serviced* and the Take operation is executed on behalf of a call from a consumer

task. When Take is completed, the barrier is again reevaluated, but now Count>0 is false because Take decremented the variable ‡28. Additional calls from consumer tasks that are on the queue remain blocked. The protected action is now completed.

Preference for servicing queues

Consider the following scenario, where we assume that there are multiple producer tasks:

- A consumer attempts to take an element from an empty buffer and blocks on the entry queue.
- Several producers attempt to append elements to the buffer. One will be allowed to execute the entry, appending its element to the buffer.
- When the entry is completed, there are two ways to continue:
  1. Another producer can be allowed into the entry body to append its element.
  2. The queue can be serviced so that a call from the enqueued consumer can take the newly appended element.

The language design could specify either of these possibilities or it could leave the choice unspecified.

18	For a protected object, the above servicing of entry queues continues until there are no open entries with queued calls, at which point the protected action completes.	**§9.5.3**

A new entry call (here from a producer) will not begin a protected operation until the ongoing protected action is completed; in other words, there is a *preference* for servicing calls already enqueued on an entry queue.

Figure 13.1 shows how protected objects should be viewed: an outer shell protecting access to the resources, and an inner set of operations and entry queues.

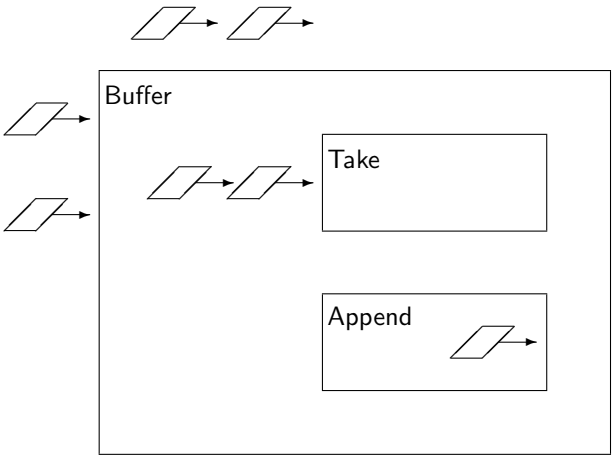

Figure 13.1: Protected object

Entry calls, represented by parallelograms with arrows, may be in one of three places: (i) executing an entry body (Append), (ii) blocked in an entry queue (Take), and (iii) attempting to enter the protected object. Calls that have already passed the outer shell are considered part of the 'club' and have preference over calls that have not yet been accepted.

There are two reasons for specifying this behavior, called *immediate resumption* of blocked tasks:

- If one task modifies a variable that will open a barrier for a blocked task, the awakened task can assume that no third task will intervene and change the state.
- Blocked tasks will not be starved by a stream of tasks entering from outside the protected object.

Note that there is no queue associated with the mutual exclusion on the protected object. (See Section 16.2 on the implementation of protected objects using ceiling priorities.) This will not be a problem if protected entries and subprograms are kept very short so that calls are either quickly processed or quickly enqueued awaiting an event; in either case the mutual exclusion is released.

13.3 Rendezvous

Let us examine more closely the entry body for Append:

```
entry Append(I: in Integer) when Count < Index'Last is
begin
  B(In_Ptr) := I;
  Count := Count + 1;
  In_Ptr := In_Ptr + 1;
end Append;
```

The barrier performs flow control for the calling task, and the entry parameter and the assignment to B are used to communicate—to pass data—between the task and the protected object. But incrementing Count and In_Ptr are solely concerned with updating the internal data structure of the protected object. Nevertheless, the calling task is responsible for executing these statements.[2]

Suppose that instead of using an array, the buffer was stored in a data structure or file system that required significant internal processing between insertions and extractions. We would like to overlap this processing with the execution of the producers and consumers. This can be done by making the buffer itself an additional task. Synchronization and communication is done directly with the buffer task, rather than through an intermediate structure (Figure 13.2). The producer and consumer both initiate calls on entries of the buffer task, but in the case of the consumer, the data flow (small arrow) is opposite the direction of the call.

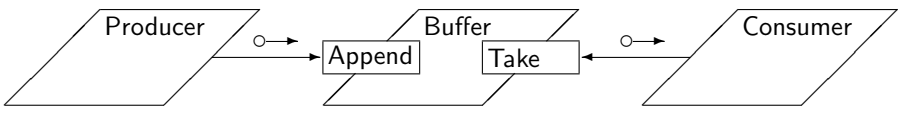

Figure 13.2: Active buffer

[2]See the subsection at the end of this section for a more detailed explanation.

Case study: producer–consumer (rendezvous)

```
1   --                                                    -- File: TASKPC
2   -- Producer-consumer using a buffer task.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  procedure TaskPC is
6    type Index is mod 128;
7    type Buffer_Array is array(Index) of Integer;
8
9    task Buffer is
10     entry Append(I: in Integer);
11     entry Take (I: out Integer);
12   end Buffer;
13
14   task body Buffer is
15     B: Buffer_Array;
16     In_Ptr, Out_Ptr, Count: Index := 0;
17   begin
18     loop
19       select
20         when Count < Index'Last =>
21           accept Append(I: in Integer) do
22             B(In_Ptr) := I;
23           end Append;
24         Count := Count + 1;
25         In_Ptr := In_Ptr + 1;
26       or
27         when Count > 0 =>
28           accept Take(I: out Integer) do
29             I := B(Out_Ptr);
30           end Take;
31         Count := Count - 1;
32         Out_Ptr := Out_Ptr + 1;
33       end select;
34     end loop;
35   end Buffer;
36
37   task Producer;
38   task body Producer is
39   begin
40     for N in 1..200 loop
41       Put_Line("Producing " & Integer'Image(N));
42       Buffer.Append(N);
43     end loop;
44   end Producer;
```

```
45
46    task type Consumer(ID: Integer);
47    task body Consumer is
48       N: Integer;
49    begin
50       loop
51          Buffer.Take(N);
52          Put_Line(Integer'Image(ID) & " consuming " & Integer'Image(N));
53       end loop;
54    end Consumer;
55
56    C1: Consumer(1);
57    C2: Consumer(2);
58
59 begin
60    null;
61 end TaskPC;
```

The producer and consumer tasks are unchanged from the previous solution. The declaration of the buffer task ‡9–12 contains the declaration of two entries. Unlike protected units, the declaration of a task can contain only entries and representation clauses §13.1(2), even in the private part §9.1(4–5). Synchronization and communication with the buffer task are done using a selective accept statement.

Accept statements

The task Producer calls the entry Append of Buffer; the entry call causes an accept statement to be executed in the called task.[3]

§9.5.2

3 accept_statement ::=
accept *entry*_direct_name [(entry_index)]
parameter_profile [**do**
handled_sequence_of_statements
end [*entry*_identifier]];

24 ... execution of the accept_statement is then blocked until a caller of the corresponding entry is selected (see 9.5.3), whereupon the handled_sequence_of_statements, if any, of the accept_statement is executed, with the formal parameters associated with the corresponding actual parameters of the selected entry call. Upon completion of the handled_sequence_of_statements, the accept_statement completes and is left. ...

25 The above interaction between a calling task and an accepting task is called a *rendezvous*. After a rendezvous, the two tasks continue their execution independently.

[3] The reserved word **do** is used in Ada only in this context! The reserved word **is** would be more consistent with other constructs in the language.

The calling task and the called task must execute a *rendezvous*. The basic principle of a rendezvous is that the first party to reach the rendezvous point must wait until the second party arrives. The semantics of a rendezvous are illustrated by the time lines in Figures 13.3 and 13.4. Solid lines indicate intervals during which the process is ready or executing; dashed lines indicate intervals when the process is blocked on the statement written within parentheses.

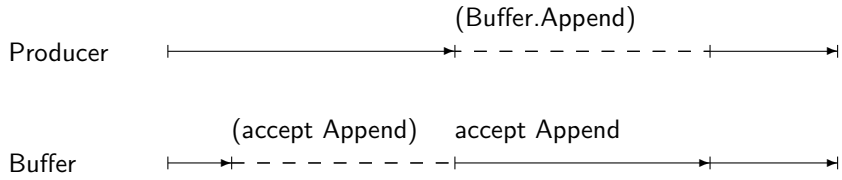

Figure 13.3: Rendezvous: blocking on an accept statement

In Figure 13.3, the buffer task executes until it reaches the accept statement ‡21, at which point it is blocked. The producer is allowed to continue until it executes the entry call Buffer.Append. Now the producer is blocked and the buffer executes the the sequence of statements within the accept statement. When the rendezvous is completed, both tasks are made ready; in a single-processor system the scheduler will have to choose one of them.

Figure 13.4 shows another possibility. Here the producer task blocks when it calls the entry because the buffer task has not yet reached the accept statement. The producer task is made ready again only when the accept statement has been completed.

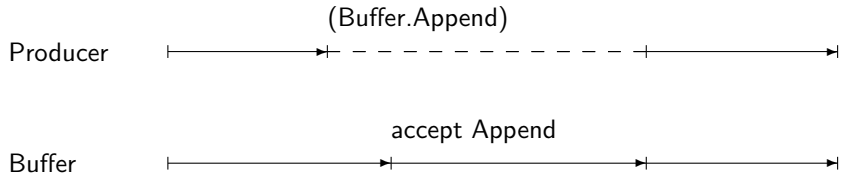

Figure 13.4: Rendezvous: blocking on an entry call

Before commencing the rendezvous, **in** and **in out** parameters are transferred to the accepting task; upon completion, **out** and **in out** parameters are transferred back to the calling task. Thus data transfer is bidirectional even though the entry call is unidirectional. Furthermore, the call is asymmetrical, because the calling task knows the name of the accepting task but not conversely.[4]

When the rendezvous is complete, both tasks can proceed independently so the producer is not blocked while the internal data structure is updated ‡24–25. The increased concurrency has been obtained at the price of additional overhead associated with the extra task, and additional *context switches* to block the caller and then resume it. Since protected objects can be implemented very efficiently on a single processor (Section 16.2), they are to be preferred unless the additional concurrency is actually needed.

[4]See Ben-Ari (1990) for a comparison of the Ada rendezvous with the more symmetrical synchronization constructs in occam and Linda.

Selective accept

The buffer task provides two services: Append for producers and Take for consumers. Suppose that the task Buffer arrives at the rendezvous, but neither producers nor consumers have yet issued a call. We would like Buffer to serve the *first* task that calls one of its entries; however, a task body is just a sequence of statements that have to be executed one after another. If we write:

```
loop
  accept Append do ...
  accept Take do ...
end loop
```

the Buffer would block pending a call from a producer even if there were waiting consumers and data in the buffer.

The selective accept statement ‡19–33 enables task to wait simultaneously for calls from multiple entries.

§9.7.1

```
2  selective_accept ::=
     select
       [guard] select_alternative
   { or
       [guard] select_alternative }
   [ else
     sequence_of_statements ]
   end select;
3  guard ::= when condition =>
4  select_alternative ::=
     accept_alternative | delay_alternative | terminate_alternative
5  accept_alternative ::=
     accept_statement [sequence_of_statements]
```

(Delay and terminate alternatives and the else-part are discussed in Section 14.3.) The semantics of the selective statement are as follows:

14	A select_alternative is said to be *open* if it is not immediately preceded by a guard, or if the condition of its guard evaluates to True. It is said to be *closed* otherwise.	**§9.7.1**
15	For the execution of a selective_accept, any guard conditions are evaluated; open alternatives are thus determined. ... Selection and execution of one open alternative, ... then completes the execution of the selective_accept; the rules for this selection are described below.	
16	Open accept_alternatives are first considered. Selection of one such alternative takes place immediately if the corresponding entry already has queued calls. If several alternatives can thus be selected, one of them is selected ... When such an alternative is selected, the selected call is removed from its entry queue and the handled_sequence_of_statements (if any) of the corresponding accept_statement is executed; after the rendezvous completes any subsequent sequence_of_statements of the alternative is executed. If no selection is immediately possible (in the above sense) ..., the task blocks until an open alternative can be selected.	

The Append entry is guarded ‡20 to ensure that a producer does not insert an element into a full buffer, and a guard also exists for Take ‡27. If the buffer is neither full nor empty, both guards are open; the Buffer task will select one of the alternatives with enqueued calls, if any, and perform a rendezvous. If there are no enqueued calls it will block, waiting for either a producer or a consumer to call an entry.

If one alternative is closed (say the buffer is full so that the Append alternative is closed), the Buffer task will rendezvous with a task blocked on the queue for the open alternative Take, if any, or Buffer will be blocked pending an entry call on the open alternative Take. A new call by a producer will be ignored. It is impossible for both alternatives in this example to be closed. (Prove!)

21	The exception Program_Error is raised if all alternatives are closed	**§9.7.1**

Implementation of entry calls**

§9.5.3(13) (quoted on page 202) talks about servicing a queued *entry call*, not a queued *task*. A protected object is just a passive set of components and operations, which is not associated with any particular task.

22	An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action.	**§9.5.3**

This is illustrated in Figure 13.5. Assume that the producer (in the program using protected objects) attempts to append an item to a full buffer. The entry call Buffer.Append will be enqueued, the producer task will be blocked and a consumer task will be allowed to execute. Eventually, the consumer task will take an item from the buffer. Upon completion of the body of Take, servicing of the entry queue can be done by the *consumer task*. In effect, appending an item is executed by

the consumer on behalf of the producer. Upon completion of the entry call, both tasks become ready.

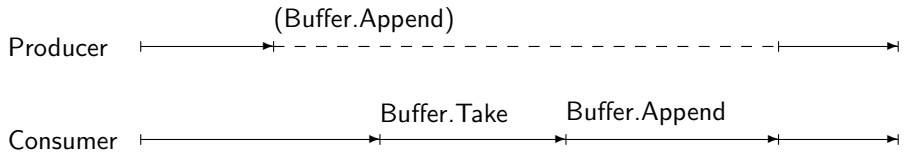

Figure 13.5: Implementation of protected entry call

In contrast, a rendezvous normally requires context switches as shown in Figure 13.6. Assume that the buffer task is blocked on the selective accept statement and that the alternative for Append is closed because the buffer is full. The producer task blocks on the entry call Buffer.Append. When the consumer task executes calls Buffer.Take, the rendezvous take place, removing an item from the buffer. When the selective accept is executed again, the alternative is now open and the entry call Buffer.Append can be accepted. Two extra context switches are needed (vertical arrows): one for the buffer task to execute its accept statement and another to switch either to the producer or to the consumer task.

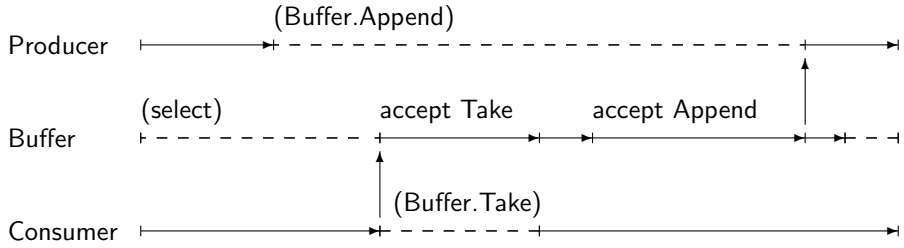

Figure 13.6: Implementation of task entry call

Protected objects are designed to be extremely efficient when implemented on a single processor. On a multiprocessor, rendezvous provides more parallelism that can be utilized. In the figure, you can see that the consumer can continue its execution immediately upon completion of its rendezvous with the buffer task, performing useful computation in parallel with the subsequent rendezvous between the producer and the buffer.

13.4 Case study: the CEO problem

The concurrent program presented in this section (see Ben-Ari (1998*a*)) employs both rendezvous and protected actions for synchronization. The case study will be used to study important constructs in Ada tasking: entry families, the requeue statement and the abort statement. The problem is to implement a synchronization scheme described by the following story:

> A CEO (Chief Executive Officer) of a company likes to play golf. He does not allow himself to be interrupted by single employees with problems; instead, they must form

themselves into groups before coming to consult him. The size of the group depends on the department to which the employee belongs: engineering, marketing, finance. A waiting group from finance has precedence over a group from marketing, which (obviously!) has precedence over an engineering group.

Let us look first at the structure of the program (Figure 13.7). There will be one task for the CEO and one task for each employee: engineers, salespersons in the marketing department, and accountants in the finance department. These are declared as task types, and the tasks themselves are dynamically allocated in the main subprogram. Protected objects of type Room are used to synchronize the groups.

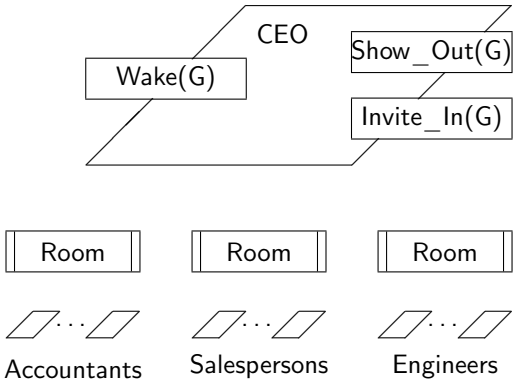

Figure 13.7: The CEO program

Because of the length and complexity of the program, we will present the source code of the program in this section together with a general description of its components. Details will be given in the following sections as we study each new construct.

The global types and constants are an enumeration type Departments naming the three departments, ID_Numbers for each employee in a group, and the Group_Size that defines for each department the size of the group that the CEO is willing to see.

```
1   --                                                    -- File: CEOT
2   --  CEO case study.
3   --
4  pragma Queuing_Policy(Priority_Queuing);
5  with Ada.Text_IO; use Ada.Text_IO;
6  procedure CEOT is
7
8    type Departments is (Engineering, Finance, Marketing);
9    type ID_Numbers is range 0..10;
10   Group_Size: constant array(Departments) of ID_Numbers := (
11     Engineering => 5, Finance => 3, Marketing => 2);
```

The declaration of the CEO task contains three entries: Wake to awaken the CEO task, Invite_In, which each employee task calls to enter the CEO's office, and Show_Out, which will block employee tasks until the consultation is finished. The declaration appears at the beginning of the program because its Wake entry will be called from within the protected type Room.

```
12  task CEO is
13    entry Wake(Departments);
14    entry Invite_In(Departments)(ID: ID_Numbers);
15    entry Show_Out(Departments)(ID: ID_Numbers);
16  end CEO;
```

Groups are synchronized in a waiting room implemented by a protected type. Entry Register is called by an employee task wishing to join a group. Once the correct number of employees for a group of this department is in the room, the entrance is closed and the group waits for the CEO to receive them. Procedure Open_Door is called by the CEO to allow a new group to register.

The protected type has a private entry Wait_for_Last_Member that is used to block registering tasks until the last member of the group has arrived. Private components are a count of waiting employee tasks, and the doors for entering and exiting the room. The protected body will be explained in the following sections.

```
17  type Door_State is (Open, Closed);
18
19  protected type Room(Department: Departments; Size: ID_Numbers) is
20    entry Register;
21    procedure Open_Door;
22  private
23    entry Wait_for_Last_Member;
24    Waiting: ID_Numbers := 0;
25    Entrance: Door_State := Open;
26    Exit_Door: Door_State := Closed;
27  end Room;
28
29  protected body Room is
30    entry Register when Entrance = Open is
31    begin
32      if Size = 1 then
33        Entrance := Closed;
34        requeue CEO.Wake(Department) with abort;
35      end if;
36      Waiting := Waiting + 1;
37      if Waiting < Size then
38        requeue Wait_for_Last_Member with abort;
39      else
40        Waiting := Waiting - 1;
41        Entrance := Closed;
42        Exit_Door := Open;
43      end if;
44    end Register;
45
```

```
46      entry Wait_for_Last_Member when Exit_Door = Open is
47      begin
48        Waiting := Waiting - 1;
49        if Waiting = 0 then
50          Exit_Door := Closed;
51          requeue CEO.Wake(Department) with abort;
52        end if;
53      end Wait_for_Last_Member;
54
55      procedure Open_Door is
56      begin
57        Entrance := Open;
58      end Open_Door;
59    end Room;
```

There is one room for each department which is obtained by declaring an object of the protected type Room. The protected type has two discriminants ‡19, which are used to specify the department and the size of the group.

```
60    Engineering_Room:    Room(Engineering, Group_Size(Engineering));
61    Finance_Room:        Room(Finance, Group_Size(Finance));
62    Marketing_Room:      Room(Marketing, Group_Size(Marketing));
```

The task body for the CEO is straightforward, consisting of a selective accept with an alternative for each of the three departments. The syntax of the selective accept statement is such that a (possibly guarded) accept statement must immediately follow the **select** and each **or**, but an arbitrary sequence of statements may be included in the alternative following the accept §9.7.1(5–6). The body of the accept statement for Wake is empty because it serves simply as a synchronization point. The sequence of statements following the accept statement implements the CEO's algorithm: the employees are invited in, allowed to consult with the CEO and are then shown out.

```
63    task body CEO is
64      I: ID_Numbers;
65    begin
66      loop
67        Put_Line("CEO is playing golf");
68        select
69          accept Wake(Finance);
70          for N in 1..Group_Size(Finance) loop
71            accept Invite_In(Finance)(ID: ID_Numbers) do
72              I := ID;
73            end Invite_In;
74            Put_Line("Accountant "&ID_Numbers'Image(I)&" in office");
75          end loop;
76          Put_Line("CEO is talking");
```

```
77          for N in 1..Group_Size(Finance) loop
78            accept Show_Out(Finance)(ID: ID_Numbers) do
79              I := ID;
80            end Show_Out;
81            Put_Line("Accountant "&ID_Numbers'Image(I)&" left office");
82          end loop;
83          Finance_Room.Open_Door;
84        or
85          when Wake(Finance)'Count = 0 =>
86            accept Wake(Marketing);
87          for N in 1..Group_Size(Marketing) loop
88            accept Invite_In(Marketing)(ID: ID_Numbers) do
89              I := ID;
90            end Invite_In;
91            Put_Line("Salesperson "&ID_Numbers'Image(I)&" in office");
92          end loop;
93          Put_Line("CEO is shouting");
94          for N in 1..Group_Size(Marketing) loop
95            accept Show_Out(Marketing)(ID: ID_Numbers) do
96              I := ID;
97            end Show_Out;
98            Put_Line("Salesperson "&ID_Numbers'Image(I)&" left office");
99          end loop;
100         Marketing_Room.Open_Door;
101       or
102         when Wake(Finance)'Count = 0 and
103             Wake(Marketing)'Count = 0 =>
104           accept Wake(Engineering);
105         for N in 1..Group_Size(Engineering) loop
106           accept Invite_In(Engineering)(ID: ID_Numbers) do
107             I := ID;
108           end Invite_In;
109           Put_Line("Engineer "&ID_Numbers'Image(I)&" in office");
110         end loop;
111         Put_Line("CEO is screaming");
112         for N in 1..Group_Size(Engineering) loop
113           accept Show_Out(Engineering)(ID: ID_Numbers) do
114             I := ID;
115           end Show_Out;
116           Put_Line("Engineer "&ID_Numbers'Image(I)&" left office");
117         end loop;
118         Engineering_Room.Open_Door;
```

```
119         or
120           terminate;
121         end select;
122       end loop;
123     end CEO;
```

The employee task types are very simple: an employee who needs to consult with the CEO registers at the department waiting room, waits until invited in and then receives orders from the CEO until shown out. Note that each task declaration has a discriminant that is used to give the task its ID number. The delay statements are used to introduce some asymmetry in the execution of the program.

```
124     task type Engineer_Task(ID: ID_Numbers);
125     task body Engineer_Task is
126     begin
127       loop
128         Put_Line("Engineer "&ID_Numbers'Image(ID)&" debugging");
129         delay 1.0;
130         Engineering_Room.Register;
131         CEO.Invite_In(Engineering)(ID);
132         Put_Line("Engineer "&ID_Numbers'Image(ID)&" receiving orders");
133         CEO.Show_Out(Engineering)(ID);
134       end loop;
135     end Engineer_Task;
136
137     task type Finance_Task(ID: ID_Numbers);
138     task body Finance_Task is
139     begin
140       loop
141         Put_Line("Accountant "&ID_Numbers'Image(ID)&" calculating");
142         delay 4.0;
143         Finance_Room.Register;
144         CEO.Invite_In(Finance)(ID);
145         Put_Line("Accountant "&ID_Numbers'Image(ID)&" receiving orders");
146         CEO.Show_Out(Finance)(ID);
147       end loop;
148     end Finance_Task;
149
150     task type Marketing_Task(ID: ID_Numbers);
151     task body Marketing_Task is
152     begin
153       loop
154         Put_Line("Salesperson "&ID_Numbers'Image(ID)&" talking");
155         delay 2.0;
156         Marketing_Room.Register;
```

```
157         CEO.Invite_In(Marketing)(ID);
158         Put_Line("Salesperson "&ID_Numbers'Image(ID)&" receiving orders");
159         CEO.Show_Out(Marketing)(ID);
160       end loop;
161     end Marketing_Task;
```

The tasks are allocated dynamically in loops so that each employee receives a distinct ID as a discriminant constraint in the allocator ‡172,175,178. Accesses to the tasks are stored in arrays so that the employees can be fired by using the abort statement. The CEO task will terminate because it has a terminate alternative ‡120. These constructs are explained in Section 14.1.

```
162     type Engineer_Ptr    is access Engineer_Task;
163     type Finance_Ptr     is access Finance_Task;
164     type Marketing_Ptr   is access Marketing_Task;
165
166     Engineers:      array(1..7) of Engineer_Ptr;
167     Accountants:    array(1..5) of Finance_Ptr;
168     Salespersons:   array(1..8) of Marketing_Ptr;
169
170   begin
171     for I in Engineers'Range loop
172       Engineers(I) := new Engineer_Task(ID_Numbers(I));
173     end loop;
174     for I in Accountants'Range loop
175       Accountants(I) := new Finance_Task(ID_Numbers(I));
176     end loop;
177     for I in Salespersons'Range loop
178       Salespersons(I) := new Marketing_Task(ID_Numbers(I));
179     end loop;
180
181     delay 15.0;
182     Put_Line("The company is bankrupt, fire everyone");
183
184     for I in Engineers'Range loop
185       abort Engineers(I).all;
186     end loop;
187     for I in Accountants'Range loop
188       abort Accountants(I).all;
189     end loop;
190     for I in Salespersons'Range loop
191       abort Salespersons(I).all;
192     end loop;
193   end CEOT;
```

13.5 Entry families

There are separate doors to the waiting rooms of each department and separate 'doorbells' to awake the the CEO from his golf practice. Rather than declare a single entry for each department, task CEO is declared with *entry families* Wake, Invite_In and Show_Out ‡13–15.

