Programming in Ada 2012

Programming in Ada 2012

JOHN BARNES

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107424814

First published 2014

Printed in the United Kingdom by Clays, St Ives plc

Typeset By John Barnes Informatics

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data

ISBN 978-1-107-42481-4 Paperback

To Barbara

Contents

Foreword

Programming Languages and Software practice are always engaged in a game of leapfrog: a forward looking programming language introduces new ways of thinking about software development, and its constructs shape the way programmers think about their craft; creative programmers invent new idioms and patterns to tackle ever more complex programming tasks, and these idioms become incorporated in the next generation of programming languages.

The latest version of Ada, whose description we owe once again to the inimitable expository talents of John Barnes, has exemplified this dynamic repeatedly over the last 30 years.

- Ada 83 showed programmers how programming in the large should be organized (packages, strong typing, privacy) and convinced them that indices out of range were not a common pitfall of programming but elementary errors that could be controlled with proper declarations and constraint checking. Ada 83 also put concurrent programming in a mainstream programming language.
- Ada 95 benefited from a decade-long development in object-oriented programming techniques, and successfully grafted the ideas of polymorphism and dynamic dispatching onto a strongly-typed language with concurrency. It enhanced programming-in-the large capabilities with child units and their generic incarnations.
- Ada 2005 showed how data-based synchronization (protected types) and concurrency (task types) could be unified through a novel use of interface inheritance, and adopted a conservative model of multiple inheritance of interfaces that has proved more robust than the more unrestricted models of MI. Ada 2005 also introduced into the language an extensive container library, following here the example of other established languages and many earlier experimental high-level languages that showed the usefulness of reasoning over data aggregates.

And now – Ada 2012, the latest version of the language whose description you are holding, reflects both aspects of this dialectic process: it introduces new ways of thinking about program construction, and it reflects developments in software practice that hark back to the earlier days of our profession but that have seldom, if ever, found their way into well-established programming languages.

The general rubric for these new/old ideas is Programming by Contract. The term became well-known through the pioneering work of Bertrand Meyer and the design of Eiffel, but it probably found more significant use in the SPARK community, in the design of critical software for applications that require a real measure of formal verification for their deployment.

Ada 2012 offers the programmer a wealth of new tools for specifying the intent of a program: preconditions, postconditions, type invariants, subtype predicates. All of these allow the software architect to present more clearly the intent of a piece of software, and they allow the compiler and/or the run-time system to verify that the software behaves as intended. The use of pre- and postconditions was proposed a generation ago by E. DIkjstra and C.A.R .Hoare, but their pioneering efforts were not widely adopted by the software community, among other things because good language support for these mechanisms was lacking. Their introduction in a language whose user community is particularly concerned with mission-critical software reflects the fact that concerns about safety and security are more urgent than ever. We can expect that these techniques will be adopted early and enthusiastically by the aerospace and automotive software development community, as they have been in the small and dedicated SPARK community.

Preconditions, postconditions, type invariants and type predicates are logical assertions about the behavior of a given construct. When these assertions involve data aggregates (vectors, sets, and other container types) it is particularly convenient to use notations from first-order logic, namely quantified expressions.

An important syntactic innovation of Ada 2012 is the introduction of quantified expressions both in their universal form (all elements of this set are French Cheeses) and their existential form (some element of this vector is purple). As a result, the language includes the new keyword **some**. These quantified expressions are of course implicit loops over data aggregates, and in parallel with their introduction, Ada 2012 has extended considerably the syntax of iterators over containers. A generalized notion of indexing now allows the programmer to define their own iterable constructs, as well as mapping between arbitrary types.

Contracts, quantified expressions, and generalized indexing may appear to be miscellaneous additions to an already large language; in fact they are elegantly unified under the umbrella of a new construct: the Aspect Specification, which also generalizes and unifies the earlier notions of attributes and pragmas. The coherence of the language has thus been enhanced as well.

Programming languages must also respond to developments in Computer Hardware. The most significant development of the last decade has been the appearance of multicore architectures, which provide abundant parallelism on a single chip. Making efficient use of the computer power now available on a single processor has been the goal of much development in language design. Ada 2012 provides tools for describing multicore architectures, and for mapping computing activities onto specific cores or sets of them.These are novel capabilities for a general-purpose programming language, and we can expect them to have a profound impact on the practice of parallel programming.