2 entry_declaration ::= **§9.5.2**
entry defining_identifier
[(discrete_subtype_definition)] parameter_profile;

20 An entry_declaration with a discrete_subtype_definition (see 3.6) declares a *family* of distinct entries having the same profile, with one such entry for each value of the *entry index subtype* defined by the discrete_subtype_definition. A name for an entry of a family takes the form of an indexed_component, where the prefix denotes the entry_declaration for the family, and the index value identifies the entry within the family. The term *single entry* is used to refer to any entry other than an entry of an entry family. **§9.5.2**

According to §9.5.2(20), each entry of the family is a distinct entry with its own queue. Entry families are somewhat like arrays of entries: calls and accept statements for an entry of a family use indexed notation as shown in ‡106,131, etc. The index need not be constant, but it is not possible to write an accept statement that will wait simultaneously on all entries of a family. If D is a variable, **accept** Wake(D) waits for an entry call on the queue corresponding to the current value of D. This construct was used to perform a rendezvous sequentially with each member of the family. Of course you would only use this construct if you knew that each entry would actually be called; otherwise, the program could deadlock. See Section 14.3 for other polling techniques, and Section 13.8 for the rules for entry families of protected objects.

The guards ‡85 and ‡102–103 in the selective accept statement of the CEO task use the Count attribute: E'Count gives the number of tasks currently waiting in the queue for entry E §9.9(4–5). This is used to implement the precedence requirement: if the CEO returns from the golf course and finds that more than one group is attempting to wake him, he will rendezvous with Marketing_Group only if there are no tasks waiting in the queue for Wake(Finance_Group). The guard for Engineering_Group is correspondingly more complex. Obviously, this programming paradigm for precedence is inconvenient and inefficient if the size of the entry family is large.

The solution still does not satisfy the precedence requirement because of a subtle race condition. See the discussion of pragma Queuing_policy ‡4 in Section 16.1.

13.6 Protected subprograms

The CEO task calls the *protected procedure* Open_Door ‡55–58 to indicate that the waiting room for this group can be re-opened.

1 A *protected subprogram* is a subprogram declared immediately within a protected_definition. Protected procedures provide exclusive read–write access to the data of a protected object; protected functions provide concurrent read-only access to the data. **§9.5.1**
2 Within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit is defined to be a constant (that is, its subcomponents may be read but not updated). Within the body of a protected procedure (or a procedure declared immediately within a protected_body), and within an entry_body, the current instance is defined to be a variable (updating is permitted).

Unlike entries, protected procedures have no queues associated with them and a call is not blocked once it passes the outer exclusion 'shell' of the protected object.

Several tasks can execute protected functions concurrently provided that no other task is executing a protected procedure or entry §9.5.1(4–5). §9.5.1(2) prevents functions from modifying components of the protected object, so no race conditions can result from this weakening of mutual exclusion. Protected functions are usually used to return values of the private components:

```
function Crowded return Boolean is
begin
  return Waiting >= Group_Size / 2;
end Crowded;
```

You can declare an access to a protected subprogram §3.10(11). This can be useful in the implementation of callbacks, as shown in Section 9.3.

13.7 The requeue statement

The implementation of the protected type Room is based on the concept of *cascaded wakeup*. As employee tasks enter the room to register, the calls will be enqueued upon the private entry Wait_for_Last_Member. Its barrier is closed, so the tasks will be blocked until the last task of the group has registered and opened the barrier. Re-evaluation of the barrier will cause one of the waiting tasks to become unblocked. Since there is nothing more to do in the body of the entry, the protected action for the unblocked task will complete, causing re-evaluation of the barrier and unblocking of another waiting task. This cascade of awakened tasks will continue until all tasks enqueued on Wait_for_Last_Member are released and their protected actions completed. The last released task will awaken the CEO.

The requeue statement is used to implement this algorithm.

1	A requeue_statement can be used to complete an accept_statement or entry_body, while redirecting the corresponding entry call to a new (or the same) entry queue. ...	§9.5.4
2	requeue_statement ::= **requeue** *entry*_name [**with abort**];	
8	For the execution of a requeue on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the requeued call is either selected immediately or queued, as for a normal entry call (see 9.5.3).	
9	For the execution of a requeue on an entry of a target protected object, after leaving the enclosing callable construct:	
10	if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object—see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);	

Room contains examples of requeue on both task and protected entries. Let us start with the case where employee tasks call Register.[5] After incrementing Waiting ‡36, all but the last task in the group will be requeued on the private entry Wait_for_Last_Member ‡38. The barrier of this entry (Exit_Door=Open ‡46) is closed, so the calls will be enqueued §9.5.4(10). When the final task executes Register, it will close the entrance door ‡41 and open the exit door ‡42.

Closing the entrance door closes the barrier Entrance=Open of entry Register ‡30 and prevents other employees of the same type from overtaking the group that has been formed. Opening the exit door opens the barrier of Wait_for_Last_Member ‡46; the protected action for the first enqueued task will be executed and will decrement Waiting ‡48, leading to the cascade described above. The last task will close the exit door ‡50 and requeue on the *task entry* CEO.Wake(Department) ‡51 to awaken the CEO.

Requeue is essential if we want to avoid overtaking. Suppose that an group of accountants has been formed, that is, the last task of the group has completed Wait_for_Last_Member. Without requeue, the last accountant task would have to complete the protected action and then call the entry CEO.Wake(Finance_Group) from within the sequence of statements of its task body:

```
Finance_Room.Register;
if I_am_Last_Member then CEO.Wake(Finance_Group); end if;
CEO.Invite_In(Finance)(ID);
```

This task could be preempted, so that it would be possible for a group of engineers to be formed and enqueued on CEO.Wake(Engineering_Group) *after* the protected action Finance_Room.Register completes, but *before* the call to CEO.Wake(Finance_Group) is issued. With requeue, the protected action of the last accountant task is not *completed* until the task entry call has been made and immediately selected or enqueued. If it is enqueued, the guards on the accept statements prevent overtaking.

[5] This explanation assumes that Group_Size is greater than one.

13.8 Rules for entries of protected types*

Formal parameters in barriers

18	A name that denotes a formal parameter of an entry_body is not allowed within the entry_barrier of the entry_body.	**§9.5.2**

For example, in the protected object Buffer, we cannot refuse to append negative numbers by writing:

```
entry Append(I: in Integer) when Count < Index'Last and I >= 0 is
-- Error, barrier cannot use formal parameter
```

The reason is that all calls on the queue for an entry are considered to be waiting for the same event to occur. If you want calls to wait for distinct events, you should use different entries or an entry family. Furthermore, allowing formal parameters in barrier would make it inefficient to evaluate barriers, because the run-time system would have to scan the entry queue and re-evaluate the barrier for each call. This way, the code executed for each barrier is *fixed* regardless of how many calls are enqueued.

If blocking of a call really does depend on the formal parameters, you can call an entry with the barrier True, examine the formal parameters within the body and requeue on other private entries.

Potentially blocking operations

A protected action is not allowed to invoke an operation that could result in blocking the calling task *within* the protected action.

8	During a protected action, it is a bounded error to invoke an operation that is *potentially blocking*. ...	**§9.5.1**

Potentially blocking operations are listed in §9.5.1(9–16); in particular, calling an entry is potentially blocking.

The bounded error *need not* be detected by the implementation §9.5.1(17)! Hopefully, you will get an error or warning message from the compiler, but since the potentially blocking operation could be invoked indirectly via a call on an arbitrary subprogram, it is impossible to catch all violations. It is not too difficult to analyze your program for violations of this rule, because you normally write very short protected operations which merely manipulate counters and state variables, rather than invoking external subprograms or synchronization constructs.

Parameters of the requeue target

3	The *entry*_name of a requeue_statement shall resolve to denote an entry (the *target entry*) that either has no parameters, or that has a profile that is type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement.	**§9.5.4**
12	If the new entry named in the requeue_statement has formal parameters, then during the execution of the accept_statement or entry_body corresponding to the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit.	

The requeue efficiently continues the same protected action with no need to allocate or deallocate local variables. In the case study, both requeues are to entries without parameters. The entry family index in **requeue** CEO.Wake(Group) is not a parameter.

Internal and external requeues

A call or requeue can be to the same protected object—an *internal call or requeue*—or it can be to a subprogram or entry of a different object—an *external call or requeue* §9.5(2–7). A call must be to a subprogram, because a call to an entry is potentially blocking. An external call or requeue uses the syntax of a selected component to distinguish it from an internal call or requeue: **requeue** Wait_for_Last_Member is internal and legal, but **requeue** Room.Wait_for_Last_Member is an external call to the same protected object and thus potentially blocking §9.5.1(15).

There is a minor difference in the semantics of internal and external requeues:

10	if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object—see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);	**§9.5.4**
11	if the requeue is an external requeue (that is, the target protected object is not implicitly the same as the current object—see 9.5), a protected action is started on the target object and proceeds as for a normal entry call (see 9.5.3).	

An external requeue will initiate a new protected action including an evaluation of the barrier; if the barrier is true, the requeued task will immediately begin executing the entry body. An internal requeue is merely enqueued on the the existing queue; then, as part of the completion of the protected action, the barrier will be re-evaluated. The requeued task will receive no precedence over tasks already enqueued on the entry.

Families of protected entries

An entry family can be declared in a protected unit; like an entry family for a task, it can be considered as if it were an array of entries. The syntax of the entry body is different from the

syntax for accept statements. One entry body with an entry_index_specification defines the common code for processing all entries in the family.

§9.5.2

```
5  entry_body ::=
     entry defining_identifier entry_body_formal_part
           entry_barrier is
       declarative_part
     begin
       handled_sequence_of_statements
     end [entry_identifier];
6  entry_body_formal_part ::=
     [(entry_index_specification)] parameter_profile
7  entry_barrier ::= when condition
8  entry_index_specification ::=
     for defining_identifier in discrete_subtype_definition
```

The following declaration of E1 declares a family of entries, each with one formal parameter. There is a *separate* entry and queue for each member of the family. E2 declares a *single* entry with two formal parameters.

```
entry E1(Departments)(I: in Integer);
entry E2(D: in Departments; I: in Integer);
```

The corresponding entry bodies are:

```
entry E1(for D in Departments)(I: in Integer) when Open is ...
entry E2(D: in Departments; I: in Integer) when Open is ...
```

The barrier of E1 can depend on its entry index D, though, as we have seen, the barrier of E2 cannot depend on its formal parameter D.

14 Advanced Concurrency

14.1 Activation and termination

> 10 Over time, tasks proceed through various *states*. A task is initially *inactive*; upon activation, and prior to its *termination* it is either *blocked* (as part of some task interaction) or *ready* to run. While ready, a task competes for the available *execution resources* that it requires to run. **§9**

Activation

It is important to distinguish between the elaboration of a task and its activation. Elaboration is performed as part of the elaboration of the enclosing package, subprogram or task. Elaboration of the task declaration can set its priority §D.1 and the amount of storage allocated for the task §13.3(61); elaboration of the task body essentially does nothing §9.1(13). Activation creates the task and allows it to begin execution, or at least to compete for the available execution resources. The rules for task activation §9.2 can be summarized as follows:

- Task objects (such as the producer and consumer tasks) are activated before the enclosing unit begins executing its statements §9.2(3). The enclosing unit does not begin executing until all its tasks have been activated §9.2(5).
- Task created by allocators (such as the employee tasks in the CEO program) are activated as part of the evaluation of the allocator §9.2(4).

The rationale for these rules is that during activation of a task, the elaboration of declarations in the task body §9.2(1) may raise an exception; similarly, the enclosing unit may raise an exception after elaborating the task §9.2(5–6). The rules ensure that the state of the tasks is well-defined.

Termination

Termination of tasks is defined in terms of a dependency tree. Each task *depends* on one or more *masters* §9.3(1), which are enclosing dynamic constructs such as a subprograms or tasks §7.6.1(3). The main subprogram is a task declared in an anonymous *environment task* §10.2(8). Tasks declared in library packages are also considered to be dependent on this master. The producer, consumer and buffer tasks were declared within the main subprogram and depend upon it. Tasks created by allocators (such as the employee tasks) depend on the master containing the

declaration of the access *type* §9.3(2). The reason is that the task can live as long as the type lives; it can even become a garbage task:

```
E := new Engineer_Task(7);
E := null;
```

A task cannot terminate until all dependent tasks have terminated.

> 5 A task is said to be *completed* when the execution of its corresponding task_body is completed. A task is said to be *terminated* when any finalization of the task_body has been performed (see 7.6.1). The first step of finalizing a master (including a task_body) is to wait for the termination of any tasks dependent on the master. The task executing the master is blocked until all the dependents have terminated. Any remaining finalization is then performed and the master is left. **§9.3**

§9.3(5) explains why we could write a null body for the main subprogram of the producer–consumer program. The main subprogram is a master, and the producer and consumer tasks depend on it, so the main subprogram cannot terminate until they do. The producer completes when it completes the statements of its task body. Since the consumer tasks never complete, the main subprogram waits indefinitely, as does the environment task that contains it. The program is deadlocked and the execution must be stopped by an operating system command such as CTRL-C.

How can server tasks such as Buffer and CEO be terminated. Almost by definition, a server task does not know when it has finished serving all potential clients. One possibility is to declare a special entry that can be used to signal the server:

```
loop
  select
    accept Append(I: in Integer) do ...
  or
    accept Take(I: out Integer) do ...
  or
    accept Stop;
    exit;
  end select;
end loop;
```

A better solution is to use the terminate alternative §9.7.1(7) on the selective accept statement. See ‡119–120 of the CEO program for an example. If every task that could possibly call the entries of the server task is completed (or also waiting on a selective accept statement with a terminate alternative), the server task becomes completed §9.3(6).

A protected object is not a task, so there is no question of its ‘terminating’. However, tasks can be blocked on its entry queues, and these tasks cannot be completed until they complete the entry call. Either you must explicitly program a protected operation that will free the tasks, or you must abort the blocked tasks.

Aborting a task

> 1 An abort_statement causes one or more tasks to become abnormal, thus preventing any further interaction with such tasks. ... **§9.8**
> 2 abort_statement ::= **abort** *task*_name {, *task*_name};

In the CEO program, we abort all the employee tasks because they contain infinite loops. Task CEO need not be aborted, because it contains a selective accept statement with a terminate alternative. Once the employee tasks are terminated and the main subprogram is completed, there are no longer any potential callers, so the CEO task can complete and terminate.

An abort statement is not a sledgehammer which blindly destroys tasks. The semantics of the statement ensure—as far as practicable—that the overall consistency of the tasks in a program is maintained even if one or more are aborted.

> 5 When the execution of a construct is *aborted* (including that of a task_body or of a sequence_of_statements), the execution of every construct included within the aborted execution is also aborted, except for executions included within the execution of an *abort-deferred* operation; the execution of an abort-deferred operation continues to completion without being affected by the abort; ... **§9.8**

Abort-deferred operations are listed in §9.8(6–11). In particular, protected actions and rendezvous are abort-deferred so that the state of the protected unit or accepting task remains consistent.

Furthermore, the implementation need not perform the abort immediately; instead, it can defer the abort to places that actually affect the synchronization of the program.

> 15 If the execution of an entry call is aborted, an immediate attempt is made to cancel the entry call (see 9.5.3). If the execution of a construct is aborted at a time when the execution is blocked, other than for an entry call, at a point that is outside the execution of an abort-deferred operation, then the execution of the construct completes immediately. ... Other than for these immediate cases, the execution of a construct that is aborted does not necessarily complete before the abort_statement completes. However, the execution of the aborted construct completes no later than its next *abort completion point* (if any) that occurs outside of an abort-deferred operation; ... **§9.8**

Abort completion points are listed in §9.8(16–19); examples are the start or end of an entry call or accept statement.

A task that is in an entry queue is immediately aborted, because the task has yet to affect the accepting task or protected object. The question now arises: What about a task that has executed requeue? On one hand, the task has already started a protected action and should be allowed to complete it. On the other hand, the task could be indefinitely blocked and presumably we had a good reason to abort it. The language does not attempt to choose between these alternatives; instead, when you write a requeue statement, you can choose whether to allow the abort to cancel the call, or whether the call should be protected against cancellation §9.5.4(13–16). In the CEO

program, the abort statement is used ‡185,188,191 when all employees are being fired and the company shut down, so we choose to allow cancellation by specifying **with abort** on the requeue statement ‡34,38,51.

Task attributes

The attribute T'Callable §9.9(2) can be used to check if an entry of task T can be called, that is, if the task is not completed or abnormal. Similarly, T'Terminated §9.9(3) checks if T is terminated or not.

14.2 Exceptions

There are special rules for exceptions that occur in multitasking programs. An exception in a task should not affect another task:

3	When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is *abandoned*; ... Then:	**§11.4**
4	If the construct is a task_body, the exception does not propagate further;	

It is good practice to include an exception handler in every task body; otherwise, an exception will cause the task to terminate silently while the rest of the program continues to execute.

If two tasks are engaged in a common action such as a rendezvous, both tasks should be allowed to handle the exception:

26	If an exception is raised during the execution of an entry_body, it is propagated to the corresponding caller (see 11.4).	**§9.5.3**

The predefined exception Tasking_Error is used to signal inconsistencies in the tasks of a program; in particular, calling a task that has already completed or become abnormal raises Tasking_Error §9.5.3(21). The exception is also raised in case of problems during task activation; see §9.2(5) for details. Certain catastrophic errors cause Program_Error to be raised: an exception in the evaluation of a barrier §9.5.3(17), and a selective accept statement with no open alternatives §9.7.1(21). You should keep barrier and guard expressions very simple so that you can prove that these problems will not occur.

14.3 Time

So far we have discussed concurrency in terms of arbitrary interleaving of execution sequences. Most real programs, of course, will have timing constraints. In this section, we present the core concepts of time in Ada; extensions for real-time programming are discussed in Section 16.3.

There are two concepts that must be clearly distinguished: a *point* in time given by the private type Time declared in package Ada.Calendar §9.6(8,10), and an *interval* between two points in time is

given by the predefined fixed point type Duration §10.1(43) and §9.6(7) (Figure 14.1). The function Clock §9.6(12,23) returns the current time, and package Ada.Calendar contains subprograms §9.6(13–15, 24–25) for decomposing a value of type Time into year, month, day and seconds, and conversely for creating a value of type Time from these values.

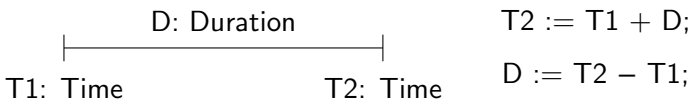

Figure 14.1: Time and Duration

The types Time and Duration are closely connected, in that appropriate arithmetical and relational operations are defined on them §9.6(16–17). For example:

```
function "-"(Left: Time; RIght: Time) return Duration;
```

gives the interval between two points of time.

Values of types Time and Duration are used to specify delays.

§9.6

1 A delay_statement is used to block further execution until a specified *expiration* time is reached. The expiration time can be specified either as a particular point in time (in a delay_until_statement), or in seconds from the current time (in a delay_relative_statement). ...

2 delay_statement ::=
delay_until_statement | delay_relative_statement

3 delay_until_statement ::= **delay until** *delay*_expression;

4 delay_relative_statement ::= **delay** *delay*_expression;

21 The task executing a delay_statement is blocked until the expiration time is reached, at which point it becomes ready again. If the expiration time has already passed, the task is not blocked.

A task is unblocked upon expiration of the delay, but there is no guarantee that the task is scheduled immediately. In other words, the delay is a lower bound on the interval that the task will not be running. Even if the task is not blocked because the delay is zero or negative, the statement is meaningful: it is an abort completion point §9.8(18), and it can cause a context switch. In fact, **delay** 0.0 is a convenient way to call the scheduler.

14.4 Periodic tasks

Control algorithms are implemented by periodic tasks that use delay statements to execute subprograms at predetermined time intervals, rather than ‘as fast as possible’.

Case study: periodic task with delay

The following program prints '*' every 0.2 seconds. Clock is called ‡12 to obtain the current time and then the Interval of type Duration ‡11 is added to obtain the Time of the Next execution of the periodic task. The program executes a delay_until_statement with this time; upon expiration of the delay, the 'algorithm' is executed ‡17 and then the Next wakeup time is computed ‡18.

```
1   --                                                    -- File: PERIOD
2   --  Periodic task.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  with Ada.Calendar; use Ada.Calendar;
6  procedure Period is
7    Start, Stop: Time;
8
9    task Periodic;
10   task body Periodic is
11     Interval: constant Duration := 0.2;
12     Next: Time := Clock + Interval;
13   begin
14     Start := Clock;
15     for N in 1..50 loop
16       delay until Next;
17       Put('*');
18       Next := Next + Interval;
19     end loop;
20     Stop := Clock;
21   end Periodic;
22
23 begin
24   loop
25     exit when Periodic'Terminated;
26     delay 0.0;
27   end loop;
28   New_Line;
29   Put_Line("Elapsed time = " & Duration'Image(Stop-Start));
30 end Period;
```

Additional tasks could be declared to run in the background while the periodic task is blocked. The delay_until_statement ensures that the program is self-synchronizing, regardless of the variability of the computation represented by the Put statement. Furthermore, even if the periodic task is not immediately scheduled after the delay has expired, a shorter delay will be computed on the next period.

You do not want to replace the delay_until_statement with the delay_relative_statement:

```
delay Interval;
```

The periodic task will then be executed once every Interval seconds *plus* the number of seconds it takes to execute the computation *plus* the number of seconds that the task is ready but not scheduled!

It is less obvious, but you also don't want to write:

```
delay Next-Clock;
```

because a race condition could occur. Suppose that the task were preempted *after* evaluating Clock, but *before* evaluating the subtraction. The interval during which the task was blocked would not be taken into account in the computation of the delay.

Implementation of Time and Duration**

As with all fixed point types, there is a tradeoff between range and precision in the implementation of Duration. The requirements are as follows:

27	The implementation of the type Duration shall allow representation of time intervals (both positive and negative) up to at least 86400 seconds (one day); Duration'Small shall not be greater than twenty milliseconds. ...	**§9.6**
30	Whenever possible in an implementation, the value of Duration'Small should be no greater than 100 microseconds.	

Twenty milliseconds is achievable using the 50–60 Hz frequency of ordinary alternating-current electricity. However, almost all computers have an electronic time base, so the 100 microsecond precision should be easy to implement. Time can represent at least all dates between the years 1901 and 2099 §9.6(11) with a precision given in seconds by System.Tick §13.7(30).

There is an additional tradeoff between efficiency and the rate at which the clock is updated.

23	The time base associated with the type Time of package Calendar is implementation defined. The function Clock of package Calendar returns a value representing the current time for this time base. The implementation-defined value of the named number System.Tick (see 13.7) is an approximation of the length of the real-time interval during which the value of Calendar.Clock remains constant.	**§9.6**
35	There is no necessary relationship between System.Tick (the resolution of the clock of package Calendar) and Duration'Small (the *small* of type Duration).	

Duration might have enough precision to store time intervals down to single microseconds, but for efficiency the clock may be 'ticked' only once every 50 microseconds.

A time base is *monotonic* if the value of Clock never decreases. Clock may not be if the computer system clock is adjusted because the system is flown from one time zone to another or because of daylight savings time.

31	The time base for delay_relative_statements should be monotonic; it need not be the same time base as used for Calendar.Clock.	**§9.6**

Time and Duration have relatively large ranges with reasonable precision and are sufficient for 'ordinary' timing requirements, such as financial computations and tasks with periods in the tens of milliseconds. More precise timing is available if your system supports Annex §D 'Real-Time Systems' (Section 16.1).

14.5 Timed and conditional entry calls

Delays can be used to implement polling. In this programming technique, rather than having a task block while waiting for an event to occur, the task periodically checks for the occurrence of the event. The event that we can check for is to see if an entry call can be accepted either immediately, or within a specified period of time; if not, the task performs some alternate computation. Be careful not to confuse this construct with the selective accept statement, which has a similar syntax (Sections 13.3 and 14.7). A timed or conditional entry call is used in the calling task, not the accepting task; the calling task cannot block waiting for one of several calls to be accepted.

§9.7.2

2 timed_entry_call ::=
select
entry_call_alternative
or
delay_alternative
end select;

§9.7.2

3 entry_call_alternative ::=
entry_call_statement [sequence_of_statements]

4 For the execution of a timed_entry_call, the *entry*_name and the actual parameters are evaluated, as for a simple entry call (see 9.5.3). The expiration time (see 9.6) for the call is determined by evaluating the *delay*_expression of the delay_alternative; the entry call is then issued.

5 If the call is queued (including due to a requeue-with-abort), and not selected before the expiration time is reached, an attempt to cancel the call is made. If the call completes due to the cancellation, the optional sequence_of_statements of the delay_alternative is executed; if the entry call completes normally, the optional sequence_of_statements of the entry_call_alternative is executed.

By specifying a zero delay, the call is cancelled if it cannot be immediately accepted. There is a special syntax for this case, called a *conditional entry call* that uses the reserved word **else** instead of **or**.

Mixing a timed or conditional entry call with the entry attribute E'Count can be dangerous §9.9(7–8). If you write a guard like **when** E'Count>0, it is possible that between the time that the accepting task evaluates E'Count and the time it executes the accept statement, the task on the queue was cancelled. This can lead to deadlock if the cancelled task was the only caller.

Case study: periodic task with conditional entry call

Polling is demonstrated in the following program, where task User represents a background computation concurrent with the periodic task. The program prints '*' every 0.2 seconds unless a key is pressed.

-- File: COND

```
1   --                                                          -- File: COND
2   -- Conditional entry call.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  with Ada.Calendar; use Ada.Calendar;
6  procedure Cond is
7
8     task User is
9        entry Trigger;
10    end User;
11    task body User is
12       C: Character;
13       Available: Boolean;
14    begin
15       loop
16          Get_Immediate(C, Available);
17          if Available then
18             accept Trigger;
19             New_Line;
20             Put_Line("User trigger");
21             exit;
22          else
23             delay 0.0;
24          end if;
25       end loop;
26    end User;
27
28    Period: constant Duration := 0.2;
29    Next: Time := Clock + Period;
30 begin
31    for N in 1..50 loop
32       select
33          User.Trigger;
34          exit;
35       else
36          delay until Next;
37          Put('*');
38          Next := Next + Period;
39       end select;
40    end loop;
```

```
41    if not User'Terminated then abort User; end if;
42 end Cond;
```

Before each iteration ‡31–40, the periodic task uses a conditional entry call ‡32–39 to check if task User is willing to accept a call on the entry Trigger ‡9. The User task calls Get_Immediate ‡16, a non-blocking input procedure §A.10.7(11–12). If you press a key, the body of the if-statement is executed and the task blocks on the accept statement ‡18. A rendezvous with the conditional call ‡33 will take place during the next iteration of the periodic task. If you do not press a key, User will execute indefinitely, so we abort it ‡41 when the periodic task completes. The delay statement ‡23 is used to ensure that the task reaches an abort completion point §9.8(15–19) so that the abort can take effect.

14.6 Asynchronous transfer of control*

The problem with the example in Section 14.5 is that the periodic task must explicitly poll the user task to check for the trigger. A preferable solution would be to have the user task interrupt the execution of the periodic task when the triggering event occurs. This can be done with an asynchronous transfer of control (ATC) §9.7.4.

Case study: periodic task with ATC

```
1    --                                                    -- File: ASYNC
2    -- Asynchronous transfer of control.
3    --
4  with Ada.Text_IO; use Ada.Text_IO;
5  with Ada.Calendar; use Ada.Calendar;
6  procedure Async is
7
8    task User is
9      entry Trigger;
10   end User;
11   task body User is
12     C: Character;
13     Available: Boolean;
14   begin
15     loop
16       Get_Immediate(C, Available);
17       if Available then
18         accept Trigger;
19         New_Line;
20         Put_Line("User trigger");
21         exit;
```

```
22          else
23             delay 0.0;
24          end if;
25       end loop;
26    end User;
27
28    Period: constant Duration := 0.2;
29    Next: Time := Clock + Period;
30 begin
31    select
32       User.Trigger;
33       Put_Line("Triggering alternative taken");
34    then abort
35       for N in 1..50 loop
36          delay until Next;
37          Put('*');
38          Next := Next + Period;
39       end loop;
40       abort User;
41    end select;
42 end Async;
```

The ATC construct is composed of two parts: the abortable part ‡35–40 and the triggering alternative ‡32–33. If the triggering alternative, (here an entry call) completes normally, the abortable part is aborted. This will happen if you press a key so that the call User.Trigger ‡32 is accepted ‡18. Otherwise, the abortable part is executed to completion.

The triggering alternative can also be a delay statement; when the delay expires, the abortable part is aborted. The precise semantics of this construct are explored in Quiz 66.

Asynchronous transfer of control has many uses. It can be used to interrupt a computation when an external event occurs, or to implement a ‘watchdog’ that terminates a computation that is apparently non-terminating. In a real-time system, it can be used to allow as much time as is available to perform a difficult computation, while using a partial result when a time interval expires or an event occurs.

14.7 Alternatives for selective accept

We have seen that a selective accept statement may have a terminate alternative. It may also have a delay alternative or an else-part, which can be used to implement timeouts or polling in an accepting task §9.7.1(10–11). For example, we could modify the CEO task to include a delay alternative:

```
1       select                                                    -- File: CEOD
2         accept Wake(Finance_Group);
3           ...
4       or
5         when Wake(Finance_Group)'Count = 0 =>
6           accept Wake(Marketing_Group);
7             ...
8       or
9         when Wake(Finance_Group)'Count = 0 and
10            Wake(Marketing_Group)'Count = 0 =>
11          accept Wake(Engineering_Group);
12            ...
13      or
14        delay 1.0;
15        Put_Line("Flying to new golf course");
16      end select;
```

If no group calls one of the Wake entries within one second, the CEO flies away to a try a new golf course. Alternatively, delay-until could be used to have the CEO wait until three o'clock and then fly away.

A real application of this technique would be an alarm system that receives periodic messages from a sensor. If no message is received within a given time interval, the system can assume that the line has been cut and can raise an alarm.

An else-part can be used to allow the accepting task to perform a computation if there are no calling tasks that can be immediately accepted. If all alternatives are closed and there is an else-part, Program_Error will not be raised §9.7.1(21). A terminate alternative, one or more delay alternatives and an else-part are mutually exclusive §9.7.1(12). You will want to read §9.7.1 carefully if you plan to use these constructs.

14.8 Case study: concurrent simulation

This section presents a concurrent version of the discrete event simulation. Instead of creating all the events before running the simulation, the two parts of the program will run concurrently. In fact, each subsystem will create events concurrently. The program will be sufficiently general so that several independent simulations could be run within the same program.

Figure 14.2 shows the simulation framework. As before, package Root_Event provides the abstract tagged type Event and the abstract procedure Simulate. A 'simulation' is defined by a child package Root_Event.Simulation that declares a Producer type. This type contains the data associated with a single simulation: a Queue (with a protected component Lock to synchronize access by the event creators and the task doing the simulation), a random number Generator and an indication of the Latest simulated time already used, so that a monotonic sequence of events can be created. The interface to the queue is through subprograms Put and Get that take parameters of type Producer, rather than of the encapsulated Queue.

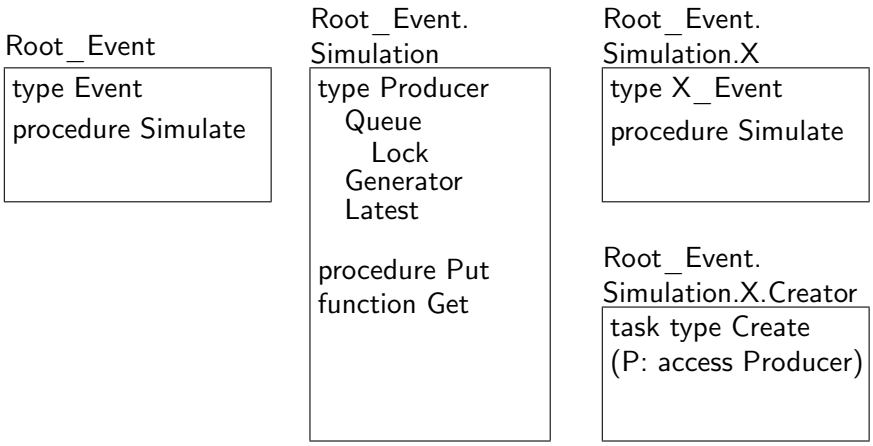

Figure 14.2: Concurrent simulation framework

For each subsystem X to be simulated, a child package Root_Event.Simulation.X is declared containing the derived events and the overridden Simulate procedure. A further child package declares a type for a task that Create's events. This task takes an access discriminant to the Producer object of this simulation so that it knows where to enqueue the event.

We now specialize the simulation framework for the rocket. The usual derived types and simulation subprograms are declared, followed by package Rocket_Simulation (Figure 14.3). This package contains a Producer object and Create objects (tasks) for each derived type. The tasks create objects concurrently and enqueue them.

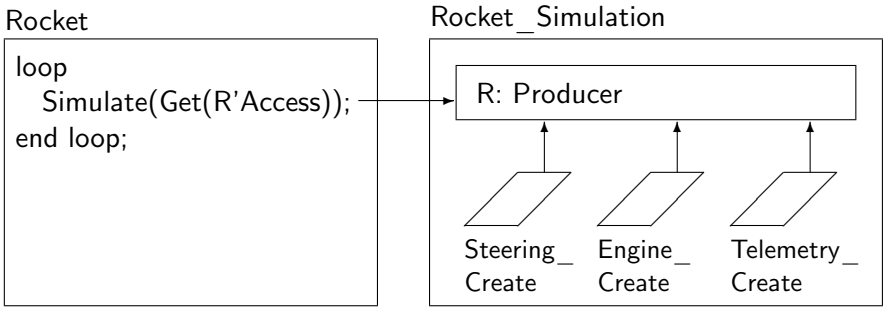

Figure 14.3: Concurrent rocket simulation

The main subprogram Rocket contains the standard simulation loop. It removes events one-by-one from the producer R and dispatches to the appropriate Simulate procedure. The creators are 'free-running', using delay statements to regulate the rate of event creations; the simulation loop runs continuously, blocking on the queue's Lock if the queue is empty.

To generalize to more than one simultaneous simulation, you can declare another simulation package with a Producer object. Instead of a single loop directly within the main subprogram (which is in fact a task), you would place each simulation loop in a separate task.

We now present the essential part of the source code for the concurrent rocket simulation. The most important package is Root_Event.Simulation.

```
1  --                                                    -- File: ROCKETT
2  -- Discrete event simulation of a rocket.
3  -- Creation of events concurrently with simulation.
4  --
5  with Heterogeneous_Priority_Queue;
6  with Ada.Numerics.Discrete_Random;
7  package Root_Event.Simulation is
8    type Producer is limited private;
9
10   procedure Put(E: in Event'Class; P: in out Producer);
11   function Get(P: access Producer) return Event'Class;
12 private
13   package Event_Queue is new
14     Heterogeneous_Priority_Queue(Event'Class);
15   package Random_Time is new
16     Ada.Numerics.Discrete_Random(Natural);
17
18   type Producer is
19     record
20       Queue: aliased Event_Queue.Queue;
21       Generator: Random_Time.Generator;
22       Latest: Simulation_Time := Simulation_Time'First;
23     end record;
24
25   function Random(P: Producer) return Natural;
26   function Random_Update(P: access Producer) return Simulation_Time;
27   -- Return a random time greater than Latest and update Latest
28 end Root_Event.Simulation;
```

The package exports only the type Producer ‡8 and its two subprograms Put and Get ‡10–11. The type has three fields ‡20–22: a Queue implemented by instantiating a generic package ‡13–14, a random number Generator obtained by instantiating the library package Discrete_Random ‡15–16, and the Latest time component. The generic package Heterogeneous_Priority_Queue (not shown) contains, in addition to the binary tree, a protected type Lock. The private subprograms Random and Random_Update ‡25–26 are used by the (grand-)child packages to create events.

The child packages that declare the event types are familiar except that they contain only the Simulate procedure, not the Create function. Creation of events is done in a Creator child package such as the following one for Steering.

```
29 package Root_Event.Simulation.Steering.Creator is
30   type Create(P: access Producer) is limited private;
31 private
32   task type Create(P: access Producer);
33 end Root_Event.Simulation.Steering.Creator;
34
```

```
35 package body Root_Event.Simulation.Steering.Creator is
36    task body Create is
37       function Create_Event return Steering_Event is ...
38    begin
39       loop
40          Put(Create_Event, P.all);
41          delay 1.0;
42       end loop;
43    end Create;
44 end Root_Event.Simulation.Steering.Creator;
45
```

The package exports a type Create ‡30, which is implemented by a task type ‡32. The access discriminant P is used to pass the Producer object to the task. The task itself simply executes a loop calling Create_Event ‡37 to create a random event and placing the event on the queue encapsulated in the producer ‡40.

Package Rocket_Simulation declares and exports a Producer object R ‡49 and declares and hides the creators ‡57–59. Pragma Elaborate_Body ‡48 is required because there are no declarations in the specification that require completions in a body, but we want the body with the hidden declarations to be elaborated. An access to the producer object R is passed to each creator.

```
46 with Root_Event.Simulation;
47 package Rocket_Simulation is
48    pragma Elaborate_Body;
49    R: aliased Root_Event.Simulation.Producer;
50 end Rocket_Simulation;
51
52 end Root_Event.Simulation.Steering;
53 with Root_Event.Simulation.Telemetry.Creator;
54 with Root_Event.Simulation.Engine.Creator;
55 with Root_Event.Simulation.Steering.Creator;
56 package body Rocket_Simulation is
57    T: Root_Event.Simulation.Telemetry.Creator.Create(R'Access);
58    E: Root_Event.Simulation.Engine.Creator.Create(R'Access);
59    S: Root_Event.Simulation.Steering.Creator.Create(R'Access);
60 begin
61    null;
62 end Rocket_Simulation;
```

Finally, the main subprogram just contains the simulation loop:

```
63 with Root_Event.Simulation;
64 with Rocket_Simulation;
65 procedure RocketT is
66 begin
67   loop
68     Root_Event.Simulate(
69       Root_Event.Simulation.Get(
70         Rocket_Simulation.R'Access));
71   end loop;
72 end RocketT;
```

14.9 Tasks as access discriminants**

An access discriminant can be an access to a task type. The following program shows how access discriminants can be used to configure a set of tasks at run-time. There is one task of type Main_Task ‡13–17, which will call an entry of two worker tasks in succession ‡15–16. Main_Task has two access discriminants Left and Right ‡8, which are initialized with accesses to tasks of type Worker_Task. Note the implicit dereferencing in the entry calls to the worker tasks ‡15–16.

```
1   --                                                    -- File: CONFIG
2   -- Access discriminants used for task configuration.
3   --
4  package Tasks is
5    task type Worker_Task(ID: Character) is
6      entry Input;
7    end Worker_Task;
8    task type Main_Task(Left, Right: access Worker_Task);
9  end Tasks;
10
11 with Ada.Text_IO;
12 package body Tasks is
13   task body Main_Task is
14   begin
15     Left.Input;
16     Right.Input;
17   end Main_Task;
18
```

```
19    task body Worker_Task is
20    begin
21       accept Input do
22          Ada.Text_IO.Put_Line(ID & " called");
23       end Input;
24    end Worker_Task;
25 end Tasks;
26
27 with Tasks;
28 procedure Config is
29    W1: aliased Tasks.Worker_Task('A');
30    W2: aliased Tasks.Worker_Task('B');
31    M: Tasks.Main_Task(W2'Access, W1'Access);
32 begin
33    null;
34 end Config;
```

Here the main task M ‡31 is declared as an object in the main subprogram and initialized with worker tasks W1 and W2 ‡29–30. In a real application, the main task would be dynamically allocated and the worker tasks would be determined at run-time.

Simulating dispatching on entries

Access discriminants can be used to simulate dispatching on entry calls. The following program demonstrates how an access value to a task can be obtained from a heterogeneous data structure, and used to call an entry (Figure 14.4).

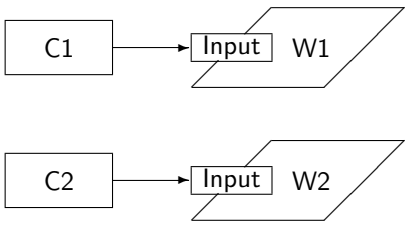

Figure 14.4: ‘Dispatching’ on task entries

Two task types Worker_1 and Worker_2 ‡5–7, 10–12 are declared, as well as access types W1_Ptr and W2_Ptr ‡8, 13 to the task types. The tasks just print their identifications.

```
1  --                                                    -- File: DISPTASK
2  -- Access discriminants for "dispatching" to an entry.
3  --
4  package Tasks is
5     task type Worker_1 is
6        entry Input;
7     end Worker_1;
```

```
 8    type W1_Ptr is access Worker_1;
 9
10    task type Worker_2 is
11       entry Input;
12    end Worker_2;
13    type W2_Ptr is access Worker_2;
14 end Tasks;
15
16 with Ada.Text_IO;
17 package body Tasks is
18    task body Worker_1 is
19    begin
20       accept Input do
21          Ada.Text_IO.Put_Line("Worker 1");
22       end Input;
23    end Worker_1;
24
25    task body Worker_2 is
26    begin
27       accept Input do
28          Ada.Text_IO.Put_Line("Worker 2");
29       end Input;
30    end Worker_2;
31 end Tasks;
```

Types W1_Channel and W2_Channel ‡40–41,44–45 contain an access discriminant to the respective task type. These types are derived from the tagged type Channel ‡33, and override ‡42,46 the abstract dispatching subprogram Output ‡34. This subprogram does nothing more than call an entry of the task pointed to by the access discriminant ‡52,57.

```
32 package Channels is
33    type Channel is abstract tagged limited null record;
34    procedure Output(C: access Channel) is abstract;
35    type Channel_Ptr is access all Channel'Class;
36 end Channels;
37
38 with Tasks;
39 package Channels.Workers is
40    type W1_Channel(W: access Tasks.Worker_1) is
41       new Channel with null record;
42    procedure Output(C: access W1_Channel);
43
44    type W2_Channel(W: access Tasks.Worker_2) is
45       new Channel with null record;
46    procedure Output(C: access W2_Channel);
47 end Channels.Workers;
```

```
48
49 package body Channels.Workers is
50    procedure Output(C: access W1_Channel) is
51    begin
52       C.W.Input;
53    end Output;
54
55    procedure Output(C: access W2_Channel) is
56    begin
57       C.W.Input;
58    end Output;
59 end Channels.Workers;
```

In the main subprogram, two workers W1 and W2 are dynamically allocated ‡63–64, as well as two channels—one for each worker task. These are stored in a 'heterogenous data structure', here just two variables C1 and C2 of access to *class-wide* type ‡65–66. The declaration of Get_Channel ‡67 stands for removing an arbitrary channel channel from the data structure. Procedure Output, whose formal parameters are of access to a tagged type, is called with an actual parameter of access to the class-wide type ‡69. By §3.9.2(2), this is a dispatching call on the designated type. Whichever version of Output that is dispatched to now calls the entry for the task pointed to by the channel, so in effect we have 'dispatched' an entry call.

```
60 with Tasks;
61 with Channels.Workers; use Channels;
62 procedure DispTask is
63    W1: Tasks.W1_Ptr := new Tasks.Worker_1;
64    W2: Tasks.W2_Ptr := new Tasks.Worker_2;
65    C1: Channel_Ptr := new Workers.W1_Channel(W1);
66    C2: Channel_Ptr := new Workers.W2_Channel(W2);
67    Get_Channel: Channel_Ptr := C2;
68 begin
69    Output(Get_Channel);
70 end DispTask;
```

See Ben-Ari (1996*c*) and Ben-Ari (1998*b*) for case studies that use these techniques.

15 Systems Programming*

15.1 Implementation dependencies

Our discussion has focused on Ada as a formalism for writing programs: syntax, semantics and stylistics. However, Ada is firmly rooted in the requirements of computer systems with their hardware, operating systems and libraries, and in the requirements of projects in terms of performance, reliability and reuse of existing subsystems.

The design of a programming language for use in real projects must cope with conflicting requirements:

- If the language is small, implementation-specific extensions and 'third-party' add-ons will be needed, making the software non-portable. If the language is large, it may be too expensive or even impossible to implement completely for important target computers and operating systems.
- If the language specification is too general, implementations will fill in the details as they see fit and the software developer will not be able to rely on a portable behavior. If the language specification is extremely detailed, again there may be difficulty implementing the language as specified on a target architecture.

The Ada approach to implementation dependency can be summarized as follows:

> Certain features in the language need not be implemented, but if you do implement them, this must be done as described in the standard. Where the standard leaves a decision up to the implementation, the decision must be documented.

The documentation provides the information you need in order to choose an implementation that satisfies the requirements of your project. The standard specification of 'optional' features means that programming techniques and even existing source code can be easily adapted to a new implementation. Furthermore, little retraining is necessary for software engineers moving from one implementation to another, because the concepts, terminology and even the type and subprogram declarations will be almost identical across all implementations.

There are two levels of implementation dependency in Ada. First, there are six 'Specialized Needs Annexes':

- Annex C Systems Programming
- Annex D Real-Time Programming
- Annex E Distributed Systems
- Annex F Information Systems

- Annex G Numerics
- Annex H Safety and Security

An implementation need not provide any of these annexes. The marketing literature for an implementation will almost certainly list which annexes are supported and which are not.

The second level of support for optional features is the freedom granted to an implementation. These freedoms appear throughout the *ARM*, many in paragraphs entitled 'Implementation Permissions' and 'Implementation Advice'. In addition, the annexes contain paragraphs entitled 'Documentation Requirements' that give the detailed information you need about individual constructs. A centralized list of 136 items appears in Annex §M 'Implementation-Defined Characteristics'. Once you have chosen candidate implementations that support the annexes your project requires, you can further compare levels of support for individual features.

In time-critical systems, you need to be able to predict the performance of the software. Paragraphs entitled 'Metrics' require the implementation to document performance characteristics, mostly bounds on execution time in terms of processor cycles. While this information may not be exact, it can be extremely useful when you are designing the program and choosing the language constructs to be used. For example, you may want to compare the overhead of an entry call of a protected object with that of a task.

Information Systems were discussed in Section 10.7 and Numerics outlined in Section 10.8. This chapter will present the other Specialized Needs Annexes, as well as Annex §B 'Interface to Other Languages'. Since this material is mostly new for Ada 95, the *Rationale* for the annexes is quite extensive. When you study an annex in the *ARM*, you will also want to study the corresponding section in the *Rationale* that discusses intentions, justifications, tradeoffs and examples.

15.2 Annex B Interface to Other Languages

Very few software systems are developed in a vacuum. Your program will almost certainly have to call operating system services and subprograms from libraries; it may also need to be integrated with existing 'legacy' code. Since these subsystems may have been written in other languages, Ada supplies facilities for *mixed-language* programming.

There are two technical problems that must be solved:

- The representation of similar types may differ from one language to another. Two famous examples are the use of null-terminated strings in C instead of an array plus current length used in other languages, and the storage of multi-dimensional arrays in Fortran in column-major order rather than in row-major order.
- Subprogram naming and calling conventions. For example, C++ uses a language-specific encoding of the external names of subprograms called 'name mangling', to perform typesafe linkage, that is, checking the profiles of function calls at link-time.

In Ada, there are two approaches to solving these problems:

- You can specify a *convention* for a type, object or subprogram, requesting that the representation of the entity be appropriate for the specified language. Computer vendors usually share conventions among languages that they support, often by re-using existing code generators.
- Language-specific child packages of Interface declare Ada types that are represented in the same way as types in other languages. You can declare an Ada object that is a null-terminated C string, convert an Ada string to this object and then pass it to a system service that expects parameters in the C convention.

Interfacing pragmas

> 1 A pragma Import is used to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada. In contrast, a pragma Export is used to export an Ada entity to a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language. The pragmas Import and Export are intended primarily for objects and subprograms, although implementations are allowed to support other entities. **§B.1**
>
> 2 A pragma Convention is used to specify that an Ada entity should use the conventions of another language. It is intended primarily for types and "callback" subprograms. For example, "**pragma** Convention(Fortran, Matrix);" implies that Matrix should be represented according to the conventions of the supported Fortran implementation, namely column-major order.

Predefined conventions, such as the default convention Ada, are defined in §6.3. An implementation may define other conventions, such as language-specific conventions. The pragmas Import and Export take two required parameters—the convention and the entity it applies to—and one of two optional parameters, either an External_Name §B.1(34) or a Link_Name §B.1(35):

```
pragma Import(C, CFunc, Link_Name => "_cfunc");
```

If the system can guess the link name of the imported entity from the local name, neither is needed; if not, the system may be able to deduce it from the external name in the foreign language; finally, you can give the exact name expected by the linker. Obviously, each option in this sequence is less portable than the previous one.

> 22 A pragma Import shall be the completion of a declaration. ... **§B.1**

In particular, it can be the completion of a subprogram or a deferred constant. For example:

§B.1

```
51  package Fortran_Library is
      function Sqrt (X : Float) return Float;
      function Exp (X : Float) return Float;
    private
      pragma Import(Fortran, Sqrt);
      pragma Import(Fortran, Exp);
    end Fortran_Library;
```

An Ada subprogram can be exported and called from a foreign language; in fact, the main program may be in the foreign language. Implementations are advised to supply two subprograms adainit and adafinal that can be called from the foreign main program to perform elaboration of Ada library units and finalization of the environment task §B.1(39).

Package Interfaces

Types and subprograms for interfacing to other languages are contained in package Interfaces. The package itself §B.2 contains declarations of numerical types (signed and modular integer types, and floating point types) directly supported by the target computer, as well as shift and rotate instructions for the integer types. Clearly, using these types will tend to make the program non-portable and they should only be used to declare objects that are passed directly to the hardware.

An implementation may provide interfaces to other programming languages as child packages of Interfaces §B.2(11). Standard interfaces for C, COBOL and Fortran are declared in this annex. These interfaces declare types corresponding to each of the types in the foreign language, as well as functions To_*Lang* and To_Ada for converting the types of language *Lang* to and from Ada.

C

Package Interfaces.C §B.3 contains declarations of types corresponding to C's types such as int and unsigned_char. char_array is a character array, and the conversion functions To_Ada and To_C can deal with the null terminator §B.3(47–56). The suggested correspondence between Ada subprograms and parameters and those of C is given in §B.3(63–71). For example, a **void**-returning C function corresponds to an Ada procedure, and a parameter of type T* corresponds to an **in out** parameter of type T in Ada.

String processing in C is usually done on dynamically allocated objects rather than on static arrays. The following two declarations give rise to the data structures shown in Figure 15.1.[1] The declaration of s1 corresponds to a value of type char_array.

```
char s1[] = "Hello world";
char *s2 = "Hello world";
```

[1] The figure is reproduced from Ben-Ari (1996*b*, p. 192).

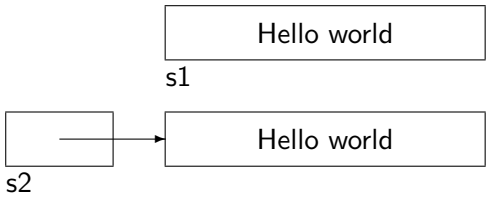

Figure 15.1: Array versus pointer in C

Package Interfaces.C.Strings declares a private type chars_ptr for pointers to C strings of type char* such as s2 in the figure, and chars_ptr_array for arrays of pointers. The Ada program can manage the storage for these strings. New_Char_Array §B.3.1(25–28) and New_String §B.3.1(29–30) allocate a C string initialized by an existing value of type char_array and String, respectively. Free §B.3.1(31–32) releases the storage.

Given an (aliased) object S of type char_array, To_Chars_Ptr §B.3.1(23–24) returns a value of type chars_ptr pointing to S'Access. The package also contains subprograms for the reverse conversion and subprograms for directly updating C strings.

If you are really homesick for C programming, generic package Interfaces.C.Pointers §B.3.2 declares type Pointer as an access to a generic formal parameter, and provides subprograms for pointer arithmetic and copying arbitrary arrays!

COBOL

2	The COBOL interface package supplies several sets of facilities:	**§B.4**
3	A set of types corresponding to the native COBOL types of the supported COBOL implementation (so-called “internal COBOL representations”), allowing Ada data to be passed as parameters to COBOL programs	
4	A set of types and constants reflecting external data representations such as might be found in files or databases, allowing COBOL-generated data to be read by an Ada program, and Ada-generated data to be read by COBOL programs	
5	A generic package for converting between an Ada decimal type value and either an internal or external COBOL representation	

For details, see the *ARM*.

Recall that decimal fixed point arithmetic is supported in the Ada language (Section 10.7), and picture editing is supported in Annex §F ‘Information Systems’ (Section 10.7), so you do not need to write COBOL code to obtain this functionality.

Fortran

Package Interfaces.Fortran §B.5 declares types corresponding to Fortran types such as double precision and logical. There is very little functionality in the Fortran language that does not exist in Ada, so the primary use of this interface will be to use numerical and scientific libraries, and to

integrate legacy software. Pragma Convention is particularly useful in this case, as demonstrated by the following example from the *ARM*:

§B.5

```
30  type Fortran_Matrix is array (
      Integer range <>, Integer range <>) of Double_Precision;
    pragma Convention (Fortran, Fortran_Matrix);
    -- stored in Fortran's column-major order
    procedure Invert(
      Rank: in Fortran_Integer; X: in out Fortran_Matrix);
    pragma Import(Fortran, Invert);              -- a Fortran subroutine
```

15.3 Annex C Systems Programming

Annex §B describes interface capabilities at the applications software level. This annex specifies capabilities needed to interface with the hardware and the underlying operating system (if any). Many of the features needed for hardware interface are specified in §13 'Representation Issues'. The core language does not require §13.1(20) that the implementation actually support the features described in §13. but this permission is not available if Annex §C is supported.

§C.2

> 2 The implementation shall support at least the functionality defined by the recommended levels of support in Section 13.

The recommended level of support is specified in the 'Implementation Advice' paragraphs in §13.

Discard_Names

Some Ada constructs have a string associated with them so that they can be displayed:

- The attributes Image and Value, and IO for enumeration types.
- Ada.Exceptions.Exception_Name §11.4.1(5).
- Ada.Tags.Expanded_Name §3.9(10).

Pragma Discard_Names §C.5 allows the implementation to save storage by not storing these strings at run-time; in this case, the null string will probably be returned by these functions. This feature is important for saving memory in embedded computer systems that have no display device, or no use for the constructs listed above. When I applied Discard_Names to the rocket simulation program, the size of the object files was reduced from 39,085 to 38,012 bytes.