This thumbnail description of the high points of the new version of the language is intended to whet your appetite for the pages that follow. Once again, John Barnes has provided a wonderfully lucid, learned, and insightful description of the latest version of Ada. He has been the tireless explicator of the design and evolution of the language over more than three decades, and the Ada community has acquired its

understanding and love of the language from his prose. A programming language is a tool for thought, and most Ada users have learned how to think about programs from John Barnes's books. I can only hope that the widest possible audience will learn to think straight from the exciting descriptions that follow.

The design of Ada 2012 is once again the result of the collective effort of the Ada Rapporteur group, an extremely talented group of language designers who combine deep industrial experience with an equally deep knowledge of programming language semantics and theoretical computer science. The ARG, of which John Barnes has been an invaluable member from the inception of Ada, has once again created a modern and elegant programming language that addresses the needs of a new generation of software designers. It has been an enormous privilege to work with them. I trust the reader will enjoy the result of their work for years to come. Happy Programming!

Ed Schonberg
AdaCore
Chairman, Ada Rapporteur Group
New York, March 2014

Preface

Welcome to *Programming in Ada 2012* which has been triggered by the recent ISO standardization of Ada 2012.

The original language, devised in the 1980s, is known as Ada 83 and was followed by Ada 95, Ada 2005, and now Ada 2012. Ada has gained a reputation as being the language of choice when software needs to be correct. And as software pervades into more areas of society so that ever more software is safety critical or security critical, it is clear that the future for Ada is bright. One observes, for example, the growth in use of SPARK, the Ada based high integrity language widely used in areas such as avionics and signalling.

Ada 83 was a relatively simple but highly reliable language with emphasis on abstraction and information hiding. It was also notable for being perhaps the first practical language to include multitasking within the language itself.

Ada 95 added extra flexibility to the strongly typed and secure foundation provided by the Software Engineering approach of Ada 83. In particular it added the full dynamic features of Object Oriented Programming (OOP) and in fact was the first such language to become an ISO standard. Ada 95 also made important structural enhancements to visibility control by the addition of child units, it greatly improved multitasking by the addition of protected types, and added important basic material to the standard library.

Ada 2005 then made improvements in two key areas. It added more flexibility in the OOP area by the addition of multiple inheritance via interfaces and it added more facilities in the real-time area concerning scheduling algorithms, timing and other matters of great importance for embedded systems. It also added further facilities to the standard library such as the ability to manipulate containers.

Ada 2012 makes further important enhancements. These include features for contracts such as pre- and postconditions, tasking facilities to recognize multicore architectures, and major additions and improvements to the container library.

In more detail, the changes include

- Contracts – pre- and postconditions, type invariants, and subtype predicates are perhaps the most dramatic new features. The introduction of these features prompted a rethink regarding the specification of various properties of entities in general. As a consequence the use of pragmas has largely been replaced by the elegant new syntax of aspect specifications which enables the properties to be given with the declaration of the entities concerned.

- Expressions – the introduction of the contract material showed a need for more flexible expressions. Accordingly, Ada now includes conditional expressions, case expressions, quantified expressions and more flexible forms of membership tests. A new form of function is also introduced in which the body is essentially given by a single expression.
- Structure and visibility – perhaps the most startling change in this area is allowing functions to have parameters of all modes. This removes the need for a number of obscure techniques (dirty tricks really) which had been used. Other important improvements concern incomplete types.
- Tasking – Ada 2012 has new features describing the allocation of tasks to individual processors and sets of processors; these additions were prompted by the rapid growth in the use of multicore architectures.
- Generally – new flexible forms of iterators and dereferencing are introduced mainly for use with containers. Better control of storage pools is permitted by the introduction of subpools.
- Predefined library – some improvements are made concerning directories and a feature is added for the identification of locale. However, the most important improvement is the addition of many new forms of containers. These include multiway trees and task-safe queues. There are also bounded forms of all containers which are important for high integrity systems where dynamic storage management is often not permitted.

This book follows the tradition of its predecessors. It presents an overall description of Ada 2012 as a language. Some knowledge of the principles of programming is assumed but an acquaintance with specific other languages is by no means necessary.

The book comprises 27 chapters grouped into four parts as follows

- Chapters 1 to 4 provide an overview which should give the reader an understanding of the overall scope of the language as well as the ability to run significant programs as examples – this is particularly for newcomers to Ada.
- Chapters 5 to 11 cover the small-scale aspects such as the lexical details, scalar, array and simple record types, control and expression structures, subprograms and access types.
- Chapters 12 to 22 discuss the large-scale aspects including packages and private types, contracts, separate compilation, abstraction, OOP and tasking as well as exceptions and the details of numerics.
- Chapters 23 to 27 complete the story by discussing the predefined library, interfacing to the outside world and the specialized annexes; there is then a finale concluding with some ruminations over correctness and a brief introduction to SPARK.