Preelaboration

Embedded computer systems need the ability to quickly restart a program in case of a failure such as loss of power. In Ada, the time to start a program includes the elaboration time, which may

be significant. Elaboration time can be reduced by *preelaborating* as much of the program as possible. Furthermore, an implementation can store the constants of a preelaborated unit in ROM.

§10.2.1(2–12) specify restrictions that a library unit must satisfy for it to be preelaborable. Roughly, a unit can be preelaborated if no 'code' need be executed at run-time. For example, tasks are illegal because they give require run-time structures that must be initialized. However, the core language does not specify what 'privileges' a preelaborable unit has—only that it is elaborated after a pure unit and before an ordinary unit.

Annex §C imposes requirements on the implementation of preelaborated units.

2 The implementation shall not incur any run-time overhead for the elaboration checks of subprograms and protected_bodies declared in preelaborated library units. 3 The implementation shall not execute any memory write operations after load time for the elaboration of constant objects declared immediately within the declarative region of a preelaborated library package, so long as the subtype and initial expression (or default initial expressions if initialized by default) of the object_declaration satisfy the following restrictions. The meaning of *load time* is implementation defined.	**§C.4**

The restrictions given in §C.4(4–11) are intended to ensure that expressions are static.

15.4 Hardware interfacing

Machine code

The architecture of most computers includes specialized instructions; use of these instructions can be essential when implementing time-critical algorithms. Of course, hardly anything makes a program less portable than use of machine code!

2 The implementation shall support machine code insertions (see 13.8) or intrinsic subprograms (see 6.3.1) (or both). Implementation-defined attributes shall be provided to allow the use of Ada entities as operands. 4 The interfacing pragmas (see Annex B) should support interface to assembler; ...	**§C.1**

Suppose that the machine includes an atomic increment instruction. To implement a concurrent algorithm, it may be essential that this instruction be used rather than an assignment X:=X+1. There are three ways in which this could be supported.

The implementation could simply allow to program to call an assembly language subprogram. The problem with this method is that there may be significant overhead associated with subprogram call and return.

A better method is to have the implementation simply supply an intrinsic subprogram: Inc(X).

> 4 The *Intrinsic* calling convention represents subprograms that are "built in" to the compiler. ... **§6.3.1**

Intrinsic subprograms are probably the best solution when there are a few commonly used machine-code instructions. However, to access the full set of instructions and addressing modes, support for *machine code insertions* §13.8 is a better solution. The implementation supplies a package System.Machine_Code that defines types that allow qualified expressions to be written for each instruction and addressing mode. Attributes must also be defined for addressing ordinary objects of the Ada program. For example, if an object is addressed by a page number and an offset, the implementation could support a statement like:

```
Instruction'(Inc, X'Page, X'Offset);
```

> 10 The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms. **§C.1**

Interrupts

In this section, we describe a simple model for interrupts and constructs that can be used to implement an interrupt handler. There is an extensive discussion in the *ARM* and the *Rationale* on adapting the constructs to support other architectures. Note in particular the extensive documentation requirements associated with interrupt support §C.3(12–22).

> 2 An *interrupt* represents a class of events that are detected by the hardware or the system software. Interrupts are said to occur. An *occurrence* of an interrupt is separable into generation and delivery. *Generation* of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program. *Delivery* is the action that invokes part of the program as response to the interrupt occurrence. Between generation and delivery, the interrupt occurrence (or interrupt) is *pending*. Some or all interrupts may be *blocked*. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. ... Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be *attached* to that interrupt. The execution of that program unit, the *interrupt handler*, is invoked upon delivery of the interrupt occurrence. **§C.3**

A typical implementation is shown in Figure 15.2. When the interrupt occurs, the hardware looks in a fixed memory location called a *vector* for the address of a subprogram called the *handler*. Delivery of the interrupt consists of preempting the currently executing task and executing the statements of the handler; any needed stack space is 'borrowed' from the current task. Upon completion of the handler, a 'return from interrupt' instruction restores the stack and registers of the preempted task. A bit may be set in a *mask* register to block a pending interrupt.

Interrupts implement a simple form of mutual exclusion: during the execution of the handler, software tasks are blocked because interrupts execute at a higher priority than any software task.

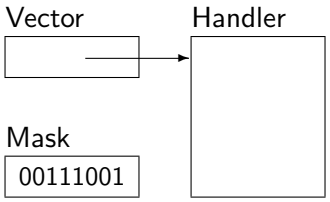

Figure 15.2: Interrupts

If a task wishes to update a variable shared with an interrupt handler, it simply sets the mask register to block the interrupt.

This interrupt model maps onto procedures of protected objects. The interrupt is modelled as an anonymous hardware task executing a protected procedure, and mutual exclusion with other protected operations is implemented by masking the interrupt.

§C.3.1

4 **pragma** Attach_Handler(*handler*_name, expression);
5 ... the *handler*_name shall resolve to denote a protected procedure with a parameterless profile.
10 As part of the initialization of that object, if the Attach_Handler pragma is specified, the *handler* procedure is attached to the specified interrupt. ...
13 When a handler is attached to an interrupt, the interrupt is blocked ... during the execution of every protected action on the protected object containing the handler.

It is also possible to dynamically attach, detach or exchange interrupt handlers using subprograms declared in package Ada.Interrupts §C.3.2. In this case, the pragma Interrupt_Handler §C.3.1(2,9) must be used on the protected procedure instead of Attach_Handler.

15.5 Low-level tasking**

Shared variables

If several tasks are declared within a subprogram or package, they can all read and write variables declared previously in the enclosing declarative region. Normally, sharing variables is not considered a good programming technique, because the variables should be encapsulated in protected objects or tasks. However, low-level systems programming needs this capability. Note that the problem is only in sharing variables; the executable code is assumed not to modify itself ('pure code') and can be executed concurrently by several tasks §6.1(35).

There are two potential problems that can arise from concurrent access to shared variables:

- The variable may not be atomic; this is likely to happen if more than one memory word is needed to store the variable. Suppose that one task updates one word of a two-word variable and then it is preempted by another task that updates both words of the same variable. When the first task is resumed, it will update the second word, leaving the variable with a mixture of two values.

- Code generators in general, and optimizers in particular, make assumptions that may not be valid in the presence of concurrency.

For example, given the statements:

```
X := Y + 4;
Z := X + Y;
```

the code generator might not write to the memory word for X; instead, when computing the expression X+Y, it might use the value of Y+4 that was stored in a register. Then another task might use the old value in X. An extreme case of this problem is shown by the sequence of instructions:

```
Mem := 16#F000#;
Mem := 16#000C#;
```

where the assignment is done solely for the side effect of issuing a command to a memory-mapped peripheral. A 'good' optimizer will simply discard the first assignment.

§9.10 defines what it means for two actions to be *sequential*. Roughly, they are sequential if they are part of the same task, or if they are synchronized by a task rendezvous or protected action. Reading and updating a shared variable is erroneous unless the actions are sequential §9.10(11).

For systems programming, we may want to read and update shared variables without the overhead of rendezvous or protected actions. This is done by using pragmas to specify that an object or all objects of a type are *atomic* or *volatile*.

15	For an atomic object (including an atomic component) all reads and updates of the object as a whole are indivisible.	**§C.6**
16	For a volatile object all reads and updates of the object as a whole are performed directly to memory.	
17	Two actions are sequential (see 9.10) if each is the read or update of the same atomic object.	
8	... every atomic type or object is also defined to be volatile. ...	

Pragma Atomic is simply an assertion that an object can be read and updated indivisibly, and thus can be accessed concurrently §9.10. If the implementation cannot ensure this, the program is illegal §C.6(10). Pragma Volatile (which is also implied by Atomic) affects code generation: no temporary copies will be kept and no reads or updates will be discarded. Volatile can be used on a compound object that cannot be accessed atomically; you will have to use other means to ensure that the object contains a consistent value.

It is also possible to specify that the components of an array are atomic or volatile, even if the entire array is not §C.6(5–6). These pragmas can be applied to a constant object §C.6(13), provided that Import is also applied so that an external program (or perhaps the hardware) can modify it. §C.6(12,18,19) discuss the rules for passing atomic or volatile objects as parameters.

Task identification and attributes

Tasks in Ada are quite flexible: given a task type, you can create a dynamic data structure of tasks and you can associate information with a task by using discriminants, or by creating a record with

a component of the task type. However, since tasks are typed, you cannot create a data structure that contains tasks of arbitrary type, as would be needed for writing an operating system. Package Ada.Task_Identification §C.7.1 declares a nonlimited private type Task_ID that can hold the identification of a task, regardless of its type.

> 5 A value of the type Task_ID identifies an existent task. The constant Null_Task_ID does not identify any task. Each object of the type Task_ID is default initialized to the value of Null_Task_ID. **§C.7.1**
> 8 The function Current_Task returns a value that identifies the calling task.
> 11 For a prefix T that is of a task type (after any implicit dereference), the following attribute is defined:
> 12 T'Identity—Yields a value of the type Task_ID that identifies the task denoted by T.
> 13 For a prefix E that denotes an entry_declaration, the following attribute is defined:
> 14 E'Caller—Yields a value of the type Task_ID that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an entry_body or accept_statement corresponding to the entry_declaration denoted by E.

The function Current_Task §C.7.1(8) is not well-defined within a protected entry body. The reason is that when a blocked task is awakened, the entry body will probably be executed by the awakening task to avoid a context switch.

> 17 It is a bounded error to call the Current_Task function from an entry body or an interrupt handler. Program_Error is raised, or an implementation-defined value of the type Task_ID is returned. **§C.7.1**

To associate data items with all tasks in a program, you instantiate the generic package Ada.Task_Attributes §C.7.2 one or more times to create user-defined attributes. Procedure Set_Value writes to the attribute and function Value reads the attribute.

Case study: task identification and attributes

These packages are demonstrated in the following program, which models a server receiving requests from an arbitrary set of client tasks. At any point in time, the server maintains the Task_ID of the two 'most important' tasks, where 'importance' is an attribute associated with each task. The most important tasks will be released before other tasks.

Importance ‡10 is an instantiation of Ada.Task_Attributes with an attribute of type Integer. Task Server stores the IDs and attributes of the two most important tasks ‡21–22, updating these variables when a request is accepted ‡32–43.

```
1   --                                                          -- File: ID
2   -- Task identification and attributes.
3   --
4  with Ada.Text_IO; use Ada.Text_IO;
5  with Ada.Task_Identification;
6  with Ada.Task_Attributes;
7  with Ada.Numerics.Discrete_Random;
8  procedure ID is
9
10    package Importance is new Ada.Task_Attributes(Integer, 0);
11
12    task Server is
13       entry Request;
14       entry Release;
15    private
16       entry Slow_Release;
17    end Server;
18
19    task body Server is
20       use Ada.Task_Identification;
21       ID1, ID2: Task_ID;
22       Imp1, Imp2: Integer := 0;
23
24       procedure Print(S: String; T: Task_ID) is
25       begin
26          Put_Line(S & Image(T) & Integer'Image(Importance.Value(T)));
27       end Print;
28
```

The attribute E'Caller is used ‡33,38 to obtain the ID of the calling task. When a client calls Release ‡45–50, its ID is compared with the stored ID to see if it is one of the two 'most important' tasks. If so, it is 'processed' by calling Print ‡24–27. Note the use of the function Image §C.7.1(3) to obtain a string uniquely identifying a task. If the task is not important, it is requeued on the private entry[2] Slow_Release ‡52–55, whose guard ensures that it is only executed if no important tasks are being served.

[2]Tasks, not just protected objects, can have private entries.

```
29  begin
30    loop
31      select
32        accept Request do
33          if Importance.Value(Request'Caller) > Imp1 then
34            Imp2 := Imp1;
35            ID2 := ID1;
36            ID1 := Request'Caller;
37            Imp1 := Importance.Value(ID1);
38          elsif Importance.Value(Request'Caller) > Imp2 then
39            ID2 := Request'Caller;
40            Imp2 := Importance.Value(ID2);
41          end if;
42          Print("Request ", Request'Caller);
43        end Request;
44      or
45        accept Release do
46          if Release'Caller /= ID1 and Release'Caller /= ID2 then
47            requeue Slow_Release;
48          end if;
49          Print("Release ", Release'Caller);
50        end Release;
51      or
52        when Release'Count = 0 =>
53          accept Slow_Release do
54            Print("Slow release ", Slow_Release'Caller);
55          end Slow_Release;
56      or
57        terminate;
58      end select;
59    end loop;
60  end Server;
```

Clients set their own importance ‡68,77 by calling a random number generator ‡61–63.

```
61  subtype Numbers is Integer range 1..100;
62  package Random_Numbers is new Ada.Numerics.Discrete_Random(Numbers);
63  G: Random_Numbers.Generator;
64
```

There are two task types ‡65–81, which are used to declare ten client tasks ‡83–84, but the same ID and attributes types are be used by all of them.

```
65    task type Client1;
66    task body Client1 is
67    begin
68       Importance.Set_Value(Random_Numbers.Random(G));
69       Server.Request;
70       delay 0.5;
71       Server.Release;
72    end Client1;
73
74    task type Client2;
75    task body Client2 is
76    begin
77       Importance.Set_Value(Random_Numbers.Random(G));
78       Server.Request;
79       delay 0.5;
80       Server.Release;
81    end Client2;
82
83    C1: array(1..5) of Client1;
84    C2: array(1..5) of Client2;
85 begin
86    null;
87 end ID;
```

16 Real-Time and Distributed Systems*

16.1 Annex D Real-Time Systems

The essence of real-time systems is *predictability*. The software requirements of these systems include reactive time constraints: when an input event occurs, the system must react within a specified time by computing and sending the correct output. Real-time systems frequently need to be very *efficient*—reacting to a large number of events within a very short time—but there is no necessary relationship between the two concepts. Annex §D goes into great detail on two main topics: task scheduling and time. Documentation requirements and metrics are as important as the prescribed language features, because the systems engineer needs this information to design a program that will fulfill the requirements.

> 1 ... To conform to this Annex, an implementation shall also conform to the Systems Programming Annex. **§D**

The reason is that real-time systems invariably control hardware and need the interfacing support described in §13 and §C.

16.2 Scheduling

Recall (Section 14.1, §9(10)) that a *ready* task competes for resources such as processors that it needs to run. However, the core language does not specify how a ready task is chosen if there are more ready tasks than resources. Similarly, the language does not specify how a task blocked on an entry queue is chosen if there is more than one open protected entry §9.5.3(17) or selective accept alternative §9.7.1(16). There could also be more than one expired delay §9.7.1(18). §9.5.3(17) does specify that any particular entry queue will be served in first-in, first-out (FIFO) order of arrival of the calling tasks.

§D.1 through §D.5 describe detailed scheduling rules. For portability, an implementation must support these rules, though it is free to support other scheduling rules needed by applications.

Ada 95 is designed to be 'multiprocessor-friendly', but for simplicity, we will describe the standard rules for a single processor and refer you to the *ARM* and *Rationale* for the modifications needed for multiprocessors.

Priorities

Figure 16.1 shows the queue of ready tasks.[1] Each task has a priority and there is a queue of tasks for each priority. Interrupts have higher priority than tasks.

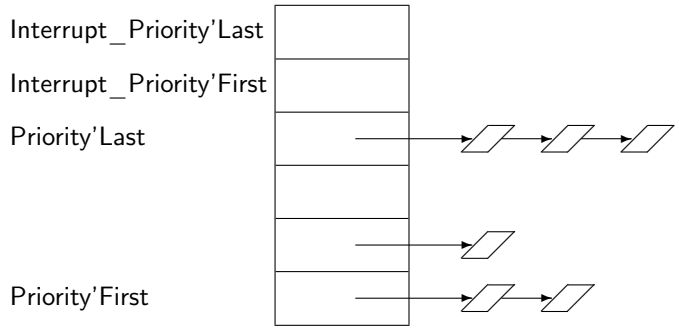

Figure 16.1: Ready queues

Priority is specified by a pragma Priority or Interrupt_Priority in the declaration of a task or protected unit §D.1(3–13). The priority is usually static, but it can depend on a discriminant §D.1(27) so different tasks of the same type can be assigned different priorities. Priorities are used to control task dispatching:[2]

> 4 *Task dispatching* is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called *task dispatching points*. A task reaches a task dispatching point whenever it becomes blocked, and whenever it becomes ready. In addition, the completion of an accept_statement (see 9.5.2), and task termination are task dispatching points for the executing task. Other task dispatching points are defined throughout this Annex. **§D.2.1**
>
> 6 ... Whenever a task running on a processor reaches a task dispatching point, one task is selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

[1] It is important to emphasize that the figure is conceptual; the *ARM* obviously does not specify exactly what data structures an implementation must use.

[2] This use of the term, dispatch, has nothing to do with the concept of dynamic dispatching of subprograms.

Normally, a task will run until it reaches a task dispatching point such as an entry call. Tasks can also be preempted:

7	A preemptible resource is a resource that while allocated to one task can be allocated (temporarily) to another instead. Processors are preemptible resources. Access to a protected object (see 9.5.1) is a nonpreemptible resource. When a higher-priority task is dispatched to the processor, and the previously running task is placed on the appropriate ready queue, the latter task is said to be *preempted*.	**§D.2.1**
8	A new running task is also selected whenever there is a nonempty ready queue with a higher priority than the priority of the running task, or when the task dispatching policy requires a running task to go back to a ready queue. These are also task dispatching points.	

Preemption can occur when a delay expires or when an interrupt causes some blocked task to become ready.

Task dispatching policy

The task dispatching policy concerns the maintenance of the ready queues. The standard policy FIFO_Within_Priorities can be specified by a pragma; if not, the policy is implementation defined §D.2.2(1–6). The policy FIFO_Within_Priorities is illustrated in Figure 16.2. Let us first

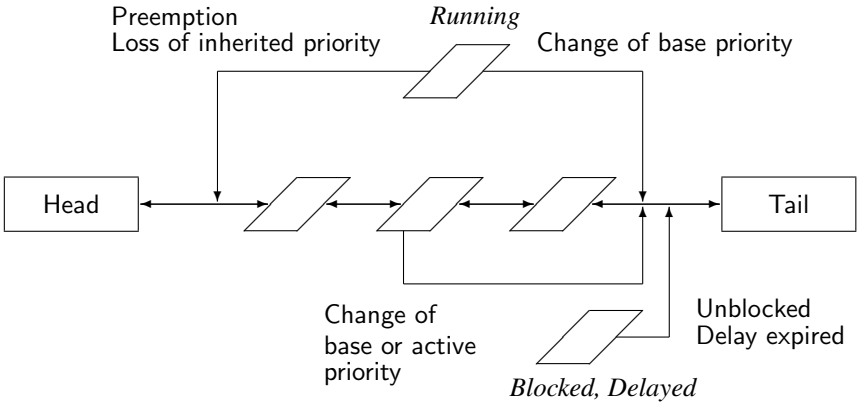

Figure 16.2: FIFO_Within_Priorities policy

consider the transition in the lower right of the figure.

8	When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.	**§D.2.2**

This is reasonable since there are other tasks of the same priority who have been ready, possibly for quite some time. This transition is also taken for a delay statement whose expiration time has already passed; even though the task is not blocked §9.6(21), it is put on the tail of the queue §D.2.2(12) to allow voluntary round-robin scheduling.

One of the transitions in the upper left of Figure 16.2 is given in the following rule:

> 13 In addition, when a task is preempted, it is added at the head of the ready queue for its active priority. **§D.2.2**

If a task is preempted—say by an interrupt—there is no reason to 'punish' it, so it goes back on the head of the queue. Of course, it may not become the next running task, if a higher-priority task has become ready.

The other transitions show what happens when priorities are changed as described in the next subsection.

Base and active priorities

Suppose that task T_1 with priority 1 begins executing an entry E of a protected object PO, and suppose that T_1 is preempted by task T_2 with priority 2. Next, T_2 in turn is preempted by task T_3 with priority 3, which immediately calls the *same* entry E of PO (Figure 16.3). T_3 will be queued

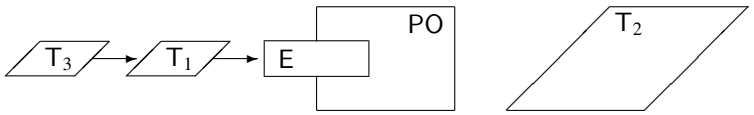

Figure 16.3: Priority inversion

pending the completion of the entry body, but this will not occur as long as T_2 continues to execute because T_2 has a higher priority than T_1! Only when T_2 blocks, perhaps by calling an entry, will it relinquish the processor to T_1 and the entry will be completed. The state of the computation is that a high-priority task T_3 is waiting (and waiting and waiting ...) for a lower-priority task T_2 to block. There are special rules designed to prevent this state, called *priority inversion*.

> 15 ...The *base priority* of a task is the priority with which it was created, ... At all times, a task also has an *active priority*, which generally reflects its base priority as well as any priority it inherits from other sources. *Priority inheritance* is the process by which the priority of a task or other entity (e.g. a protected object; see D.3) is used in the evaluation of another task's active priority. **§D.1**
>
> 20 At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant.
>
> 22 During rendezvous, the task accepting the entry call inherits the active priority of the caller (see 9.5.3).
>
> 23 During a protected action on a protected object, a task inherits the ceiling priority of the protected object (see 9.5 and D.3).

(The meaning of 'ceiling' will be discussed later in this section.)

When a task's priority is reduced by loss of inherited priority (Figure 16.2, top left), the task goes to the head of the ready queue for its priority §D.2.2(9). This can prevent an additional context switch if no higher-priority tasks are ready.

In our example, the protected object would be given a high priority such as 5. The low priority task T_1 would inherit this priority which becomes its active priority; T_2 with priority 2 cannot

execute as long as T_1 is executing the entry body. Upon completion of the entry, T_1 loses its inherited priority and returns to its low base priority. T_3 will execute the entry *before* T_2 is allowed to execute, because it has a higher priority.

Priority inversion still occurs, but it is *bounded* by the maximum duration of an entry body. You can analyze the duration of all protected operations to obtain a bound on the maximum duration of a priority inversion §D.2.2(14–16).

Entry queuing policies

We are not yet finished with priority inversion. If the entry queues are served in FIFO order, a high-priority task could be enqueued behind a long series of lower-priority tasks. This default *queuing policy* is called FIFO_Queuing §D.4(7).

Priority inversion can also occur if there is more than one open accept alternative or entry. Since the choice is not specified by the language §9.5.3(17), a queue with a low-priority task could be served before a queue with a high-priority task. Annex §D specifies an additional queuing policy called Priority_Queuing, which can be chosen by using pragma Queuing_Policy §D.4(2–4).

§D.4

9 The calls to an entry (including a member of an entry family) are queued in an order consistent with the priorities of the calls. The *priority of an entry call* is initialized from the active priority of the calling task at the time the call is made, but can change later. Within the same priority, the order is consistent with the calling (or requeuing, or priority setting) time (that is, a FIFO order).

14 When more than one alternative of a selective_accept is open and has queued calls, an alternative whose queue has the highest-priority call at its head is selected. If two or more open alternatives have equal-priority queued calls, then a call on the entry in the accept_alternative that is first in textual order in the selective_accept is selected.

See §D.4(12–13) for protected entries and delay alternatives.

In the CEO problem (Section 13.4), we specified Priority_Queuing so that entries would be selected in textual order. This prevents the following race condition: the guard **when** Wake-(Finance_Group)'Count = 0 of the alternative **accept** Wake(Marketing) is evaluated and found to be true; that is, there is no higher-priority accountant group waiting to wake the CEO. But before the CEO task can execute the accept statement, it is preempted and an accountant task is enqueued on Wake(Finance_Group). To ensure that the precedence specification is fulfilled, the rendezvous should be made with the accountant task, but this cannot be guaranteed with arbitrary selection of alternatives. Priority_Queuing ensures that the accountant task is accepted before the salesperson task, because it is waiting on an alternative that is textually before the other.

Note that the queuing policy applies only to calls that have begun the protected action and are enqueued on an entry queue. If several tasks are trying to *start* a protected action, they are not queued and nothing can be said about the order in which they will begin the protected action §9.5.1(19).

It is not a good idea to specify Priority_Queuing if you don't need it, because maintaining the entry queues in order of priority will be less efficient than simply adding an node to the tail of a FIFO queue.

Dynamic priorities

Package Ada.Dynamic_Priorities §D.5 contains subprograms Set_Priority and Get_Priority that set and return the *base* priority of a task. The priority change is deferred during a protected action §D.5(10). A change of priority will send the task to the tail of the ready queue for the new priority §D.2.2(9–10) (Figure 16.2).

Dynamic priority modification can be inefficient because entry queues must be updated. Furthermore, there are complications in the interaction between dynamic priority modification and protected objects. Be sure to study the details in the *ARM* and the *Rationale* before using this feature.

Priority ceiling locking

Our discussion of priorities has centered on tasks and rendezvous. Priority ceiling locking describes how priorities are used with protected objects. Ceiling_Locking can be specified with pragma Locking_Policy. In the absence of the pragma, the locking policy is implementation defined §D.3(2–6).

§D.3

8 Every protected object has a *ceiling priority*, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

12 While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object.

13 When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails.

With ceiling locking, protected objects can be implemented on a single processor with no additional locking! Let us assume that task T_1 is executing a protected action with ceiling priority P_C. We show that another task T_2 cannot start a protected action of the same object.

- T_1 is executing a protected action at the inherited ceiling priority P_C §D.3(12).
- T_1 cannot block §9.5(8).
- T_2 cannot preempt T_1 to start the protected action: its priority must be less or equal to P_C §D.3(13), but this is the priority of T_1, and a running task can only be preempted by a task of higher priority §D.2.1(8).
- However, T_1 could be preempted by a third task of priority higher than P_C, which could change the priority of T_2 to an arbitrary priority P_2.

- If $P_2 > P_C$, T_2 is not allowed to call the protected action §D.3(13).
- If $P_2 < P_C$, the scheduler will choose to run T_1, which is of higher priority.
- If $P_2 = P_C$, T_2 will be queued at the tail of the ready queue for this priority §D.2.2(9), but T_1 is at the head of the queue §D.2.2(13), so it will be chosen in preference to T_2.

> 5 If the FIFO_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition. **§D.2.2**

16.3 Monotonic Time

The package Ada.Real_Time §D.8(2–17) specifies a high-resolution, monotonic clock. As with Ada.Calendar, there is a private type Time and a function Clock that returns the current time. Intervals of time are given by the private type Time_Span, rather than a predefined fixed point type like Duration. Time and Time_Span are given in terms of the (same) fixed real number called Time_Unit and Time_Span_Unit, respectively §D.8(19,23). This value must be no more than 20 microseconds §D.8(30), as compared with the required 20 milliseconds and recommended 100 microseconds precision for Duration.

The type Time has no connection with astronomical or geographical time; instead, it is measured from an arbitrary point called the *epoch* (Figure 16.4).