The finale includes, as in its predecessors, with the fantasy customer in the shop trying to buy reusable software components and whose dream now seems as far away or indeed as near at hand as it did many years ago when I first toiled at this book. The discussion continues to take a galactic view of life and perhaps echoes the cool cover of the book which depicts the Ice Comet.

Those familiar with *Programming in Ada 2005* might find the following summary of key changes helpful

- The number of chapters has grown from 25 to 27. The new chapter 8 covers the new forms of expressions such as if expressions, case expressions, and quantified expressions. The new chapter 16 discusses the material on contracts. The chapter on containers (now chapter 24) has grown because of the introduction of containers for multiway trees, single indefinite objects, and various forms of queues. A number of existing chapters have additional sections such as that on aliasing.
- The revisions to produce Ada 2012 have impacted to a greater or lesser extent on many aspects of the language. Most chapters conclude with a checklist summarizing important points to remember and listing the main additions in Ada 2012.
- As a consequence the book is now some 120 pages longer. It would have been even longer had I not decided that it was unnecessary to include the answer to every exercise. Accordingly, the printed answers cover just the introductory chapters (for the benefit of those entirely new to Ada) and those exercises that are referred to elsewhere in the book. But all the answers are on the associated website.

The website also includes the six sample programs both in text form and as executable programs, some material from earlier versions of this book which now seem of lesser importance but which I nevertheless was reluctant to lose completely. More details of the website will be found below.

And now I must thank all those who have helped with this new book. The reviewers included Janet Barnes, Alan Burns, Rod Chapman, Jeff Cousins, Bob Duff, Stuart Matthews, Ed Schonberg, Tucker Taft, and Andy Wellings. Their much valued comments enabled me to improve the presentation and to eliminate a number of errors. Some of the new material is based on parts of the *Ada 2012 Rationale* and I must express my special gratitude to Randy Brukardt for his painstaking help in reviewing that document.

Finally, many thanks to my wife Barbara for help in typesetting and proof-reading and to friends at Cambridge University Press for their continued guidance and help.

John Barnes
Caversham, England
March 2014

Notes on the website

The website is www.cambridge.org/barnes. It contains three main things: the full answers to all the exercises, some obscure or obsolete material on exceptions, discriminants, and iterators which were in previous versions of the book, and additional material on the six sample programs.

I do hope that readers will find the sample programs on the website of interest. I am aware that they are a bit intricate. But this seems almost inevitable in order to

illustrate a broad range of features of Ada in a reasonably concise manner. However, in most cases they build on examples in preceding chapters and so should not be difficult to follow.

Each example commences with some remarks about its purpose and overall structure. This is followed by the text of the program and then some notes on specific details. A desire to keep the program text short means that comments are at a minimum. However, the corresponding source text on the website includes much additional commentary. The website also includes further discussion and explanation and suggestions for enhancement. In general the programs use only those features of the language explained in detail by that point in the book.

The first program, Magic Moments, illustrates type extension and dispatching. It shows how the existence of common components and common operations enable dispatching to compute various geometrical properties almost by magic.

The Sylvan Sorter is an exercise in access types and basic algorithmic techniques including recursion.

The Rational Reckoner provides two examples of abstract data types – the rational numbers themselves and the stack which is the basis of the calculator part of the program.

The Super Sieve illustrates multitasking and communication between tasks both directly through entry calls and indirectly through protected objects. For added interest it is made generic so that more general primes than the familiar integers may be found. This provides the opportunity to use a discriminated record type and a modular type to represent binary polynomials.

The program Wild Words is probably the hardest to follow because it is not based on any particular example described in the preceding chapters. It illustrates many of the facilities of the character and string handling packages as well as the generation of random numbers.

The final program, Playing Pools, shows how users might write their own storage allocation package for the control of storage pools. The example shown enables the user to monitor the state of the pool and it is exercised by running the familiar Tower of Hanoi program which moves a tower of discs between three poles. Variety is provided by implementing the stack structures representing the three poles (and defined by an interface) in two different ways and dispatching to the particular implementation. The website includes an extended version which uses three different ways.

Information on many aspects of Ada such as vendors, standards, books and so on can be obtained from the websites listed in the Bibliography.

Part 1

An Overview

This first part covers the background to the Ada language and an overview of most of its features. Enough material is presented here to enable a programmer to write complete simple programs.

Chapter 1 contains a short historical account of the origins and development of various versions of Ada from Ada 83 through Ada 95 and Ada 2005 and leading to Ada 2012 which is the topic of this book. There is also a general discussion of how the evolution of abstraction has been an important key to the development of programming languages in general.

Broadly speaking, the other three chapters in this part provide an overview of the material in the corresponding other parts of the book. Thus Chapter 2 describes the simple concepts familiar from languages such as C and Pascal and which form the subject of the seven chapters which comprise Part 2. Similarly Chapter 3 covers the important topic of abstraction which is the main theme of Part 3. And then Chapter 4 rounds off the overview by showing how a complete program is put together and corresponds to Part 4 which covers material such as the predefined library.

Chapter 3 includes a discussion of the popular topic of object oriented programming and illustrates the key concepts of classes as groups of related types, of type extension and inheritance as well as static polymorphism (genericity) and dynamic polymorphism leading to dynamic binding. It also includes a brief comparison between the terminology used by Ada and that used by some other languages. This chapter concludes with an introduction to tasking which is a very important aspect of Ada and is a topic not addressed by most programming languages at all.

This part concludes with the first of a number of complete programs designed to give the reader a better understanding of the way the various components of the language fit together. This particular program illustrates a number of aspects of type extension and polymorphism.

Those not familiar with Ada will find that this part will give them a fair idea of Ada's capabilities and lays the foundation for understanding the details presented in the remainder of the book.

1 Introduction

Ada 2012 is a comprehensive high level programming language especially suited for the professional development of large or critical programs for which correctness and robustness are major considerations. In this introductory chapter we briefly trace the development of Ada 2012 (and its predecessors Ada 83, Ada 95, and Ada 2005), its place in the overall language scene and the general structure of the remainder of this book.

1.1 Standard development

Ada 2012 is a direct descendant of Ada 83 which was originally sponsored by the US Department of Defense for use in the so-called embedded system application area. (An embedded system is one in which the computer is an integral part of a larger system such as a chemical plant, missile or dishwasher.)

The story of Ada goes back to about 1974 when the United States Department of Defense realized that it was spending far too much on software, especially in the embedded systems area. To cut a long story short the DoD sponsored the new language through a number of phases of definition of requirements, competitive and parallel development and evaluation which culminated in the issue of the ANSI standard for Ada in 1983[1]. The team that developed Ada was based at CII Honeywell Bull in France under the leadership of Jean D Ichbiah.

The language was named after Augusta Ada Byron, Countess of Lovelace (1815–52). Ada, the daughter of the poet Lord Byron, was the assistant and patron of Charles Babbage and worked on his mechanical analytical engine. In a very real sense she was therefore the world's first programmer.

Ada 83 became ISO standard 8652 in 1987 and, following normal ISO practice, work leading to a revised standard commenced in 1988. The DoD, as the agent of ANSI, the original proposers of the standard to ISO, established the Ada project in

1988 under the management of Christine M Anderson. The revised language design was contracted to Intermetrics Inc. under the technical leadership of S Tucker Taft. The revised ISO standard was published on 15 February 1995 and so became Ada 95[2].

The maintenance of the language is performed by the Ada Rapporteur Group (ARG) of ISO/IEC committee SC22/WG9. The ARG under the leadership of Erhard Plödereder identified the need for some corrections and these were published as a Corrigendum on 1 June 2001[3].

Experience with Ada 95 and other modern languages such as Java indicated that further improvements would be very useful. The changes needed were not so large as the step from Ada 83 to Ada 95 and so an Amendment was deemed appropriate rather than a Revised standard. The ARG then developed the Amended language known as Ada 2005 under the leadership of Pascal Leroy[4].

Further experience with Ada 2005 showed that additions especially in the area of contracts and multiprocessor support would be appropriate. Moreover, it was decided that this time a consolidated Edition should be produced. This was accordingly done under the leadership of Ed Schonberg with the editor being Randy Brukardt and resulted in the version known as Ada 2012 which is the subject of this book. Ada 2012 became an ISO standard towards the end of 2012[5].

1.2 Software engineering

It should not be thought that Ada is just another programming language. Ada is about Software Engineering, and by analogy with other branches of engineering it can be seen that there are two main problems with the development of software: the need to reuse software components as much as possible and the need to establish disciplined ways of working.

As a language, Ada (and hereafter by Ada we generally mean Ada 2012) largely solves the problem of writing reusable software components – or at least through its excellent ability to prescribe interfaces, it provides an enabling technology in which reusable software can be written.

The establishment of a disciplined way of working seems to be a Holy Grail which continues to be sought. One of the problems is that development environments change so rapidly that the stability necessary to establish discipline can be elusive.