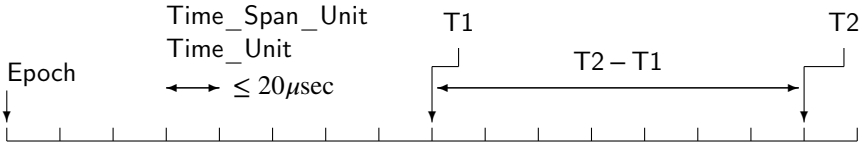

Figure 16.4: Monotonic time

The epoch will usually be the time at which the system is switched on. Clock will return an integral number of Time_Units from the epoch until 'just before now' §D.8(19), and intervals are measured in terms of the integral number of Time_Span_Units between two Times §D.8(20–23). In the figure, at real times T1 and T2, the clock returns an integral number of units less than or equal to the the time, so the time span T2-T1 is also integral. Subprograms are provided to do the usual arithmetical and relational operations on the private types.

Time is required to support a 50 year range §D.8(30), which can be implemented in 64 bits. Time_Span has a much shorter range of plus/minus one hour §D.8(31), as compared with one day for Duration. These requirements are relaxed for implementations with a word size of less than 32 bits §D.8(46).

You can convert a value of type Time_Span to and from a value of type Duration §D.8(25), but there is no direct way to convert between Ada.Calendar.Time and Ada.Real_Time.Time. Instead, procedure Split §D.8(29) converts a value T of type Ada.Real_Time.Time measured in implementation-dependent Time_Units into SC, an integral number of seconds since the epoch

of type Seconds_Count, and TS, a remainder of type Time_Span (Figure 16.5). If you call Ada.Calendar.Clock once at the beginning of the program (or otherwise enter an ordinary time), you can use the seconds count to compute an ordinary time based on monotonic time. There is also a function Time_Of that does the reverse conversion.

Figure 16.5: Splitting monotonic time

Values of type Ada.Real_Time.Time can be used in a delay_until_statement §D.8(18). Values of type Time_Span can be converted into values of type Duration for use in a delay_relative_-statement, delay alternatives or timed entry calls. §D.9 specifies certain performance requirements on delay statements that are intended to make real-time programs more predictable.

16.4 More on real-time systems**

There are five additional clauses in Annex §D. We briefly survey these topics and refer you to the *Rationale* for detailed justifications and examples.

Preemptive abort

Recall the concept of an abort completion point:

> 15 ... the execution of the aborted construct completes no later than its next *abort completion point* (if any) that occurs outside of an abort-deferred operation; ... **§9.8**

In a real-time system, it can be important that a task be aborted as soon as possible in order to release the resources it holds. Furthermore, a task in an infinite loop may never reach an abort completion point. §D.6(2) requires that an aborted constructed be completed *immediately*, provided that it is not within an abort-deferred operation. See §D.6(3–8) for documentation and metric requirements, especially as they relate to multiprocessors.

Tasking restrictions

There are no predefined limits to tasking in Ada: you can allocate an indefinite number of tasks at run-time, any number of which can be blocked on a queue. Furthermore, the program text is not limited in terms of the number of select alternatives, nor in the number of task and protected entries. An implementation may need to use dynamic data structures to support this flexibility. In addition, certain features like abort can impose significant overhead on the algorithms that implement tasking.

> 1 This clause defines restrictions that can be used with a pragma Restrictions (see 13.12) to facilitate the construction of highly efficient tasking run-time systems. **§D.7**

§D.7(2–19) list restrictions that can be placed on the use of tasking in a program. If you can specify restrictions on the number of tasks or entries, the implementation may be able to use efficient static arrays rather than dynamic lists for the internal data structures.

20	It is implementation defined whether the use of pragma Restrictions results in a reduction in executable program size, storage requirements, or execution time. If possible, the implementation should provide quantitative descriptions of such effects for each restriction.	**§D.7**

Even if your implementation does not create a more efficient run-time system when restrictions are requested, you may want to use the pragma to flag uses of features that you have decided to refrain from using in your design.

Synchronous task control

A task that wishes to suspend itself can simply call a protected entry with a barrier that evaluates to false, and wait until another task changes the value of the variables in the barrier so that it evaluates to true. §D.10 defines a lower-level primitive that can be used for this purpose when protected objects are not appropriate. Package Ada.Synchronous_Task_Control defines a limited private type Suspension_Object and subprograms on the type. They can be used to implement *two-stage suspension*: a task indicates that it is about to suspend on an object of the type and then it suspends itself. Another task will eventually release the suspension. The construct is equivalent to a binary semaphore with a queue of size one for blocked tasks. As such it is a very low-level, but very efficient, construct.

Asynchronous task control

Just as a task may want to suspend itself, it may need to suspend another task without its cooperation through an entry call. Package Ada.Asynchronous_Task_Control §D.11 declares the subprogram Hold for changing the priority of an arbitrary task to the *held priority*. This priority is defined to be lower than the priority of a conceptual *idle task*, which in turn is below the value of System.Priority'First. Such a task will never be scheduled until its priority is explicitly reset by calling Continue. The rules for held tasks follow naturally from this model as detailed in §D.11(14–19).

Other optimizations and determinism rules

> 2 If the implementation blocks interrupts (see C.3) not as a result of direct user action (e.g. an execution of a protected action) there shall be an upper bound on the duration of this blocking. **§D.12**
> 3 The implementation shall recognize entry-less protected types. The overhead of acquiring the execution resource of an object of such a type (see 9.5.1) shall be minimized. ...
> 4 Unchecked_Deallocation shall be supported for terminated tasks that are designated by access types, and shall have the effect of releasing all the storage associated with the task. This includes any run-time system or heap storage that has been implicitly allocated for the task by the implementation.

This first requirement is important for predicting interrupt response time ('latency'). The second encourages the use of simple protected objects for efficient mutual exclusion. The third enables you to reclaim storage for tasks allocated through an access type declared at library level; otherwise, storage may not be reclaimed until the environment task is left, that is, when the program terminates.

16.5 Annex E Distributed Systems

> 2 A *distributed system* is an interconnection of one or more *processing nodes* (a system resource that has both computational and storage capabilities), and zero or more *storage nodes* (a system resource that has only storage capabilities, with the storage addressable by one or more processing nodes). **§E**
> 3 A *distributed program* comprises one or more partitions that execute independently (except when they communicate) in a distributed system.
> 4 The process of mapping the partitions of a program to the nodes in a distributed system is called *configuring the partitions of the program*.
> 5 The implementation shall provide means for explicitly assigning library units to a partition and for the configuring and execution of a program consisting of multiple partitions on a distributed system; the means are implementation defined.

Note the terminology: what is normally called a 'program' is called a *partition* §10.2 in Ada, while a program is a set of partitions that can be assigned to nodes. One or more or all (active) partitions may be assigned to a (processing) node. §E.3 requires version consistency among the units of a distributed system.

Categorization

The central problem of programming a distributed system is to establish the semantic connections between units assigned to different active partitions §E.1(2). If a type T is declared in a package P,

and the package is used in more than one active partition, is one type defined for all partitions, or does each partition define its own type which may not be consistent with the others? The solution in Ada is to *categorize* library units §E.2(1–2). A categorization pragma restricts the entities that can be declared within a unit; more restrictive categories can be used to maintain consistency across partitions.

There are five categories:

- Pure §10.2.1(16–17)—A pure unit has no state, and can be consistently replicated in more than one partition.
- Shared Passive §E.2.1—A shared passive unit has only passive data (variables) and subprograms, but not tasks or protected objects with entries. A *passive partition* can contain only pure and shared passive units, and can be assigned to a storage node.
- Remote Types §E.2.2—Remote types units are used to contain declarations of access-to-subprogram or access-to-class-wide-type that are used as encodings for communications between active partitions. They can also be consistently replicated in more than one partition. Variables cannot be declared in the visible part of the unit.
- Remote Call Interface §E.2.3—RCI units are split across partitions: all partitions share the package *specification*, but the package *body* is assigned to one partition. If a visible subprogram of an RCI unit is called from a partition not containing the body, the call is transparently forwarded to the partition containing the body. Variables cannot be declared in the visible part of the unit.
- Normal—No restrictions. A type declaration in a normal unit gives rise to distinct types in each partition containing the unit.

Pragmas §E.2(3) are used to specify all categories (except for normal units, of course). A unit can only depend on units of categories that appear above it in the hierarchy.

Remote subprogram calls

> 1 A *remote subprogram call* is a subprogram call that invokes the execution of a subprogram in another partition. The partition that originates the remote subprogram call is the *calling partition*, and the partition that executes the corresponding subprogram body is the *called partition*. Some remote procedure calls are allowed to return prior to the completion of subprogram execution. These are called *asynchronous remote procedure calls*. **§E.4**

The mechanics of a remote subprogram call—also known as a *remote procedure call (RPC)*—are described in §E.4(9–20) and illustrated in Figure 16.6. The calling partition (on the left side of the figure) will contain the specification of the RCI package P that contains a procedure Proc. Replacing the body of the package is a *calling stub*. The stub is the interface between the calling partition and the underlying *partition communication subsystem (PCS)* §E.5. The calling stub *marshals* the parameters of the procedure call, that is, it translates them into stream elements using the attribute Write, as was described in Section 11.3. The stream is passed to the PCS by calling

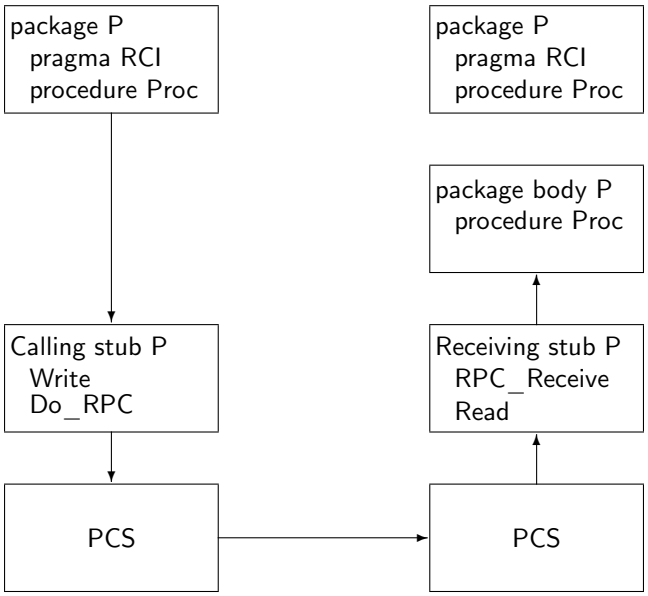

Figure 16.6: Remote procedure call

Do_RPC (or Do_APC for a non-blocking asynchronous call) declared in package System.RPC §E.5(17–20).

When the call is received by the PCS of the called partition, the *receiving stub* is notified by calling an *RPC_Receiver* procedure §E.5(21) and the parameters are *unmarshalled* from the stream. Finally, the receiving stub calls the subprogram body in the package body. If the RPC has **out** or **in out** parameters, they are returned to the calling partition in a similar manner.

This processing is transparent to the programmer; you only have to declare the units with pragmas such as Remote_Call_Interface (respecting the restrictions, of course), and configure the units into partitions using an implementation-supplied tool. The stubs, streams and PCS are the responsibility of the implementation. Only the PCS is non-portable, since it must implement the transmission of a stream over a physical communications channel.

Types of RPC

§E.4

2 There are three different ways of performing a remote subprogram call:
3 As a direct call on a (remote) subprogram explicitly declared in a remote call interface;
4 As an indirect call through a value of a remote access-to-subprogram type;
5 As a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type.
6 The first way of calling corresponds to a *static* binding between the calling and the called partition. The latter two ways correspond to a *dynamic* binding between the calling and the called partition.

We will give a short example demonstrating the third technique; more examples can be found in the *Rationale* and in Burns & Wellings (1995).

Case study: distributed simulation

Suppose that our rocket simulation can no longer be run on a single computer; we redesign it for a system with a separate node (computer) for simulating each event type (here limited to telemetry and engine events) and an additional node to create the scenario (Figure 16.7).[3] A remote types package Root_Event is shared by all partitions. Package Simulation_Server is remote call interface package assigned another partition and is responsible for dispatching calls to the Simulate subprogram.

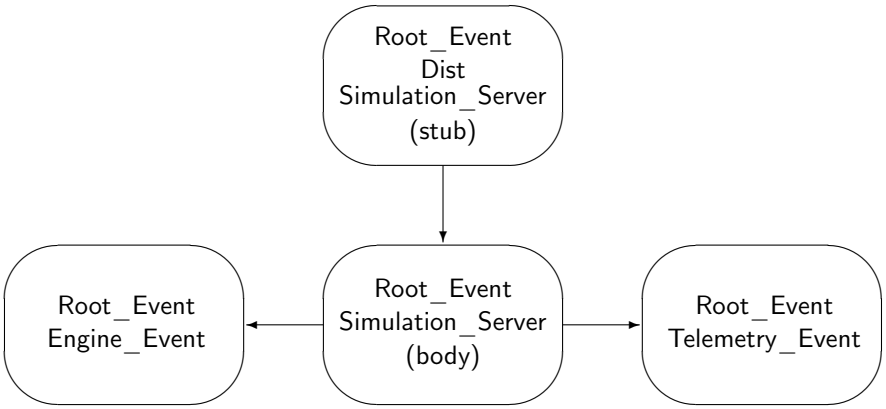

Figure 16.7: Distributed simulation

The normal declarations of a tagged type and primitive subprograms are declared in the remote types package Root_Event ‡6–8. Pragma Remote_Call_Interface ‡15 has been specified in the Simulation_Server to enable it to be split between the partition containing the client Dist and its own partition. Function Get_Event ‡89–100, which returns a pointer to the class-wide type, stands for the Get function of the heterogeneous priority queue. It prompts you to enter a character, which is used to decide which specific type to return. This is sent to the Simulation_Server which derefences the point and dispatches the call to one of the derived classes on another partition ‡22.

[3]The program in this section is significantly different from the one in the printed book; the errors in that program were not caught by an early version of the compiler. Thanks to Jack Flynn for providing this program.

```
1   --                                                    -- File: DIST
2   -- Distributed dispatching.
3   --
4  package Root_Event is
5    pragma Remote_Types;
6    type Event is abstract tagged limited private;
7    type Event_Ptr is access all Event'Class;
8    procedure Simulate(E: in Event) is abstract;
9  private
10   type Event is abstract tagged limited null record;
11 end Root_Event;
12
13 with Root_Event;
14 package Simulation_Server is
15   pragma Remote_Call_Interface;
16   procedure Go_Simulate(E_Ptr: Root_Event.Event_Ptr);
17 end Simulation_Server;
18
19 package body Simulation_Server is
20   procedure Go_Simulate(E_Ptr: Root_Event.Event_Ptr) is
21   begin
22     Root_Event.Simulate(E_Ptr.all);
23   end Go_Simulate;
24 end Simulation_Server;
25
26 package Root_Event.Engine is
27   type Engine_Event is new Event with private;
28   function Create(F, O: Natural) return Event_Ptr;
29   procedure Simulate(E: in Engine_Event);
30 private
31   type Engine_Ptr is access all Engine_Event;
32   type Engine_Event is new Event with
33     record
34       Fuel, Oxygen: Natural;
35     end record;
36 end Root_Event.Engine;
37
```

```
38 with Ada.Text_IO; use Ada.Text_IO;
39 package body Root_Event.Engine is
40    function Create(F, O: Natural) return Event_Ptr is
41       E: Engine_Ptr := new Engine_Event;
42    begin
43       E.Fuel := F; E.Oxygen := O;
44       return Event_Ptr(E);
45    end Create;
46    procedure Simulate(E: in Engine_Event) is
47    begin
48       Put_Line("Engine fuel " & Integer'Image(E.Fuel) &
49                   " L, oxygen " & Integer'Image(E.Oxygen) & " L");
50    end Simulate;
51 end Root_Event.Engine;
52
53 package Root_Event.Telemetry is
54    type Telemetry_Event is new Event with private;
55    type Subsystems is (Engines, Guidance, Communications);
56    type States is (OK, Failed);
57    function Create(Sub: Subsystems; St: States) return Event_Ptr;
58    procedure Simulate(E: in Telemetry_Event);
59 private
60    type Telemetry_Ptr is access all Telemetry_Event;
61    type Telemetry_Event is new Event with
62       record
63          ID: Subsystems;
64          Status: States;
65       end record;
66 end Root_Event.Telemetry;
67
68 with Ada.Text_IO; use Ada.Text_IO;
69 package body Root_Event.Telemetry is
70    function Create(Sub: Subsystems; St: States) return Event_Ptr is
71       E: Telemetry_Ptr := new Telemetry_Event;
72    begin
73       E.ID := Sub; E.Status := St;
74       return Event_Ptr(E);
75    end Create;
76    procedure Simulate(E: in Telemetry_Event) is
77    begin
78       Put_Line("Telemetry message " &
79                   Subsystems'Image(E.ID) & " " &
80                   States'Image(E.Status));
81    end Simulate;
82 end Root_Event.Telemetry;
```

```
83
84 with Root_Event;
85 with Simulation_Server;
86 with Root_Event.Telemetry; with Root_Event.Engine;
87 with Ada.Text_IO; use Ada.Text_IO;
88 procedure Dist is
89    function Get_Event return Root_Event.Event_Ptr is
90       C: Character;
91    begin
92       Put(" Choose system "); Get(C);
93       if C = 'e' then
94          return Root_Event.Engine.Create(500, 600);
95       else
96          return Root_Event.Telemetry.Create(
97                      Root_Event.Telemetry.Engines,
98                      Root_Event.Telemetry.Failed);
99       end if;
100   end Get_Event;
101 begin
102   Simulation_Server.Go_Simulate(Get_Event);
103 end Dist;
```

16.6 Annex H Safety and Security

The popularity of massive, widely distributed, but often bug-infested, software packages for personal computers has caused some people to forget that much software is developed for critical computer systems in closed projects. Examples are control programs for aircraft or power plants. As we discussed in the opening chapter, Ada is specifically designed for these systems, with its support for compile-time and run-time type checking, encapsulation and abstraction, and reusability. The software for these systems can be so critical that it is worth investing a large amount of effort to validate it. Mathematical verification of a program relative to its specification is an important tool in software validation, but even when this is feasible, there are implementation questions that must be addressed, such as: Is the object code produced by the compiler equivalent to the source code? Annex §H addresses additional support in Ada for these systems.

Support for program validation

In addition to the implementation-defined characteristics summarized in §M that must be documented, support of Annex §H imposes additional documentation requirements.

> 1 The implementation shall document the range of effects for each situation that the language rules identify as either a bounded error or as having an unspecified effect. If the implementation can constrain the effects of erroneous execution for a given construct, then it shall document such constraints. . . . **§H.2**

This simplifies validation by reducing the analysis that must be done for each potential error.

One way to address the correspondence of the source and object code is to prove the correctness of the compiler. Given the size of Ada compilers, this may not feasible, so the only alternative is to directly validate the object code. Pragma Reviewable §H.3.1 requires that an implementation produce information such as an object code listing, memory requirements and potential error situations, preferably in both human-readable and machine-readable form so that it can be processed by automated tools §H.3.1(19).

A valuable tool for testing and debugging programs is a hardware analyzer that—with little or no effect on the behavior of the computer—can save some or all of the state of a computation for later analysis. Pragma Inspection_Point §H.3.2 can be used within a sequence of declarations or statements to require the implementation to make some or all objects available for inspection. The pragma is not just a documentation requirement telling you where to find each object; it can modify code generation by requiring that a value be stored in memory rather than retained in a register when an inspection point is reached.

Abnormal objects

When you create and initialize an object, it is in a *normal* state §13.9.1(4). An object can become *abnormal* in certain unusual circumstances; for example, if an assignment is interrupted by an abort statement, a discriminant might be updated but not a variant that depends on the discriminant. It is erroneous to use an abnormal object §13.9.1(8), so it is important to check your program to prevent such situations from occurring.

Invalid representations

A *scalar* object can have an *invalid representation* §13.9.1(9).

```
subtype Index is Integer range 0..255;
N1: Index := 300;
N2: Index;
```

Since Index is of type Integer, it will be allocated a full word of memory (16 or more bits), but only a few of the possible contents of the word represent valid values of its subtype. The initialization of N1 will raise Constraint_Error, because the value is not within the range of the subtype. However, N2 is uninitialized, so even if the bit pattern in memory represents an invalid value no exception would be raised. Other sources of invalid data are Unchecked_Conversion and importing an object or subprogram; for example, a byte received from a communications line might be intended to represent a value of an enumeration type with only a dozen values.

You can explicitly check the validity of a scalar object by using the attribute Valid §13.9.2. The attribute never raises an exception.

Many languages and compilers automatically zero uninitialized variables. This actually reduces reliability, because zero is often a valid, though unintended, value of the subtype! By using pragma Normalize_Scalars §H.1, you request the compiler to set uninitialized objects to a predictable, but if possible invalid, representation. For example, the variable N2 above might be initialized to Integer'First. The idea is to 'flush out' the error as soon as possible by increasing the likelihood

that Constraint_Error will be raised, rather than continuing the computation with an incorrect value.

Restrictions

Validation of a program requires validation not just of your code, but also of the code of the run-time system and standard libraries. By restricting the language features in a program, an implementation can support a smaller run-time system and set of libraries, greatly reducing the validation effort. Furthermore, a program that is guaranteed not to use tasking or dynamic allocation need not be checked for problems such as deadlock or storage leaks.

Pragma Restrictions §13.12 specifies the restrictions that a program can take upon itself. An implementation that supports Annex §H is required to support certain restrictions such as No_Allocators §H.4. In addition, the restrictions described in §D.7 must be supported, in particular, restricting Max_Tasks to zero §H.4(2). This allows an implementation to remove support for tasking from the run-time system, making its validation significantly easier. You must check your implementation's documentation to see what effect restrictions have on a program §13.12(9).

A Tips for Transition

When you learn a new programming language, there is a tendency to apply inappropriate analogies from previous experience. Furthermore, there are often subtle differences between languages that may not be apparent at first glance. This appendix points out some of problems that programmers experienced in Pascal, C, C++ or Java may encounter when learning Ada.

A.1 Pascal

Since the development of Ada started from Pascal, you will not have much trouble with the general syntax and semantics of elementary constructs in Ada, though there are many differences in the details. The discussion relates to standard Pascal; extended versions of the language have many features in common with Ada.

- There is no **program** declaration. The 'main' program is simply a library unit which is a subprogram. Like all subprograms, it is terminated with a semicolon, not a period.
- Ada does not restrict the order of declarations as in Pascal. Constants, types, variables and subprograms can be declared in any order.
- The Pascal paradigm of declaring array types by first declaring a constant, then an index type and finally an array is not needed in Ada because you can declare a constrained array subtype and then use attributes. See Section 3.2.
- Semicolons *separate* statements in Pascal and *terminate* statements in Ada. Many Pascal programmers 'terminate' statements with semicolons; in reality, the semicolon separates the final statement from a null statement. Of course this does not work before else statements, so you may have learned a 'special' rule for this case. In Ada, the terminating semicolon is always needed:

```
if A > B then
   S1;                              -- Don't forget the semicolon!
else
   S2;
end if;                             -- Don't forget the semicolon!
```

- Pascal uses **begin**—**end** to bracket compound statements. Ada uses reserved word pairs like **loop**—**end loop** instead. However, Ada also uses **begin**—**end** to create a block that is used to set up run-time structures like local variables and exception handlers. The block in the following if-statement is legal, but neither needed nor recommended:

```
if A > B then
  begin                                  -- Legal, but not needed in Ada!
    S1;
    S2;
  end;                                   -- Legal, but not needed in Ada!
end if;
```

- With the exception of anonymous arrays, all types in Ada *must* be explicitly declared. There is no analogy to the Pascal declaration:

```
State: (Off, Standby, On);
```

- An Ada subprogram *body* has the reserved word **is** instead of a semicolon between the declaration and the block. Unfortunately, a subprogram *declaration*—used for example in package specifications—is terminated by a semicolon. If you replace the **is** by a semicolon, you may get strange error messages:

```
procedure P(X: in Integer);
                                         -- OK! Subprogram declaration
begin                                    -- Strange error message here
  ...
end P;
```

- Feel free to use the exit and return statements in Ada. There is no reason to continue using the contorted style needed to overcome the limitations of Pascal's control structures.

- The loop parameter of an Ada for-loop is *implicitly* declared and its scope is restricted to the loop body:

```
K, N: Integer;                           -- Explicitly declared variables
for N in 1..10 loop                      -- Implicitly declared loop parameter
  ...                                    -- Explicitly declared N is hidden here
  if Found then K := N;                  -- Save loop parameter
  ...
end loop;
Put(N);                                  -- Prints the explicitly declared N
Put(K);                                  -- Prints saved loop parameter K
```

- A value parameter in Pascal is an initialized local variable. In Ada, an **in** parameter is a *constant* which cannot be assigned to; use additional local variables if needed. Pascal **var** parameters are implemented as reference parameters, whereas scalar parameters in Ada are passed by copy. Arrays and records in Ada can be passed by copy or by reference.

- An Ada function can return any (nonlimited) type, including arrays and records, but parameters of functions are restricted to **in** mode.

- Pascal uses **with** to open the name space of a record. This is similar to **use** in Ada, which opens the name space of a package, and not at all related to **with** in Ada, which is used to import packages.

- Ada has no construct analogous to the Pascal **with**, but you can use **renames** for similar purposes: to shorten names and to help the optimizer avoid recalculation of record addresses (Section 12.5).

A.2 C

C was designed to make it easy to manipulate the underlying machine, whereas Ada was designed to make it easy to construct large, reliable software systems. It may take you a while to learn how to use Ada's type system, and the compile-time type checking may prove frustrating at first.

- In Ada, "=" is the equality *operator* while ":=" is the token used in the assignment *statement*. Assignment cannot be used in an expression and there is no multiple assignment.
- Enumeration types are true types, not just disguised integers as in C. You can convert an enumeration value to its position and back using the attributes Pos and Val, but normally the values and attributes of the enumeration type should be used directly:

```
type Waves is (Radio, Microwave, Infrared, Visible, UV, X_Ray);

for W in Waves loop ...
for W in Waves'First .. Waves'Pred(Waves'Last) loop ...
  -- Good Ada style

for I in Waves'Pos(Radio) .. Waves'Pos(X_Ray) loop ...
for I in Waves'Pos(Radio) .. Waves'Pos(X_Ray)-1 loop ...
  -- Legal, but this is programming in C style
```

- Predefined Integer should rarely be used in Ada. Subtypes and user-defined integer types allow you to precisely express both the range of a variable and its relationship to other objects.
- The bounds of a for-loop in Ada are evaluated *once* before beginning the loop. Only counting loops which increment or decrement by one are allowed. Other C loop paradigms will have to be explicitly programmed in Ada using **while** and **exit**. Furthermore, the loop parameter in Ada is implicitly declared and has its scope limited to the loop body.
- In case statements there is no 'fall-through' from one alternative to the next, so there is no need for a break statement. There is no continue statement in Ada.
- Arrays in Ada are first-class types and can be assigned and passed as parameters. The index type of an array can be any discrete subtype, not just a range of the natural numbers starting from zero. Array parameters 'carry' their bounds with them, and these can be accessed as attributes, so there is no need for additional parameters. There is no equivalence between array indices and pointers as in C.
- Normally, pointers in Ada (access types) only point to elements that are dynamically allocated. Taking the address of a statically allocated object (& in C) is not needed in Ada for parameter passing, since *any* type can be passed as a parameter and returned from a function. The Ada attribute Access is used mostly for constructing static data structures. It is subject to restrictions that make it impossible to create a dangling pointer.
- There is no **void** type or pointer in Ada, and pointers to different types cannot be converted to each other except by escaping the type system with Unchecked_Conversion.
- In C, a source code file also delimits a scope of definition; 'h'-files are used to repeat declarations in several files. Source files have no meaning in Ada; scope is determined by *units*:

packages, subprograms, etc. Similarly, lifetime is determined by these units: a variable in a library package is statically allocated like a variable declared within a C source file, and a variable declared in a subprogram is automatically allocated upon invocation. There is no analogue in Ada to a static variable declared within a C *function*.

A.3 C++

Most of what was written about C is relevant to C++ and need not be repeated. Beyond the C constructs, the most significant difference between Ada and C++ is that Ada has extensive support for encapsulation with its hierarchial packages, while C++ retains the independent-file model of C. The constructs of C++ for object-oriented programming have close conceptual analogues in Ada, though the syntax and semantics are quite different.

- C++ uses the same concepts of source code files, file scope and 'h'-file conventions as C does. While classes can define abstract data types, they cannot encapsulate them as Ada packages do. In particular, definitions of function members can be placed anywhere (not just in a package body), and depend on declarations that 'just happen' to be in the same 'h'-file.
- C++ namespaces give you control over visibility similar to that of Ada packages.
- An Ada compiler is required to have an *implicit* 'Make' facility to ensure consistency among the units.
- Static members are like variables declared within an Ada package. A static method is like a subprogram with no parameters of the abstract data type defined by the package.
- C++ uses a distinguished-receiver syntax to call a member for an object: obj.func(x) executes func(x) on obj, which becomes an implicit parameter accessible as **this**. The Ada syntax for record components, task and protected entries is similar, but for calling a primitive operation on an object Ada uses ordinary parameters. The object itself must be given explicitly as an additional parameter: func(obj, x).

  An advantage of the Ada syntax it that it is easy to dispatch on binary operators:

```
function "<"(Left, Right: Parent) return Boolean;
function "<"(Left, Right: Derived) return Boolean;
procedure P(X, Y: Parent'Class) is
begin
  if X < Y then ...
  -- Dispatches on specific type of X and Y
end P;
```

- There are no constructors and destructors in Ada. The functionality of simple constructors can be obtained by using default initial values for the components of a record type, record aggregates and initializing functions. User-defined initialization, assignment and finalization is supported by controlled types.
- For dynamic dispatching, a function member in C++ must be declared **virtual**; otherwise, static binding is used for all calls. In Ada, any primitive subprogram is potentially dispatching and the type of dispatching is determined *per call*.

- In C++ dynamic dispatching will be done on pointers and references; the type of a *value* is known at compile-time and the dispatching can be optimized away. No explicit pointer or reference is needed in Ada for dispatching; class-wide types are indefinite types are implicitly allocated at subprogram or block entry with an arbitrary object in the class.
- There is no term in C++ for what Ada calls a class; the equivalent concept is the hierarchial family of derived types. There are no class-wide types in C++; instead, rules are stated in terms of type conversion within the family.
- There is no multiple inheritance in Ada. Generics and access discriminants are used where C++ would use multiple inheritance (see Section 7.5). Section 4.6 of the *Rationale* discusses these techniques at length.
- Explicit try-blocks are not needed in Ada for exception handling, since every subprogram (more exactly, every construct containing the syntactic category handled-sequence-of-statements) can have an exception handler. Ada exceptions are simple identifiers, whereas exception handling in C++ is controlled by matching on the catch profile. The matching understands type derivation in the sense that a handler for an object of a base type will handle an object of a derived type.
- Generic units must be explicitly instantiated in Ada, unlike the implicit instantiation of templates in C++. Each instantiation of an Ada generic gives a different instance, while C++ implementations will collect all instances with the same actual parameters. In Ada, you would normally create a single instantiation of a generic package, and the instance would be 'with'ed by other units as we do with Root_Event.Random_Time (Chapter 6).
- Ada generics use a contract model which ensures that the instantiation of a generic cannot cause an error in the generic unit body. In C++, compilation of an instantiation can cause a compile-time error within the template itself.
- The Ada library does not contain implementations of data structures that exist in the Standard Template Library.

A.4 Java

Java's superficial syntax is very much like the syntax of C++, but in many ways the language is more similar to Ada: all code must be encapsulated, arrays are first-class objects and type-checking is always done. Nonetheless, the languages are quite different.

Most of the interest in Java is due to support by network browsers of the portable *Java Virtual Machine (JVM)* and associated standard libraries. With an Ada 95 to J-code compiler (Taft 1996), Ada is an attractive alternative to Java for programs in a network environment!

- Java uses reference semantics for all non-primitive types, including arrays and strings, whereas Ada uses value semantics. A common Ada paradigm is to hide its value semantics within a package body, exporting only limited private types and class-wide types.
- The index type of an array in Ada can be any discrete subtype, not just a range of natural numbers starting from zero.

- A Java class is like an Ada package that declares a single visible tagged type.
- Java packages are *not* like Ada packages! An Ada package creates an encapsulation which you have to explicitly 'with' to access. A Java package is really a construct for name space control, much like namespaces in C++. A public construct in any Java package is always accessible to any other class; **import** just makes its name directly available like **use** in Ada or **using** in C++.
- All functions in Java can dispatch. Because of reference semantics, there is no need for class-wide types to implement dynamic dispatching. Any object is—that is, can point to—any object derived from its class; calling a method on the object will cause dispatching.
- Java uses a distinguished receiver syntax; see the note for C++.
- There are no constructors in Ada; see the note for C++. Few Ada compilers implement garbage collection.
- There are no interface declarations in Ada.
- Ada does not require that all code be written within classes and there is no universal base type such as Object.
- Explicit try-blocks are not needed in Ada for exception handling, since every subprogram (more exactly, every construct containing the syntactic category handled-sequence-of-statements) is implicitly the scope of an exception handler. Java exceptions are declared by inheriting from class Throwable, and a method must declare the exceptions that it can throw. Ada exceptions are ordinary identifiers, and a subprogram can propagate any exception raised by itself or by a subprogram it calls.
- The Ada library does not contain implementations of data structures or graphical user interfaces. Annex §E defines constructs for distributed programming.
- Java threads are like Ada tasks. However, the synchronization primitives in Java are very elementary when compared with protected objects and task rendezvous in Ada, and it is difficult to write starvation-free algorithms in Java. The main differences between protected objects and Java 'monitors' are (Ben-Ari 1998*a*):
  - In Java, methods of a class can, but need not be, synchronized. All actions of an Ada protected object (except functions) are executed under mutual exclusion.
  - notify in Java releases an *arbitrary* thread, whereas entries in Ada maintain a FIFO queue.
  - The semantics of barrier re-evaluation in Ada ensure immediate resumption of blocked processes, unlike Java.

B Glossary of ARM Terms

The *ARM* is written in a precise style, with specialized terms defined and then used consistently throughout the document. This glossary collects the definitions of these terms. For brevity, the glossary does not include language constructs like *procedure*, terms from the annexes and well-known terms like *expression*. Source code examples will help you understand and remember the definitions.

The following declarations will be used in the examples:

```
type Piece is (Pawn, Knight, Bishop, Rook, Queen, King);
type Matrix is array(Integer range <>, Integer range <>) of Piece;
type Game_Board(Size: Positive) is
  record
    B: Matrix(1..Size, 1..Size);
  end record;

type Node;
type Ptr is access Node;
type Node is
  record
    Key: Integer;
    Next: Ptr;
  end record;

package P is
  type Parent is tagged null record;
  procedure Primitive(X: Parent);
  type Derived is new Parent with null record;
  procedure Primitive(X: Derived);
end P;
```

abandoned §11.4(3) See *exception*.

abort deferred §9.8(5) When an abort statement is executed, if a task is executing one of the operations listed in this section, the operation is allowed to complete before the abort takes effect. For example, a protected action is abort-deferred and will be allowed to complete.

abnormal completion §7.6.1(2) The execution of a task completes abnormally if an exception is raised and not handled, or if a task is aborted.

abnormal object §13.9.1(1) If an assignment statement is interrupted by abort or if a non-scalar object is returned from an imported subprogram, the bits representing the object might not form a value of the object's type. The object is abnormal and certain uses are *erroneous*.

A scalar object that does not contain a value of its subtype is said to have an *invalid representation*. For example, an uninitialized variable of subtype Natural has an invalid representation if the initial contents of the variable represent a negative number.

accessibility §3.10.2(3) The nesting levels of *masters* at run-time. Accessibility levels are used to prevent dangling pointers when general access types are used. It is illegal to apply the Access attribute to an object that has a *deeper* accessibility level *than* the access type. Usually, access levels are known at compile-time, in which case one level can be determined to be *statically deeper than* another §3.10.2(4). A library unit is at *library level* §3.10.2(22).

```
type Ptr is access all Integer;
function F return Ptr is
  N: Integer;                          -- Statically deeper than Ptr
begin
  return N'Access;                     -- Error !
end F;
```

actual subtype §3.3(23) See *nominal subtype*.

adjust §7.6(15–16) The operation performed on the target object of a *controlled type* after the new value is copied during assignment. Procedure Adjust can be overridden.

ambiguous §8.6(30) See *overloading*.

ancestor §3.4.1(10) See *derived type*.

ancestor §10.1.1(11) See *unit*.

anonymous type §3.2.1(7) Single arrays, tasks and protected objects are objects of anonymous type. Since the type has no name, you cannot declare other objects of the type. However, you can convert an anonymous array to a named array type §4.6(9–12).

```
Translate_Table: constant array(Character) of Character := ...;
                                                   -- Anonymous array
type Table_Type is array(Character) of Character;
T: Table_Type := Table_Type(Translate_Table);

task Bounded_Buffer is                             -- Anonymous task
  entry Put(I: in Integer);
  entry Get(I: out Integer);
end Bounded_Buffer;
```

base range §3.5(6) Scalar types, in particular numeric types, are usually implemented in a larger range than requested—the base range. Computations involving predefined operators are done in the base range. The attribute S'Base gives the *base subtype* of the type of S.

```
type Int is range 0..10_000;
K1, K2: Int      := 8_000;
K:      Int      := K1 + K2;
B:      Int'Base := K1 + K2;
```

The computation K1 + K2 will not cause an error. Constraint_Error will only be raised when the result is assigned to K, though not when assigned to B.

bounded error §1.1.5(7–8) See *error*.

by copy §6.2(2–3) Parameters of elementary types must be passed by copy, that is by creating a local object and copying the actual parameter in and/or out. The types listed in §6.2(5–9) must be passed *by reference*. These are types such as tagged types and task types that are used to represent entities (such as a task control block); they require that a reference to the entity be passed rather than a copy of the entity's contents. An implementation may choose to pass other types such as records and arrays either by copy or by reference.

by reference §6.2(4) See *by copy*.

callable §6(2) A callable entity is a subprogram or entry. A *callable construct* defines the action taken when the entity is called; it is a subprogram or entry body, or an accept statement.

check §11.5(2) A language-defined check such as Index_Check can *fail*, raising an exception.

class-wide type §3.4.1(4) A type whose values are the union of the values of all the *specific types* within a *derivation class*.

compatible §3.2.2(12) See *constraint*.

compilation §10.1(2) See *unit*.

complete context §8.6(4) See *overloading*.

completely defined §3.11.1(8) See *full type*.

completion §3.11.1 Some declarations may be written in two parts; the first part is said to *require* a completion. For example, a subprogram declared in a package specification requires a body in the package body.

completion §7.6.1(2) The end of the execution of an entity. Completion may occur either by executing the last statement, or by a transfer of control caused by raising an exception or executing a statement such as return statement. After completion, a construct is *left* §7.6.1(3), meaning that its execution continues with the next action. Before leaving a master, *finalization* §7.6.1(4) must be performed: waiting for a dependent task, and for *controlled types* execution of the Finalize procedure. A *master* §7.6.1(2) is the execution of a task, entry or subprogram body, or a block or accept statement.

composite type §3.2(4) Composite types are types that can have components: record types and their extensions, array types, task types and protected types. Private types and their extensions are also considered to be composite types. Composite types (except for arrays) may have discriminants.

conformance §6.3.1 Two subprogram *profiles* conform to each other in one of four ways, each one adding additional requirements:

- *type conformant* The number and types of parameters are the same. Since two subprogram declarations are *homographs* if they are type conformant §8.3(8), they cannot be overloaded:

```
procedure Proc(X: in Integer := 5);
procedure Proc(Y: out Positive);                -- Error, homograph
N: Integer;

Proc(N);                                        -- Ambiguous
```

- *mode conformant* The profiles have identical modes. Mode conformance is sufficient for generic instantiation §12.6(8):

```
generic
  with procedure Formal(X: in Integer);
procedure Gen;

procedure Actual(Z: in Positive);
procedure Instance is new Gen(Formal => Actual);
```

  Formal(-1) is a legal call within the body of Gen even though it will cause a constraint error in the instantiation Instance.

- *subtype conformant* Subtypes of the profile must *statically match*. Subtype conformance is sufficient for overriding §3.9.2(10); the following declaration of Primitive overrides the parent primitive subprogram even though the formal parameter name has changed:

```
procedure Primitive(XXX: Derived);
```

- *fully conformant* The formal parameter names must be identical and the default expression must be fully conformant §6.3.1(19–22). A completion must be fully conformant §6.3(4):

```
package P is
  procedure P1(X: Integer; Y: Integer);
  procedure P2(X: Integer := 5);
  procedure P3(X: Integer := 5);
end P;
package body P is
  procedure P1(Y: Integer; X: Integer) is ...   -- Error
  procedure P2(X: Integer := 10); is ...        -- Error
  procedure P3(X: Integer := 2#101#) is ...     -- OK
end P;
```

  Calls P.P1(X=>3,Y=>7) and P.P2 in a client are interpreted according to the specification (since the body could always be modified and recompiled). If full conformance was not required, the procedure specifications in the body would have to be ignored, which would be confusing for the reader!

constant §3.3(13) See *object*.

constraint §3.2(7) A restriction on the possible values of a type. A value *satisfies* a constraint if it satisfies the restriction. A constraint applied to a subtype must be *compatible* with the subtype. There are three types of constraints:

- *range constraint* §3.5(2)

```
subtype Moves_Far is Piece range Knight..Queen;
subtype Important1 is Piece range Queen..King;
subtype Important2 is Moves_Far range Queen..King;
  -- Error, constraint not compatible with subtype
```

- *index constraint* §3.6.1(2)

```
Rectangle: Matrix(4,8);
```

- *discriminant constraint* §3.7.1(2)

```
Board: Game_Board(8);
```

constrained subtype §3.2(9) See *unconstrained subtype*.

controlled type §7.6(2) A controlled type is a type descended from the abstract tagged types Controlled or Limited_Controlled. These types have primitive procedures Initialize, Adjust and Finalize, which are called when a value of the type is elaborated, assigned or finalized. You can override these procedures if needed.

controlling formal parameter §3.9.2(2) If T is a tagged type and S is a primitive subprogram of T, then formal parameters of S of type T are controlling. The call will be statically bound or dynamically dispatched depending on the corresponding *controlling operands* (actual parameters) of the call. A function may also have a *controlling result*.

If the following declaration is placed after the declaration of tagged type Parent, then P1 and P3 are controlling, but P2 is not:

```
procedure S(P1: in Parent; P2: out Integer; P3: in out Parent);
```

convention §6.3.1(2) The convention defines how a callable entity is invoked. *Ada* is the convention for subprograms and *Intrinsic* is the convention for built-in operations. Protected operations have their own convention. Conventions are also used to specify the interface between Ada types and subprograms, and those of another programming language §B.1(2). Some language rules refer to the convention of a subprogram; for example, you cannot use the Access attribute on a subprogram of Intrinsic convention §3.10.2(32):

```
type Ptr is access function (Left, Right: Integer) return Integer;
P: Ptr := "+" ' Access;                    -- Error
```

convertible §4.6(4) A type—the *operand type*—is convertible to another type—the *target type*—if type conversion can be performed according to the rules in §4.6. The conversion may be either a *value conversion*, which creates a new value of the target type, or a *view conversion*, which denotes the operand object.

cover §3.8.1(9–12) A discrete_choice such as 5..10 or **others** is said to cover a set of values of a subtype. In contexts like aggregates, variant records and case statements, a discrete_choice_list

must cover all the values of the subtype. The term is used in a similar way in exception choices §11.2(6).

cover §3.4.1(9) A class-wide type is said to cover all the types in the class, while a specific type covers only itself. Membership tests such as E **in** Main_Engine'Class are defined in terms of covering §4.5.2(2).

current instance §8.6(17–18) In contexts such as task types, protected types and generic units, the name of a type denotes a value or object of the type rather than the type itself. This object or value is the one currently being executed. In the following protected type, the call PT.E refers to the entry of the current instance of the protected object:

```
protected type PT is
  procedure P;
  entry E;
end PT;

protected body PT is
  procedure P is
  begin
    PT.E;
  end P;
  entry E ...
end PT;
```

declaration §3.1(5) A declaration *defines a view* of an entity and associates a name with the view. If may also *define the entity* itself. *Entity* is a general term used for 'things' like objects, subprograms, exceptions, etc. A declaration may be *explicit* or *implicit*:

```
type Piece is (Pawn, Knight, Bishop, Rook, Queen, King);
  -- Explicitly defines the type Piece and the enumeration literals
  -- Piece'First is an implicitly declared attribute
function EOL return Boolean renames Ada.Text_IO.End_Of_Line;
  -- EOL is a new view of an existing subprogram
```

declarative region §8.1(1) A place where a nested declaration can occur. Most declarations are declarative regions, for example, subprograms and record types. However, a for-loop statement is also a declarative region, because the loop parameter is nested within. Note that the declarative region of a package includes its body and children.

deeper than §3.10.2(3) See *accessibility*.

deferred constant §7.4(2) A constant of a private type can be declared before the full type is declared. The value of the constant must be completed following the full type declaration:

```
package P is
   type ID is private;
   Null_ID: constant ID;
private
   type ID is new String(1..5);
   Null_ID: constant ID := "*****";
end P;
```

defining name §3.1(10) The occurrence of name where it is defined is its defining name. Subsequent occurrences of the name are called *usage names*. Syntactic categories for defining names start with 'defining_'; categories for usage names are direct_name and selector_name.

definite subtype §3.3(23) See *indefinite subtype*.

definition §3.1(7) See *declaration*.

depends on §3.7(20–24) See *per-object constraint*.

dereference §4.1(8) A dereference of an access value returns a view of the designated object or subprogram. A dereference may be either *explicit* or *implicit*:

```
Head: Ptr = new Node;
N:    Node := Head.all;                -- Explicit dereference
I:    Integer := Head.Key;             -- Implicit dereference
```

derivation class §3.4.1(1) See *class-wide type*.

derived type §3.4 A derived type is a new type derived from an existing *parent type*. It is said to a *descendant* of its parent type, which is the derived type's *ancestor*. These terms are extended transitively: if a type is derived from a descendant of a parent type, it is also a a descendant of the parent type.

descendant §3.4.1(10) See *derived type*.

descendant §10.1.1(11) See *unit*.

designate §3.10(1) The object or subprogram 'pointed-to' by an access value. The subtype of an object designate is called the *designated subtype* §3.10(10) and the profile of a subprogram designate is called the *designated profile* §3.10(11).

determined class §12.5(6) A generic formal parameter determines a class of types; a matching generic actual parameter must be in this class. For example, the class determined by **private** is the class of all nonlimited types §12.5.1(17).

```
generic
   type T is private;
procedure Gen;

with Gen; with Ada.Exceptions;
package Instance is new Gen(Ada.Exceptions.Exception_Id);
```

By §11.4.1(2), Exception_Id is a private type and can be the generic actual parameter, even though we do not know how the type is implemented. Exception_Occurrence is limited private §11.4.1(3) and cannot be a generic actual parameter for an instantiation of Gen.

direct name §4.1(3) See *name*.

directly visible §8.3(2) See *scope*.

discrete choice §3.8.1(5) A range of values in a case statement, array aggregate or variant record declaration:

```
type Values is array(Piece) of Integer;
V: Values :=
  ( Pawn => 1,                          -- Discrete choice is expression
    Knight..Bishop => 3,                -- Discrete choice is discrete range
    Rook => 5,                          -- Discrete choice is expression
    others => 10);                      -- Discrete choice is others
```

discrete type §3.2(3) Integer types and enumeration types (which include Character and Boolean). Discrete types can be used in for-loop statements §5.5(4) and as array index types §3.6(5).

dynamically tagged §3.9.2(5) A *controlling operand* of class-wide type is said to be dynamically tagged and causes dispatching at run-time. If the operand is of a specific type, it is said to be *statically tagged* and the subprogram can be statically bound at compile-time. If the operand is a function call that returns a class-wide type (and no operands of the function call are either statically or dynamically tagged), the operand is said to be *tag indeterminate*.

elaboration §3.1(11) See *execution*.

elaboration dependencies §10.2(9) See *semantic dependencies*.

elementary type §3.2(3) The elementary types are the *scalar* types and *access* types. Elementary types are *definite* §3.3(23) and are passed as *by-copy* parameters §6.2(3). Scalar types are the *discrete* types and the *real* types (*floating point* and *fixed point* types). Scalar types are ordered §3.5(1) and can have *range constraints* §3.5(7).

environment §10.1.4(1) The environment of a compilation is the context in which the compilation takes place. It is a conceptual program library that contains compiled units. Execution of a partition takes place within an *environment task* §10.2(8), which elaborates all the library units and the main subprogram.

erroneous execution §1.1.5(9–10) See *error*.

error §1.1.5 A violation of a language rule. Errors are classified in four categories:

- Errors that must be detected prior to run-time:

```
procedure Proc(X: out Integer);
Proc(17);                               -- Must be a variable §6.4.1(5)
```

- Errors that must be detected at run-time; these errors will raise exceptions.

- *bounded error* An error that need not be detected by the implementation, but the range of possible effects is limited. For example, it is a bounded error to access a formal parameter by an alias such as a global variable §6.2(12):

```
S: String := "Hello world";
procedure Proc(T: in String) is
begin
  S(1..5) := "Bye ";
  Put(T);                              -- Is T a reference to S or a copy of S?
end Proc;

Proc(S);
```

  The possible effects are: (a) read the old value, (b) read the new value, (c) raise Program_Error.

- *erroneous execution* An error that need not be detected and whose effect is unspecified. For example, if you close the standard output file and then attempt to write on it, the execution is erroneous §A.10.3(23).

evaluation §3.1(11) See *execution*.

exception §11(1) An exception has a *name* (predefined like Constraint_Error or user-defined like Underflow). Each time that an exception is raised, it creates a new *occurrence* of the exception and the execution of the enclosing construct is *abandoned* §11.4(3). The occurrence may be *handled* or *propagated*. Package Ada.Exceptions §11.4.1 declares the type Exception_Id for exceptions, and Exception_Occurrence for occurrences.

execution §3.1(11) A construct achieves its run-time effect when it is executed. Execution of an expression is called *evaluation* and execution of a declaration is called *elaboration*.

expanded name §4.1.3(4) A selected_component of the form prefix.selector_name denotes a component of an object or value of composite type such as a record, or an operation of some task or protected object. The same syntax is also used for *expanded names*, in which case the prefix denotes a named construct such as a package and the name denotes an entity declared in the package. See §4.1.3(10–13) for a list of constructs that the prefix can denote in an expanded name.

```
R: Ada.Strings.Maps.Character_Range;   -- Expanded name
R.Low := R.High;                       -- Record components
```

expected type §8.6(20) The expected type appears in the Name Resolution Rules for a construct and is used for overload resolution. See §8.6(21–25) for the definition of expected type when class-wide types are concerned. Similarly, a context can have an *expected profile* §8.6(26) for a *callable entity*.

```
function F return Duration;
function F return Float;

delay F;
  -- Delay expects Duration §9.6(5), the first F is chosen
```

expiration time §9.6(1) The time at which a delay will expire.

explicit declaration §3.1(5) See *declaration*.

explicit dereference §4.1(5) See *dereference*.

external call §9.5(4) A task entry or protected operation call that explicitly mentions the target object. An *internal call* §9.5(3) is also allowed, in which case the call is to the current task or protected object. The meanings of *external requeue* and *internal requeue* are similar §9.5(7).

```
protected type PT is
  entry E1;
  entry E2;
end PT;
PO: array(1..5) of PT;
protected body PT is
  entry E1 when ... is
  begin
    if ... then
      requeue PO(5).E2;                -- External requeue, different(?) PO
    else
      requeue PT.E2;                   -- Internal requeue, same PO
    end if;
  end E1;
  entry E2 when ... is ...
end PT;
```

Note that the external requeue is potentially blocking §9.5.1(15).

external effect §1.1.3(8) The effect of an Ada program on its external environment. Optimizations are permitted §11.6(3) provided that they don't change the external effect of the program.

fail §9.2(1) The activation of a task can fail, in which case the task becomes *completed*.

fail §11.5(2) See *check*.

finalization §7.6.1(4) See *completion*.

first subtype §3.2.1(6) Types are not considered to have names; instead, types that are not *anonymous* are called *named* types and have *nameable subtypes*. A type declaration declares both the type and the name of a first subtype. Informally, the name is also the name of the type. In the following declaration, Vector is the name of the first subtype of the named array type:

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

freezing §13.14 An entity is frozen when its representation must be fully determined. §13.14 defines which constructs freeze which entities. For example, a tagged type is frozen by an extension, and you cannot declare primitive operations after freezing the type.

full type §3.2.1(8) A declaration that defines a type. Some type declarations do not define a type; for example, a private type declaration declares a partial view of the corresponding full type declaration. The type is *completely defined* after the full type definition.

full type declaration §3.2.1(3) See *incomplete type declaration*.

full view §7.3(4) See *partial view*.

hidden §8.3(5) See *scope*.

homograph §8.3(8) Two declarations are *homographs* if they have the same name, and, for callable entities (subprograms, entries and enumeration literals), if they have type-conformant parameter profiles. One homograph can hide or override another; otherwise, it is usually illegal for two homographs to be immediately within the same declarative region.

```
package P is
  type Parent is tagged null record;
  procedure Primitive(X: Parent);
  type Derived is new Parent with null record;
  procedure Primitive(X: Derived);          -- Overrides
  procedure Primitive(XXXX: Derived);       -- Error
  package Inner is
    procedure Primitive(X: in Parent);      -- Hides
  end Inner;
end P;
```

immediately within §8.1(13) A declaration is immediately within the innermost enclosing declaration. For example, an accept statement must be immediately within a task body §9.5.2(14):

```
task body T is
  procedure Proc is
  begin
    accept E;                   -- Error
  end Proc;
begin
  accept E;                     -- OK
end T;
```

implementation-defined §1.1.3(18) The semantics of Ada is not fully specified by the reference manual. A rule may specify a set of possible effects that a construct may have, and an implementation may choose any of these possible effects. For example, an implementation may supply predefined integer types in addition to Integer §3.5.4(25). See §M for a list of characteristics that an implementation must document. *Unspecified* is the same as implementation-defined, except that the choice need not be documented. For example, it is unspecified if an array parameter is passed by copy or by reference §6.2(11).

implicit declaration §3.1(5) See *declaration*.

implicit dereference §4.1(6) See *dereference*.

incomplete type declaration §3.10.1 Recursive type definitions such as Node are implemented by declaring just the type name (and discriminants, if any). This incomplete type declaration must eventually be completed by a *full type declaration*.

indefinite subtype §3.3(23) There are three categories of indefinite subtypes:

- Unconstrained array subtypes, for example Matrix.
- Unconstrained discriminated record subtypes without defaults, for example Game_Board.
- Subtypes with unknown discriminants (including class-wide types):

  type Data_Structure(<>) **is private**;

All other subtypes, in particular all elementary types, are *definite*. If you declare an object of an indefinite subtype, you must supply a constraint either explicitly, or implicitly from an initial value or actual parameter.

internal call §9.5(3) See *external call*.

intrinsic §6.3.1(4) See *calling convention*.

invalid representation §13.9.1(2) See *abnormal object*.

left §7.6.1(3) See *completion*.

library unit §10.2 See *unit*.

limited type §7.5 A type for which neither assignment nor predefined equality are allowed. The reserved word **limited** can be used to explicitly denote that a record or private type is limited. In addition, task and protected types are limited.

```
task type Producer;                    -- Limited type
type R is                              -- Limited type because ...
  record
    P: Producer;                       -- ...this component is limited
  end record;
```

master §7.6.1(3) See *completion*.

mode §6.1(18) An indication of the direction of the data flow from the actual parameter to the formal parameter and back. There are three modes: **in**, **out** and **in out**, which allow the subprogram to read from the actual parameter, write to the actual parameter, or both, respectively.

name §4.1(2) A name denotes a declared entity such as a variable or a component of a record. A *direct name* is either an identifier or an operator symbol such as "+". A name can also be a *prefix* of another name. An implicit dereference can also be a prefix:

```
N: Node;
Head: Ptr := new Node;
```

N and Head are direct names; Head.**all** is a name that is an explicit dereference. In the name N.Key, N is a prefix that is a name, while in the name Head.Next, Head is a prefix that is an implicit dereference.

named number §3.3.2 See *object*.

nominal subtype §3.3(23) The subtype specified for (a view of) an object when the view is defined. The *actual subtype* of the object is determined when the object is created.

```
type Vector is array(Integer range <>) of Float;
procedure Proc(Parm: in Vector);
V: Vector(5..10);
Proc(V);
```

The nominal subtype of the formal parameter Parm is Vector, but during the call Proc(V), the actual subtype is Vector(5..10).

numeric type §3.5(1) The numeric types are the integer and real types. Real types can be either fixed point types (ordinary or decimal) or floating point types.

object §3.3 An entity created at run-time to contain a value. Informally, objects describe memory locations such as variables. A list of objects is given in §3.3(2-12). Objects may be *variable* objects, or *constant* objects whose value cannot be changed. A list of constants is given in §3.3(16–22). Note that a named number §3.3.2 is not a constant object; it is just a compile-time name for a static value of universal type. A view of an object may also be constant, even if the object isn't:

```
Translate_Table: constant array(Character) of Character := ...;
Stack_Size: constant := 500;                    -- Named number, not a constant
procedure P(X: in Parent) is ...                -- View conversion is constant
```

operator §6.6 A function whose name is one of the operator symbols listed in §4.5(2–7), such as "+" and "=". Note that **in**, **not in**, **and then** and **or else** are not operators and cannot be overloaded.

overloading §8.3(6) A set of declarations is overloaded if they have the same name and are directly visible at some place in the program. Name Resolution Rules are used to resolve a reference, that is, to choose one of the possible interpretations §8.6(10–15). Preference can be given to *root* types §8.6(29). Overload resolution is done by examining a *complete context* §8.6(4). A complete context should have only one acceptable interpretation; otherwise it is ambiguous §8.6(30). For example, the procedure call statement Ada.Text_IO.Put("Hello world") is a complete context, and the possible interpretations are the Put procedures for characters and for strings. The interpretation of the statement as a call to Put for strings is chosen because "Hello world" is the *expected type* §8.6(20) for that procedure.

overriding §8.3(9) A declaration of a subprogram with the same name as an implicit declaration of an inherited primitive subprogram overrides the implicit declaration.

parent type §3.4(1) See *derived type*.

part §3.2(6) See *subcomponent*.

partial view §7.3(4) The declaration of a private type declares a partial view of the type; its *full view* is given by the full declaration in the private part:

```
package P is
   type Parent is private;
   type Not_Derived is new Parent with ...;          -- Error
private
   type Parent is tagged null record;
   type Derived is new Parent with ...;              -- OK
end P;
```

The partial view of Parent is not tagged, so the extension in the declaration of Not_Derived is illegal, even though Parent is 'really' tagged as shown in the full view.

partition §10.2(2) An Ada *program* is a set of partitions that can be executed in a distributed environment by assigning each partition to a node. You can think of a partition as a 'program' running on a computer.

per-object constraint §3.8(18) A constraint in a component of a discriminated record is called a per-object constraint if it *depends on a discriminant*. If so, it is evaluated only when the object is created, not when the type declaration is elaborated.

In the following example, the discriminant D is a *per-object expression*. The constraint of Field1 is elaborated when the type declaration is elaborated, and is 1..11 for all objects. The per-object constraint of Field2 is elaborated when an object is elaborated. For the object R, the constraint is 1..10, which is too small for the value in the aggregate, so Constraint_Error will be raised.

```
G: Integer := 11;
type Rec(D: Integer) is
   record
      Field1: String(1..G);                  -- Not a per-object constraint
      Field2: String(1..D);                  -- Per-object constraint
   end record;
R: Rec := (10, "Hello world", "Hello world");
```

pool-specific §3.10(8) See *storage pool*.

potentially blocking §9.5.1(8) It is a bounded error for a protected action to invoke an operation such as a call to a task entry that could cause the task executing the action to block.

prefix §4.1(4) See *name*.

primitive operation §3.2.3 The primitive operations on a type include the predefined operations on the type as well as user-defined *primitive subprograms*, which are subprograms that have a formal parameter or result of the type. Inherited and overriding subprograms are also primitive operations.

private part §7.1(6) See *visible part*.

profile §6.1(22) The profile of a callable entry such as a subprogram or entry is its interface as defined by the declaration of the formal parameters. For a function, the profile includes the result type.

protected action §9.5.1(3) The execution of a protected subprogram or entry, including acquiring and releasing the lock.

range §3.5(4) A subset of a *scalar type* defined by a lower and an upper bound. A *discrete range* or *discrete subtype indication* is specified by giving either a range or by a subtype indication of a discrete subtype:

```
for P in Piece loop ...                    -- subtype indication
for P in Knight..Rook loop ...             -- range
```

representation item §13.1(7) An object is represented in memory by a string of bits. A representation item specifies aspects of an object's representation, such as the size, alignment and placing of the object.

root type §3.4.1(8) See *universal type*.

root library unit §10.1.1(1) See *unit*.

scalar type §3.2(3) See *elementary type*.

scope §8.2 The scope of a declaration is the portion of the program where the declaration might be *visible* §8.3. The declaration is visible if it can be referred to; it may be *directly visible* §8.3(4) either because it is *immediately visible* and can be referred to by a direct name, or because it is *use-visible* §8.4(9). A declaration is use-visible if it has been made *potentially use-visible* by a 'use' clause §8.4(8) and there are no name conflicts §8.4(10–11). Within its scope, a declaration may be *hidden* §8.3(5), either from direct visibility, or from all visibility.

```
type Index is Integer range 1..100;
type T(D1: Index := 10; D2: Index := D1) is ...
  -- Error, D1 is hidden from all visibility §8.3(19)

Head: Ptr := new Node;
First: Integer := Head.Key;
  -- Key is visible but not directly visible §8.3(2)
```

semantic dependencies §10.1.1(24) One compilation unit may depend on another; this dependency is used to determine the correctness of a unit and the visibility of declarations. Semantic dependencies also determine allowable sequences of compilation. *Elaboration dependencies* are the same as semantic dependencies, except that they can be changed by using pragmas §10.2(9). For example, a package body depends on the corresponding package declaration, and must be compiled and elaborated after it.

short-circuit control form §4.5.1(1) The constructs **and then** and **or else** do not evaluate their second operand if the value of the form can be determined from the first one:

```
while P /= null and then P.Key /= Value loop
  -- If P = null, don't evaluate P.Key
```

specific type §3.4.1(3) See *class-wide type*.

static §4.9 An static expression is one whose value can be determined at compile-time. Similarly, a subtype whose constraint can be determined at compile-time is called a *static subtype* §4.9(26). The choices of a case statement must be static §5.4(5).

```
procedure Proc(X: Piece) is
  subtype R1 is Piece range Pawn..Bishop;          -- Static
  subtype R2 is Piece range Pawn..X;               -- Not static
begin
  case P is
    when Pawn     => ...;                          -- OK
    when X        => ...;                          -- Error
    when R1       => ...;                          -- OK
    when R2       => ...;                          -- Error
  end case;
end Proc;
```

statically match §4.9.1 Certain language constructs require that a pair of subtypes or constraints be statically matching. This means that the compiler can determine that they are the same: either they are both static and equal, or they are both derived from the same definition. Array type conversion is permitted only if the subtypes of the components statically match §4.6(12) (and the index subtypes are convertible):

```
procedure Proc(N: in Integer) is
  subtype R is Integer range 1..N;
  type AT1 is array(1..10) of R;
  type AT2 is array(1..10) of R;
  type AT3 is array(1..10) of Integer range 1..N;
  A1: AT1;
  A2: AT2;
  A3: AT3;
begin
  A1 := AT1(A2);                   -- OK, statically match
  A1 := AT1(A3);                   -- Error, do not statically match
end Proc;
```

statically tagged §3.9.2(4) See *dynamically tagged*.

storage element §13.3(8) A storage element is an addressable element of memory. System.Storage_Unit is the number of bits in a storage element. On most computers, storage elements are eight-bit bytes.

```
for Ptr'Storage_Size use 1000 * (Node'Size / System.Storage_Unit);
  -- Storage pool for 1000 nodes
```

storage pool §13.11(2) Allocation of memory using **new** is done from an area of memory called the storage pool associated with a *pool-specific* access type §3.10(8). Normally, all storage

is allocated from one or more *standard pools*, but the user can define other pools and memory allocation schemes. A object of a *general* access type can contain a value of any access type with the same designated type.

subcomponent §3.2(6) A subcomponent is a *component* of an object of composite type, or recursively a component of a subcomponent. A *part* is an entire object or any set of subcomponents.

```
Chess_Board: Game_Board(8);
Chess_Board.B;                    -- Component and subcomponent
Chess_Board.B(3, 4);              -- Subcomponent
```

subsystem §10.1(3) See *unit*.

subtype §3.2(8) A type together with a *constraint* on the values of the type. The subtype of an entity is determined at run-time; violations of subtype matching cause Constraint_Error.

```
subtype Strong is Piece range Knight..Queen;
Current: Strong := Pawn;          -- Compiles OK, raises Constraint_Error
```

subunit §10.1.3 A separately compiled physical unit that is a body corresponding to a *stub* in the parent unit.

```
package body P is
  Title: constant String := ...;
  procedure Display is separate;                    -- Stub
end P;

with Ada.Text_IO;                 -- Subunit can have context clause
separate(P)
procedure Display is
begin
  Ada.Text_IO.Put(Title);         -- Subunit retains visibility
end Display;
```

tag indeterminate §3.9.2(6) See *dynamically tagged*.

target object §9.5(2) The called task or protected object in an entry or protected subprogram call.

terminated §9.3(5) The state of a task after it has been *completed* and *finalized*.

type §3.2(1) A set of values and *primitive operations* on these values. The type of an entity is determined at compile-time.

unconstrained subtype §3.2(9) A subtype that allows constraints, but for which no constraints have been defined.

```
type Game_Board(Size: Positive) is ...      -- Unconstrained subtype
subtype Chess_Board is Game_Board(8);       -- Constrained subtype
```

Subtypes with unknown discriminants are also unconstrained §3.7(26). Integer'Base and discriminated records with defaults are examples of unconstrained subtypes that are not *indefinite*; uninitialized objects of these types can be declared.

unit §10.1(1) Ada programs (more exactly, *partitions*) are composed of (possibly nested) units such as packages, tasks and subprograms. A *compilation* consists of a set of one or more *compilation units*, which are either *library units* or *subunits*. A library unit is a unit not physically nested within another (except Standard). Library units that are not children of another package except Standard are called *root* library units §10.1.1(1).

A unit may be a *parent unit* of a *child unit* §10.1.1(1), and the terms *ancestor* and *descendant* §10.1.1(11) are used transitively for this relation. A root library unit and its descendants form a *subsystem* §10.1(3).

universal type §3.4.1(6) A universal type is the conceptual class-wide type for a class of numeric types. The conceptual specific type at the root of a universal type is called a *root type* §3.4.1(8).

unspecified §1.1.3(18) See *implementation defined*.

usage name §3.1(10) See *defining name*.

view §3.1(7) A declaration defines a view, which is a way of 'looking at' an entity. There may be several views for the same entity:

- A private type is a partial view of the full type whose definition is given in its completion §7.3(4).
- A formal by-reference parameter is a view of the actual parameter §6.2(2).
- A type conversion to a tagged type is a view of the object §4.6(5,26). In the following example, the type conversion to Parent is a view conversion, so within the subprogram we can convert the parameter to Parent'Class and dispatch:

```
D: Derived;
Primitive(Parent(D));
```

visible §8.3 See *scope*.

visible part §7.1(6) The declarations in a package specification before the keyword **private** (or before the end of the specification if there is no private part) form the *visible part* of the specification. Declarations following **private** (if any) form the *private part* of the specification.

C Source Code

The source code for the examples and case studies is organized in a directory structure corresponding to the chapters of the book. The file LIST.TXT contains a list of each file and its contents. The name of the main subprogram (or quiz number) with extension ADA is used as the file name.

If you use GNAT, you will have to run 'gnatchop' to create separate files for each compilation unit.

```
gnatchop -w euler.ada
gnatmake euler
```

This step is not necessary if your compiler allows multiple compilation units per file.

D Quizzes

The quizzes will help you understand the finer points of the Ada language; they also offer an opportunity to practice reading the *ARM*. Each quiz is a small Ada program; you are to examine it and decide if it will compile, and if so, what will be the result of executing the program (for quizzes that have executable statements). The quizzes are grouped according to topic, but this grouping is only approximate, because a quiz may show the interaction between several constructs.

Appendix E Hints refers you to the relevant clauses of the *ARM* from which you should be able to solve many, if not most, of the quizzes *before* looking at Appendix F Answers!

The source code of the quizzes is on the CD-ROM. To save space in the book, 'with' and 'use' clauses for Ada.Text_IO, Ada.Integer_IO and Ada.Exceptions are omitted. Furthermore, if the source code of the quiz consists only of a main subprogram with declarations and statements, the following program structure is assumed:

```
procedure Main is
  -- Declarations
begin
  -- Statements, or null if none
end Main;
```

Types

1.
```
type Rec is
  record
    V: String;
  end record;
```

2.
```
S: String := "Hello world";

S(2..5) := S(1..4);
Put(S);
```

3.
```
procedure Proc(Stop: Character) is
  Start: constant Character := 'A';
  type R(C: Character) is
    record
      case C is
        when Start .. Character'Succ(Start) => I: Integer;
        when Stop => B: Boolean;
        when others => F: Float;
      end case;
    end record;
begin
  null;
end Proc;
```

4.
```
subtype Letters is Character range 'A'..'Z';

Put(Letters'Val(42));
Put(Positive'Val(0));
```

5.
```
type T1 is range 5..10;
type T2 is new T1 range 50..100;
```

Aggregates

6.
```
type Rec is
  record
    One, Two, Three, Four, Five: Integer;
  end record;
R: Rec := (1, 2, Four=>4, Five=>5, Three=>3);
type Vector is array(1..5) of Integer;
V: Vector := (1, 2, 3=>3, 4=>4, 5=>5);
```

7.
```
package P is
  type T1 is tagged
    record I: Integer := 0; end record;
  type T2 is new T1 with
    record N: Integer := 0; end record;
end P;
with P; use P;
procedure Main is
  A: T2'Class := (I => 2, N => 4);
begin
  null;
end Main;
```

8.
```
type Name_Array is
  array(Integer range <>, Integer range <>) of Character;
Names: constant Name_Array(1..4, 1..6) :=
  ("Kirk␣␣", "Spock␣", "McCoy␣", "Scotty");

Put(Names(4,1));
```

9.
```
type Vector is array(Integer range <>) of Integer;
V1: Vector(1..5) := (6..10 => 0);
V2: Vector(1..5) := (6 => 1, others => 0);
```

10.
```
N: Integer;
procedure P(T: String) is
begin
  Put(T'Last);
end P;

Get(N);
P((1..N => 'X'));
P((1..10 => 'X', 11..N => 'Y'));
```

Boolean types and equality

11.
```
X, Y, Z: Boolean;
B1: Boolean := X and Y and Z;
B2: Boolean := X and Y or Z;
```

12.
```
package P is
  type T1 is tagged
    record I: Integer := 0; end record;
  function "="(Left, Right: T1) return Boolean;
  type T2 is new T1 with
    record N: Integer := 0; end record;
end P;

package body P is
  function "="(Left, Right: T1) return Boolean is
  begin
    return abs(Left.I-Right.I) < 2;
  end "=";
end P;
```

```
with P; use P;
procedure Main is
  A: T2 := (I => 2, N => 4);
  B: T2 := (I => 3, N => 4);
  C: T2 := (I => 3, N => 4);
  D: T2 := (I => 3, N => 5);
begin
  Put_Line(Boolean'Image(A = B));
  Put_Line(Boolean'Image(C = D));
end Main;
```

13.

```
  type My_Boolean is new Boolean;
  function "="(Left, Right: Integer) return My_Boolean is
    B: Boolean := Left = Right + 1;
  begin
    return My_Boolean(B);
  end "=";
  function "/="(Left, Right: Integer) return My_Boolean is
    B: Boolean := Left = Right - 1;
  begin
    return My_Boolean(B);
  end "/=";
  M1: My_Boolean := 4 = 3;
  M2: My_Boolean := 3 /= 4;

  Put_Line(My_Boolean'Image(M1));
  Put_Line(My_Boolean'Image(M2));
```

14.

```
  function "/="(Left, Right: Integer) return Boolean is
    B: Boolean := Left = Right - 1;
  begin
    return B;
  end "/=";

  Put_Line(Boolean'Image(4 /= 3));
```

15.

```
  type T1 is null record;
  X, Y: T1 := (null record);

  Put(Boolean'Image( X = Y ));
```

16.

```
  type My_Boolean is new Boolean;
  I, J: Integer := 1;
  M: My_Boolean := My_Boolean(I = J);
  if M then Put("Equal"); else Put("Not equal"); end if;
```

17.
```
type T is new Integer;
function "="(Left, Right: T) return Boolean is
begin
  return Integer(Left) /= Integer(Right);
end "=";

type Vector is array(1..2) of T;
A: Vector := (1, 2);
B: Vector := (3, 4);

Put_Line(Boolean'Image(A(1) = B(1)));
Put_Line(Boolean'Image(A = B));
```

Discriminants

18.
```
type Rec(D: Positive := 100) is
  record
    V: String(1..D);
  end record;
R: Rec;
```

19.
```
type Rec(D: Positive) is
  record
    V: String(1..D-1);
  end record;
```

20.
```
subtype Index is Integer range 1..100;
type Rec(Disc: Index := 100) is
  record
    Data: String(1..Disc);
  end record;
R1: Rec(100);
C1: Character renames R1.Data(100);
R2: Rec;
C2: Character renames R2.Data(100);
```

21.
```
subtype Sizes is Integer range 1..5000;
type Queue(Size: Sizes := 100) is tagged null record;
```

22.
```
type Parent(Number: Positive; Size: Positive) is
  record
    X: String(1..Number);
    Y: String(1..Size);
  end record;
type Derived(Count: Positive) is new Parent(Count, Count);
P: Parent := (2, 3, "ab", "cde");
D: Derived := (3, "uvw", "xyz");

D := Derived(P);
Put_Line(D.X); Put_Line(D.Y);
```

Numeric types

23.
```
type Int is range 0..10_000;
A: Int := 1_000;
B: Int := 20;
C: Int := (A * B) / 5;

Put_Line( Int'Image(C) );
```

24.
```
Put( 16#a#e2 );
Put( 16#A#E2 );
Put( 16#E#EA );
```

25.
```
Put( "+"(Left => 2, Right => 3) );
```

26.
```
type M is mod 23;
X1: M := 21;
X2: M := 10;

Put_Line( M'Image( X1 xor X2 ) );
```

27.
```
type Fixed is delta 0.5 range 0.0 .. 10.1;
Put_Line(Fixed'Image(10.1));
```

28.
```
N1: aliased constant Integer := 1;
N2: aliased constant := 1;
```

29.
```
   I: Integer;
   case Integer'Pos(I) is
     when Integer'First..Integer'Last => null;
   end case;
```

Access types

30.
```
package P is
   type Ptr is access all Integer;
end P;

with P;
procedure Main is
   N: aliased Integer;
   Q: P.Ptr := N'Access;
begin
   null;
end Main;
```

31.
```
package P is
   type Parent is tagged null record;
   type Parent_Ptr is access all Parent;
   procedure Proc1(X: Parent_Ptr);
   procedure Proc2(X: access Parent);
   type Derived is new Parent with null record;
   D: aliased Derived;
end P;

with P; use P;
procedure Main is
begin
   Proc1(D'Access);
   Proc2(D'Access);
end Main;
```

32.
```
   type String_Ptr is access String;
   S: String_Ptr := new String'("Hello world");
   type Integer_Ptr is access Integer;
   I: Integer_Ptr := new Integer'(10);

   Put_Line(S(1..5));
   Put(I);
```

33.
```
   type Int_Ptr is access all Integer;
   N: aliased Integer;
   function Func return Int_Ptr is
   begin
     return N'Access;
   end Func;
   Func.all := 5;
   Put(N);
```

34.
```
package P is
   protected type PT is
     entry E(X: access Integer);
   end PT;
   task type T is
     entry E(X: access Integer);
   end T;
end P;
```

35.
```
package P is
   type Rec(D: access Integer) is limited null record;
   type Ptr is access Rec;
end P;

with P; use P;
procedure Main is
   N: aliased Integer;
   R: Ptr := new Rec(N'Access);
begin
   null;
end Main;
```

36.
```
package P is
   type Parent is tagged null record;
end P;

with P; use P;
procedure Main is
   type Derived is new Parent with null record;
begin
   null;
end Main;
```

Parameters

37.
```
   N1: Integer := 5;
   type Ptr is access Integer;
   N2: Ptr := new Integer'(5);
   procedure Proc(K: out Integer; P: out Ptr) is
   begin
     Put_Line(Integer'Image(K));
     Put_Line(Integer'Image(P.all));
   end Proc;

   Proc(N1, N2);
```

38.
```
   type Vector is array(1..2) of Integer;
   Vec: Vector := (others => 1);
   procedure Proc(V: in out Vector) is
     V(1) := 2;
     if Vec(1) = 1
       then Put("One");
       else Put("Two");
     end if;
   end Proc;

   Proc(Vec);
```

39.
```
   procedure Proc(N1: in Integer; N2: in Integer := N1) is
   begin
     null;
   end Proc;
```

40.
```
package P is
   type T1 is range 1..100;
   procedure Proc(X: T1; Y: T1);
   type T2 is new T1 range 1..50;
end P;

package body P is
   procedure Proc(X: T1; Y: T1) is
   begin
     Put(Integer(X));
     Put(Integer(Y));
   end Proc;
end P;
```

```
with P; use P;
procedure Main is
  Z: T2 := 10;
begin
  Proc(Z, 99);
end Main;
----------
```

41.

```
  procedure Proc1(I: in Integer) is
  begin
    Put(I);
  end Proc1;
  procedure Proc2(I: in Positive) renames Proc1;

  Proc2(0);
----------
```

42.

```
package P is
  type T1 is range 1..100;
  subtype Sub is T1 range 1..50;
  procedure Proc(X: Sub; Y: Sub);
  type T2 is new T1 range 51..100;
end P;

package body P is
  procedure Proc(X: Sub; Y: Sub) is
  begin
    Put(Integer(X));
    Put(Integer(Y));
  end Proc;
end P;

with P; use P;
procedure Main is
  Z1: T2 := 88;
  Z2: T2 := 99;
begin
  Proc(Z1, Z2);
end Main;
----------
```

Packages

43.
```
package P is
  type T1 is private;
  C: constant T1;
  V: T1;
private
  type T1 is new Integer;
  C: constant T1 := 0;
end P;
```

44.
```
package P is
  Mode: Integer;
end P;

package body P is
begin
  Mode := 777;
end P;

with P; use P;
procedure Main is
begin
  Put(Mode);
end Main;
```

45.
```
package P is
  type Node;
  type Ptr is access Node;
end P;

package body P is
  type Node is
    record
      Key: Integer;
      Next: Ptr;
    end record;
end P;
```

46.
```
package P is
  procedure Proc1(I: in Positive);
  procedure Proc2(I: in Natural);
  procedure Proc3(I: in Natural) renames Proc1;
end P;
```

```
package body P is
  procedure Proc1(I: in Positive) is
  begin
    null;
  end Proc1;
  procedure Proc2(I: in Natural) renames Proc1;
end P;
----------
```

Visibility

47.

```
  N: Integer := 5;
  procedure Proc is
    N: Integer := 10;
  begin
    Put(N);
    Put(Main.N);
  end Proc;

  Proc;
----------
```

48.

```
  procedure Proc(X: in Integer; Y: in Integer := 2) is
  begin
    Put(X*Y);
  end Proc;

  procedure Proc(Z: in Integer; Y: in Float := 3.0) is
  begin
    Put(Z*Integer(Y));
  end Proc;
  procedure Proc(X: in out Integer) is
  begin
    Put(X);
  end Proc;

  Proc(Z => 4);
  Proc(X => 5);
----------
```

49.
```
package P is
  procedure Proc;
end P;

package body P is
  procedure Proc is
  begin
    Put_Line("Hi from Proc in the package");
  end Proc;
end P;

with P; use P;
procedure Main is
  procedure Proc is
  begin
    Put_Line("Hi from Proc in the main subprogram");
  end Proc;
begin
  Proc;
end Main;
----------
```

50.
```
package P is
  procedure Proc;
end P;

package body P is
  Inner: exception;
  procedure Proc is
  begin
    raise Inner;
  end Proc;
end P;

with P; use P;
procedure Main is
begin
  Proc;
exception
  when E: others => Put(Ada.Exceptions.Exception_Name(E));
end Main;
----------
```

51.

```
package P is
   type T is tagged null record;
   procedure Proc(X: T);
end P;

package body P is
   procedure Proc(X: T) is
   begin
      Put_Line("Parent");
   end Proc;
end P;

with P; use P;
package Q is
   type T1 is new T with null record;
private
   procedure Proc(X: T1);
end Q;

package body Q is
   procedure Proc(X: T1) is
   begin
      Put_Line("Derived");
   end Proc;
end Q;

with Q; use Q;
procedure Main is
   A: T1;
begin
   Proc(A);
end Main;
----------
```

52.

```
   function F return Integer is
   begin
      return 1;
   end F;
   function F return Float is
   begin
      return 1.0;
   end F;

   Put(F);
   Put(Integer(F));
----------
```

53. In procedure Tabbing on page 130, a ‘use’ clause was used rather than a ‘use-type’ clause. Why?

Type extension

54.
```
package P is
  type T1 is tagged null record;
  procedure Proc1(X: in out T1);
  type T2 is new T1 with null record;
  procedure Proc1(X: in out T2);
  procedure Proc2(X: in out T1);
end P;
----------
```

55.
```
package P is
  type Parent is tagged record N: Integer; end record;
  type Derived is new Parent with record M: Integer; end record;
end P;

with P; use P;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Main is
  P: Parent := (N=>1);
  D: Derived := (N=>2, M=>3);
begin
  Parent(D) := P;
  Put(D.N);
  Put(D.M);
end Main;
----------
```

56.
```
package P is
  type T is private;
private
  type Ptr is access T'Class;
  type T is tagged
    record
      Next: Ptr;
    end record;
end P;
----------
```

57.
```
package P is
  type T1 is tagged
    record I: Integer; end record;
  function F return T1;
  type T2 is new T1 with
    record N: Integer; end record;
end P;
```

58.
```
package P is
  type T1 is tagged
    record
      N: Integer;
    end record;
  procedure Proc2(X: T1 := (N=>1));
end P;
```

59.
```
package P is
  type T1 is tagged
    record
      N: Integer;
    end record;
  procedure Proc1(X: T1);
  Empty: constant T1;
  procedure Proc2(X: T1);
private
  Empty: constant T1 := (N => 1);
end P;
```

Tasks

60.
```
procedure Main is
  task type T(ID: Integer);
  task body T is
    N: Positive := ID;
  begin
    for I in 1 .. 4 loop
      Put_Line("Hi from task " & Integer'Image(ID));
    end loop;
  end T;

  T0: T(0);
  T1: T(1);
```

```
begin
  null;
exception
  when Tasking_Error => Put_Line("A task has died");
end Main;
----------
```

61.

```
procedure Main is
  task T is
    entry E1(N: Integer);
    entry E2;
  end T;
  task body T is
  begin
    accept E1(N: Integer) do
      select
        when N > 0 => accept E2;
      end select;
    end E1;
  exception
    when Ex: others =>
      Put_Line("Task T " & Exception_Name(Ex));
  end T;

  task A;
  task body A is
  begin
    T.E1(0);
  exception
    when Ex: others =>
      Put_Line("Task A " & Exception_Name(Ex));
  end A;

begin
  T.E2;
exception
  when Ex: others =>
    Put_Line("Main " & Exception_Name(Ex));
end Main;
----------
```

62.

```
task type T is
  entry E;
end T;

task body T is
  procedure P is
  begin
    accept E;
  end P;
begin
  begin
    accept E do
      accept E;
    end E;
  end;
end T;
----------
```

63.

```
procedure Main is

  task type T(ID: Integer);
  type Ptr is access T;
  task body T is
  begin
    for I in 1 .. 5 loop
      Put_Line("Hi from task " & Integer'Image(ID));
      delay 0.5*ID;
    end loop;
  end T;
begin
  Put_Line("Main subprogram");
  declare
    P: Ptr := new T(1);
  -- type New_Ptr is access T;
  -- New_P: New_Ptr := new T(2);
  begin
    null;
  end;
  Put_Line("Back from block");
end Main;
----------
```

What would happen if the commented lines were added to the program?

64.
```
   protected P is
     entry E1;
     entry E2;
   end P;
   procedure Proc renames P.E1;
   protected body P is
     entry E1 when True is
     begin
       null;
     end E1;
     entry E2 when True is
     begin
       Proc;
       requeue Proc;
     end E2;
   end P;
----------
```

65.
```
procedure Main is

   protected PO is
     entry E;
   private
     C: Character := Character'First;
   end PO;
   protected body PO is
     entry E when Character'Pred(C) < 'A' is
     begin
       null;
     end E;
   end PO;

begin
   PO.E;
exception
   when Ex: others =>
     Put_Line("Main " & Exception_Name(Ex));
end Main;
----------
```

66.
```
procedure Main is

  task T1 is
    entry E;
  end T1;
  task body T1 is
  begin
    delay 0.2;
    accept E do
      Put_Line("Starting T1.E");
      delay 0.8;
      Put_Line("Finishing T1.E");
    end E;
  end T1;

  task T2 is
    entry E;
  end T2;
  task body T2 is
  begin
    delay 0.4;
    accept E do
      Put_Line("Starting T2.E");
      delay 0.1;
      Put_Line("Finishing T2.E");
    end E;
  end T2;

begin
  select
    T1.E;
    Put_Line("Finished triggering");
  then abort
    T2.E;
    Put_Line("Finished abortable");
  end select;
end Main;
----------
```

67.
```
task type T;
type Ptr is access T;

function F return Ptr is
begin
  return new T;
end;

task body T is
  X: Ptr := F;
  Y: Ptr := new T;
begin
  null;
end T;

Z: Ptr := new T;
```

Generics

68.
```
generic
  type Item is private;
  type Vector is array(1..10) of Item;
procedure Proc;
```

69.
```
package P is
  Local_Max: Integer := 100;
  generic
    Max: Integer := Local_Max;
  package GP is
    procedure Proc;
  end GP;
end P;

package body P is
  package body GP is
    procedure Proc is
    begin
      Put(Max);
    end Proc;
  end GP;
end P;
```

```
with P;
procedure Main is
  package First_GP is new P.GP;
  function Init return Integer is
  begin
    P.Local_Max := 200;
    return 1;
  end Init;
  N: Integer := Init;
  package Second_GP is new P.GP;
begin
  First_GP.Proc;
  Second_GP.Proc;
end Main;
----------
```

70.

```
  generic
    type T(A: Integer) is private;
  procedure Proc;
  procedure Proc is
  begin
    null;
  end Proc;

  type Base(D: Integer) is null record;
  type R(Disc: Integer) is new Base(D => Disc);
  type S is new Base(D => 10);

  procedure Proc1 is new Proc(T => R);
  procedure Proc2 is new Proc(T => S);
----------
```

71.
```
generic
  type T is range <>;
  with function Formal(Left, Right: T) return T;
package GP is
  function Func(Left, Right: T) return T;
end GP;

package body GP is
  function Func(Left, Right: T) return T is
  begin
    return 2*(Left+Right);
  end Func;
end GP;

with GP;
generic
  with package GFP is new GP(<>);
  -- with package GFP is new GP(Integer, "+");
procedure Proc;

procedure Proc is
  package GIO is new Ada.Text_IO.Integer_IO(GFP.T);
begin
  GIO.Put(GFP.Func(1,2));
  GIO.Put(GFP.Formal(1,2));
end Proc;

with GP; with Proc;
procedure Main is
  function Actual(Left, Right: Long_Integer) return Long_Integer is
  begin
    return 4*(Left+Right);
  end Actual;
  package GP_Instance is new GP(Long_Integer, Actual);
  procedure Proc_Instance is new Proc(GP_Instance);
begin
  Proc_Instance;
end Main;
----------
```

What would happen if the generic formal package parameter were changed to the commented line?

72.

```
package Q is
   type Parent is null record;
   procedure Proc(A: in Parent);
   type Actual is new Parent;
   procedure Proc(A: in Actual);
end Q;

with Ada.Text_IO; use Ada.Text_IO;
package body Q is
   procedure Proc(A: in Parent) is
   begin
      Put_Line("Parent");
   end Proc;
   procedure Proc(A: in Actual) is
   begin
      Put_Line("Actual");
   end Proc;
end Q;

with Q;
generic
   type Formal is new Q.Parent;
package P is
   type Derived is new Formal;
   procedure Inside;
end P;

package body P is
   procedure Inside is
      D: Derived;
   begin
      Proc(D);
   end Inside;
end P;

with P; with Q;
procedure Main is
   package Instance is new P(Q.Actual);
   D: Instance.Derived;
begin
   Instance.Inside;
   Instance.Proc(D);
end Main;
----------
```

E Hints

1. §3.6(10), §3.3(23)
2. §5.2(7,13)
3. §4.3.1(17), §4.9(6,21–22)
4. §3.5.1(7), §3.5.4(15), §3.5.5(6)
5. §3.5(9),§3.4(26)
6. §4.3.3(3–4), §4.3.1(4-6)
7. §4.3(4), §3.9(29)
8. §4.3.3(19)
9. §4.3.3(27), §4.3.3(26,29)
10. §4.3.3(17)
11. §4.4(2–7)
12. §3.4(17), §4.5.2(14)
13. §6.6(6)
14. §6.6(5)
15. §4.3.1(15), §4.5.2(22)
16. §3.5.3(1), §5.3(4)
17. §4.5.2(24)
18. None
19. §3.8(12)
20. §8.5.1(5)
21. §3.7(11)
22. §4.6(43)
23. §3.5(6), §3.5.4(21)
24. §2.4.1(6), §2.4.2(7–8)
25. §4.5(9)
26. §4.5.1(5)
27. §3.5.9(13)
28. §3.3.1(2), §3.3(24)
29. §3.5.5(3), §5.4(8)
30. §3.10.2(6,22), §10.2(8)
31. §3.4(18), §6.1(27)
32. §4.1(12)
33. §3.3(9)
34. §9.5.2(13)
35. §3.10.2(28)
36. §3.9.1(3)
37. §6.4.1(12–15)
38. §6.2(11)
39. §6.1(21)
40. §3.4(19)
41. §8.5.4(7)
42. §3.4(6,19,34)
43. §7.3(5)
44. §7.2(4)
45. §3.10.1(3)
46. §8.5.4(4–5)
47. §8.3(22), §4.1.3(4)
48. §6.4(6,9), §6.4.1(5), §8.6(2,14)

49. §8.4(10)
50. §11.4.1(12)
51. §3.2.3(7), §8.1(9)
52. §8.6(7,27)
53. §8.4(8)
54. §13.14(7,16), §3.9.2(13)
55. §3.3(12), §4.6(5), §5.2(5)
56. §7.3.1(8–9)
57. §3.9.3(4,6)
58. §3.9.2(11)
59. §13.14(6)
60. §9.2(1,5), §9.3(5)
61. §9.7.1(21), §9.5.2(24), §9.5.3(21)
62. §9.5.2(14–15), §5.6
63. §9.3(2)
64. §8.5.4(7)
65. §9.5.3(7)
66. §9.7.4(6,9), §9.5.3(20)
67. §8.6(17)
68. §12.5.3(3)
69. §12.1(10), §12.3(20)
70. §3.4(6), §12.5.1(11–14)
71. §12.7(10)
72. §12.3(16–17)

F Answers

1. Compile-time error. A component must be of a definite subtype; an unconstrained array type is indefinite .

2. Prints 'HHell world'. There is no 'overlap' because both sides of the assignment statement are evaluated *before* the target variable receives the value of the expression.

3. Compile-time error. The discrete choices in the variant must be static: Character'Succ(Start) is static, but Stop is not.

4. Prints an asterisk that is at position 42 in the enumeration type Character and then prints zero. The position of an enumeration value is its position within the type, not the subtype, and the position of an integer is its value. Val returns a value of the *base* type.

5. This is *not* a compile-time error; instead, Constraint_Error is raised at runtime!

6. Compile-time error. While record aggregates can have named associations after positional associations, array aggregates must be either positional or named (except for **others**).

7. Compile-time error. An aggregate cannot be of class-wide type. Qualification T2'(I=>2, N=>4) should be used to give the aggregate a specific type.

8. Prints 'S'. The bottom subaggregate can be a string literal if the component is of type Character.

9. Raises Constraint_Error. The declaration of V1 is legal: the aggregate bounds are taken from the discrete choice list and then converted during the assignment ('sliding'). Because of the **others**, the bounds of the aggregate for V2 are taken from the index constraint, and 6 is not within the bounds of the constraint. Aggregates with **others** do not slide.

10. Compile-time error. The second aggregate is illegal, because a choice may not be dynamic unless it is the only choice.

11. Compile-time error. The declaration of B1 is legal because repetitions of **and** (or **or**) are allowed by the syntax. The declaration of B2 is illegal because combining operators requires the syntax of a parenthesized expression.

12. The first statement prints 'True' because the first components 2 and 3 of A and B are equal by the overridden equality function, and the second components, which are both 4,

are equal by predefined equality which is used on non-inherited components. The second statement prints 'False' because 4 is not equal to 5 using predefined equality. We say that predefined equality is *incorporated*. Without the special rule, the second component N would be *ignored* by the inherited equality and both statements would print 'True'.

13. Prints 'True' twice! Except for predefined Boolean type, there is no relationship between "=" and "/=".

14. Compile-time error. An explicit declaration of "/=" for predefined Boolean is illegal.

15. Prints 'True'. Equality of a composite types returns true if there are no components.

16. Prints 'Equal'. The descendant of Boolean is also a *boolean type* and the condition of an if statement can be any boolean type.

17. Prints 'True' then 'False'. Predefined, rather than overridden equality, is used for untagged components of a composite type.

18. Raises an exception such as Storage_Error. The object is allocated the maximum space that might be needed by any value of the discriminant subtype.

19. Compile-time error. A discriminant must be used directly in a constraint, not as part of an expression.

20. Compile-time error. The declaration of C2 is illegal, because you can't rename something that might not exist.

21. Compile-time error. Default expressions are not allowed for discriminants of tagged type.

22. The declaration of D raises Constraint_Error. When a discriminant of the target type corresponds to more than one discriminant of the operand, they must both be equal.

23. Prints '4000'. The expression is evaluated using the *base range*, which is sufficient to hold 20_000 on any computer with at least a 16-bit word.

24. Compile-time error. The third statement is a parse error because the exponent must be decimal. With the third statement deleted, the program prints 2560 ($= 10 \cdot 16^2$) twice: exponent letters and hexadecimal digits can be of either case.

25. Prints '5'. The implicit parameter names of the predefined binary operators are Left and Right.

26. Prints '8': 01010 **xor** 10101 = 11111 = 31 and 31 **mod** 23 = 8.

27. The program is correct even if it prints 10.0. The bounds of fixed point type are not necessarily values of the type!

28. Compile-time error. The first declaration declares an aliased object. The second declaration declares a named number, which is a value not an object that is allocated storage and can be aliased.

29. Compile-time error. The attribute Pos returns a value of type *universal integer*. A case statement needs an **others** alternative if the expression is of universal type.

30. Compile-time error. Ptr is at library level and Main is deeper than the master which calls it, namely the library level environment task.

31. Compile-time error. The call to Proc1 is a call to an unknown subprogram. Parent_Ptr is not of type Parent, so it is not inherited by the derived type. The call to Proc2 is legal, because the profile of a subprogram includes the designated subtype of an access parameter.

32. Compile-time error. An implicit dereference is a prefix, thus S(1..5) returns a slice of a string, which is an acceptable parameter for Put_Line. However, there is no implicit dereference of I and a pointer is not an acceptable parameter for Put. Put(I.**all**) would make the subprogram call legal.

33. Prints 5. The dereference is a variable, and it does not matter that the access object is returned by a function. However, you cannot assign directly to the function result, because it is a constant §3.3(21).

34. Compile-time error. An entry for task cannot have an access parameter, because implementation of accept statements would be too difficult. An entry for a protected object is not subject to this restriction.

35. Compile-time error. The view D of N is of the accessibility level of the main program, which is deeper than the library access level of the anonymous type of the discriminant.

36. Compile-time error. A type extension may not be at a deeper accessibility level than the parent.

37. K is uninitialized, so the first statement will print garbage or whatever the default initialization of an integer is. The second statement will print 5 because P is initialized to the actual parameter. The special rule exists so that uninitialized pointers will never exist, as this can break the type system.

38. Prints 'One' if by copy and 'Two' if by reference, depending on your implementation.

39. Compile-time error. A formal parameter cannot be used in the formal part.

40. Prints '10 99'. The subtypes of the subprogram for the derived type are taken from the parent type.

41. Prints '0'. The subtype is taken from the renamed procedure, not the renaming declaration.

42. Raises Constraint_Error. There is no way to call the inherited subprogram, because the parameters are constrained by 1..50, while the derived values are constrained by 51..100.

43. Compile-time error. An object cannot be declared before the full declaration of its type.

44. The body is not allowed unless needed; if you delete the body, the program prints garbage or whatever the default initialization of an integer is. Use pragma Elaborate_Body §10.2.1(26) in the specification to require that the package have a body.

45. Compile-time error. The completion of an incomplete type declaration can be in the body only if the incomplete declaration is in the *private* part.

46. Compile-time error. The declaration of Proc2 is a renaming-as-body which must be subtype conformant. The declaration of Proc3 is correct because mode conformance is sufficient for a renaming-as-declaration.

47. Prints '10 5'. The homograph is hidden from direct visibility, but not from all visibility. An expanded name can be used to access the outer declaration.

48. Compile-time error. Overloading resolution can make use of the names in a named association §6.4(9), so the first call is unambiguous. However, the only way to disambiguate the second call is to note that the actual parameter of an **out** or **in out** formal parameter must be a variable. But this is a legality rule, *not* an overload resolution rule §6.4.1(5). §8.6(14) only requires that a possible interpretation satisfy the syntax rule—here §6.4(6). So overloading fails even before the legality is checked §8.6(2).

49. Prints 'Hi from Proc in the main subprogram'. A use clause will not cause ambiguity with a homograph declared directly in a declarative region.

50. Prints 'P.Inner'. The function Exception_Name returns the fully expanded name of the exception even though the exception itself is not visible.

51. Prints 'Derived' even though the overridden subprogram is not visible! Overriding subprograms need not be declared in the visible part of the specification, only in the same *declarative region* as the parent type, which includes the private part and child packages.

52. Compile-time error. The second call is ambiguous because a type conversion is not a context for overload resolution. The first statement by itself would print 1, because the call to Ada.Integer_Text_IO.Put is a context for overloading resolution.

53. String is declared in package Standard, so the operator declared in this package is not a *primitive* operator of String. 'Use-type' clauses only make primitive operators potentially use-visible.

54. Compile-time error. The declaration of the extension T2 freezes the type T1, and a primitive subprogram such as Proc2 cannot be declared after the type is frozen.

55. Prints '1 3'. A type conversion to a tagged type is a view conversion, and a view conversion of a variable is a variable as required for the target of an assignment statement.

56. The specification is legal. Even though the partial view of T untagged, you can apply the attribute T'Class provided that the full view is tagged.

57. Compile-time error. The function must be overridden, otherwise the inherited function would return a value of type T1, which cannot be assigned to an object of type T2!

58. Compile-time error. A default expression must be tag indeterminate.

59. A deferred constant does not freeze a type, so the declaration of Proc2 is legal.

60. Prints (in some order) 'A task has died' and four times 'Hi from task 1'. The exception in the elaboration of task T0 causes it to become completed and raise Tasking_Error in the main subprogram, but the main subprogram cannot terminate until *all* its dependents including T1 have terminated.

61. Prints 'Task T Program_Error' and then 'Task A Program_Error' and 'Main Tasking_Error' in some order. There is a Program_Error in T because there is no open alternative in the selective accept. The exception in the rendezvous between A and T is propagated to the caller A. Tasking_Error is raised in the main subprogram because the called task A completes before call is accepted.

62. Compile-time error. An accept statement must be *immediately within* a task body, not within a nested subprogram body such as P. Furthermore, an accept statement cannot contain another accept for the *same* entry. Note that a block is a statement, not a body, so the outermost accept in the task body is legal.

63. Prints 'Main subprogram' and then 'Hi from task 1' five times and 'Back from block' in some order. The termination of a dynamically allocated task depends on the master declaring the *access type*, in this case the main subprogram. The task is a 'garbage task' which runs but is not accessible after the termination of the block. If the commented lines are added, the block becomes the master containing the declaration of the access type New_Ptr and will await the termination of T(2). 'Back from block' will be printed only after the ten lines of output from the tasks.

64. Compile-time error. A renamed entry is a procedure, not an entry, so it cannot appear in a requeue statement.

65. Program_Error is propagated to the caller Main because an exception is raised when evaluating the barrier.

66. Because of the delay 0.2 in the body of T1, the call T1.E is queued; thus the abortable part is started, calling T2.E. When the delay expires, T1.E is accepted and prints 'Starting T1.E'. Because of the subsequent delay 0.8 within the accept statement, the call does not

finish before the call to T2.E is started and finishes, printing the lines 'Starting T2.E' and 'Finishing T2.E'. When the call T2.E and the rest of the abortable part finishes (printing 'Finished abortable'), an attempt is made to cancel T1.E, but by §9.5.3(20) the attempt fails since the call is not on an entry queue. Thus the triggering statement is not cancelled §9.7.4(9), and the rest of the triggering alternative is eventually executed, printing 'Finishing T1.E' and 'Finish triggering'.

67. Compile-time error. The use of T within the declarative region of T denotes the current instance, not the type, so in the declaration of Y is not legal. The uses of T within the function F and the object declaration Z are legal because they are not within the declarative region of T.

68. Compile-time error. The index of a generic formal array type parameter must be a subtype mark.

69. Prints 100 then 200. Generic parameter associations are evaluated when instantiated, not when elaborated.

70. Compile-time error. The discriminated record R is unconstrained, but S is constrained by a discriminant constraint. Since T is a generic formal parameter with a known discriminant part, the instantiation of Proc1 is legal because R is unconstrained and supplies a discriminant, while the instantiation of Proc2 is not legal because S is constrained without a discriminant.

71. Prints '6 12'. The formal part of the formal package parameter (function Formal) is visible only if the formal package actual part is (<>). If the actual part supplies parameters as in the commented instantiation, a compile-time error results because Integer and "+" are used, and the formals T and Formal are not visible.

72. Prints 'Parent', then 'Actual'. The copied operations from Formal are the only ones visible within the instantiation. Outside the instantiation, the 'whole new set' is visible and can be overridden.

G Further Reading

G.1 Hard copy

The *Ada Reference Manual* and *Rationale* have been published in book form in Taft & Duff (1997) and Barnes (1997), respectively.

If you find this book too difficult, you will want to start with an elementary Ada textbook such as English (1997) or Feldman & Koffman (1996).

'Competing' advanced textbooks are Barnes (1995) and Cohen (1996). Cohen has detailed comparisons between Ada 83 and Ada 95, which will be useful if you have had extensive experience in Ada 83.

Textbooks on concurrent programming are Ben-Ari (1990), Burns & Davies (1993) and Andrews (1991). Burns & Wellings (1995) is entirely devoted to concurrency in Ada 95. You may also want to read some of the articles in Brinch Hansen (1996) for a historical view of monitors and conditional critical sections, the constructs upon which protected objects are based. Sha & Goodenough (1990) is a tutorial on real-time scheduling in the context of Ada, while Gargaro, Smith, Theriault, Volz & Waldorp (1997) describes an implementation of the distributed systems annex.

This book presents the mechanics of object-oriented programming in Ada 95. For a comparison with other languages, see Ben-Ari (1996*b*). A textbook on the analysis and design of systems using object-oriented techniques is Rumbaugh, Blaha, Premerlani, Eddy & Lorensen (1991).

G.2 Electronic

- `http://www.adahome.com`—Ada Home. A comprehensive site with articles, FAQs, reviews and links to vendors of Ada compilers and other software.
- `http://www.acm.org/sigada`—The home page of the ACM Special Interest Group on Ada (SIGAda). The site emphasizes information on Ada research and education.

Bibliography

Andrews, G. R. (1991), *Concurrency Programming: Principles and Practice*, Benjamin/Cummings, Redwood City, CA.

Barnes, J. (1995), *Programming in Ada 95*, Addison-Wesley, Reading, MA.

Barnes, J., ed. (1997), *Ada 95 Rationale: The Language, The Standard Libraries*, LNCS 1247, Springer-Verlag, Berlin.

Ben-Ari, M. (1990), *Principles of Concurrent and Distributed Programming*, Prentice-Hall International, Hemel Hempstead.

Ben-Ari, M. (1996*a*), 'Structure exits, not loops', *SIGCSE Bulletin* **28**(3), 51–55, 59.

Ben-Ari, M. (1996*b*), *Understanding Programming Languages*, John Wiley & Sons, Chichester.

Ben-Ari, M. (1996*c*), 'Using inheritance to implement concurrency', *SIGCSE Bulletin* **28**(1), 180–184.

Ben-Ari, M. (1998*a*), 'How to solve the Santa Claus problem', *Concurrency: Practice & Experience* **10**(6), 485–496.

Ben-Ari, M. (1998*b*), Synchronizing multiple clients and servers, *in* 'Ada-Europe International Conference on Reliable Software Technologies Proceedings', LNCS 1411, Springer-Verlag, Berlin, pp. 40–51.

Brinch Hansen, P. (1996), *The Search for Simplicity: Essays in Parallel Programming*, IEEE Computer Society Press, Los Alamitos, CA.

Burns, A. & Davies, G. (1993), *Concurrent Programming*, Addison-Wesley, Reading, MA.

Burns, A. & Wellings, A. (1995), *Concurrency in Ada*, Cambridge University Press, Cambridge.

Cohen, N. (1996), *Ada as a Second Language (Second Edition)*, McGraw-Hill, New York, NY.

Diller, A. (1993), *LaTeX Line by Line*, John Wiley & Sons, Chichester.

English, J. (1997), *Ada 95: The Craft of Object-Oriented Programming*, Prentice-Hall, Hemel Hempstead.

Feldman, M. B. & Koffman, E. (1996), *Ada 95: Problem Solving and Program Design*, Addison-Wesley, Reading, MA.

Gargaro, A., Smith, G., Theriault, R. J., Volz, R. A. & Waldorp, R. (1997), 'Future directions in Ada—distributed execution and heterogeneous language interoperability toolsets', *Ada Letters* **XVII**(5), 51–56.

Lamport, L. (1986), *LaTeX: A Document Preparation System*, Addison-Wesley, Reading, MA.

Lions, J. L. (1996), Ariane 5: Flight 501 failure, `http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf`.

Manna, Z. & Pnueli, A. (1992), *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, NY.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991), *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ.

Sha, L. & Goodenough, J. B. (1990), 'Real-time scheduling theory and ada', *IEEE Computer* **23**(4), 53–62.

Taft, S. T. (1996), Programming the Internet in Ada 95, *in* 'Reliable Software Technologies - Ada-Europe '96', LNCS 1088, Springer-Verlag, Berlin, pp. 1–16.

Taft, S. T. & Duff, R. A., eds (1997), *Ada 95 Reference Manual: Language and Standard Libraries*, LNCS 1246, Springer-Verlag, Berlin. International Standard ISO/IEC 8652:1995(E).

Index of *ARM* Sections

Subject Index

Index entries are usually listed as nouns; for example, ‘access type’ will be found under ‘type, access’ and ‘case statement’ under ‘statement, case’. Entries from the quiz answers are indexed under the quiz number Qnn, and glossary entries are indicated by Gnnn. Since the glossary *entries* were automatically generated, there may be some duplications or inconsistencies.