But Ada is stable and encourages a style of programming which is conducive to disciplined thought. Experience with Ada shows that a well designed language can reduce the cost of both the initial development of software and its later maintenance.

The main reason for this is simply reliability. The strong typing and related features ensure that programs contain few surprises; most errors are detected at compile time and of those that remain many are detected by run-time constraints. Moreover, the compile-time checking extends across compilation unit boundaries.

This aspect of Ada considerably reduces the costs and risks of program development compared for example with C and its derivatives such as C++. Moreover an Ada compilation system includes the facilities found in separate tools such as ‘lint’ and ‘make’ for C. Even if Ada is seen as just another programming

language, it reaches parts of the software development process that other languages do not reach.

Ada 95 added extra flexibility to the inherent reliability of Ada 83. That is, it kept the Software Engineering but allowed more flexibility. The features of Ada 95 which contributed to this more flexible feel are the extended or tagged types, the hierarchical library facility and the greater ability to manipulate pointers or references. Another innovation in Ada 95 was the introduction of protected types to the tasking model.

As a consequence, Ada 95 incorporated the benefits of object oriented languages without incurring the pervasive overheads of languages such as Smalltalk or the insecurity brought by the weak C foundation in the case of C++. Ada 95 remained a very strongly typed language but provided the benefits of the object oriented paradigm.

Ada 2005 added yet further improvements to the object model by adding interfaces in the style of Java and providing constructor functions and also extending the object model to incorporate tasking. Experience has shown that a standard library is important and accordingly Ada 2005 had a much larger library including predefined facilities for containers.

Ada has always been renowned as the flagship language for multitasking applications. (Multitasking is often known as multithreading.) This position was strengthened by the addition of further standard paradigms for scheduling and timing and the incorporation of the Ravenscar profile into Ada 2005. The Ravenscar profile enables the development of real-time programs with predictable behaviour.

Further improvements which have resulted in Ada 2012 are in three main areas. First there is the introduction of material for 'programming by contract' such as pre- and postconditions somewhat on the lines of those found in Eiffel. There are also additions to the tasking model including facilities for mapping tasks onto multiprocessors. Other important extensions are additional facilities in the container library enabling further structures (such as trees and queues) to be addressed; other improvements also simplify many operations on the existing container structures.

Two kinds of application stand out where Ada is particularly relevant. The very large and the very critical.

Very large applications, which inevitably have a long lifetime, require the cooperative effort of large teams. The information hiding properties of Ada and especially the way in which integrity is maintained across compilation unit boundaries are invaluable in enabling such developments to progress smoothly. Furthermore, if and when the requirements change and the program has to be modified, the structure and especially the readability of Ada enable rapid understanding of the original program even if it is modified by a different team.

Very critical applications are those that just have to be correct otherwise people or the environment get damaged. Obvious examples occur in avionics, railway signalling, process control and medical applications. Such programs may not be large but have to be very well understood and often mathematically proven to be correct. The full flexibility of Ada is not appropriate in this case but the intrinsic reliability of the strongly typed kernel of the language is exactly what is required. Indeed many certification agencies dictate the properties of acceptable languages and whereas they do not always explicitly demand a subset of Ada, nevertheless the properties are not provided by any other practically available language. The SPARK language which is based around a kernel subset of Ada illustrates how special tools

can provide extra support for developing high integrity systems. A SPARK program includes additional information regarding data flow, state and proof. In the original versions of SPARK this was done in the form of Ada comments. However, in SPARK 2014 this information is presented using new features of Ada including pre- and postconditions and additional assertions. This information is processed by the SPARK tools and can be used to show that a program meets certain criteria such as not raising any predefined exceptions. Much progress has been made in the area of proof in recent years. A brief introduction to SPARK will be found in the very last section of this book.

1.3 Evolution and abstraction

The evolution of programming languages has apparently occurred in a rather *ad hoc* fashion but with hindsight it is now possible to see a number of major advances. Each advance seems to be associated with the introduction of a level of abstraction which removes unnecessary and harmful detail from the program.

The first advance occurred in the early 1950s with high level languages such as Fortran and Autocode which introduced 'expression abstraction'. It thus became possible to write statements such as

```
X = A + B(I)
```

so that the use of the machine registers to evaluate the expression was completely hidden from the programmer. In these early languages the expression abstraction was not perfect since there were somewhat arbitrary constraints on the complexity of expressions; subscripts had to take a particularly simple form for instance. Later languages such as Algol 60 removed such constraints and completed the abstraction.

The second advance concerned 'control abstraction'. The prime example was Algol 60 which took a remarkable step forward; no language since then has made such an impact on later developments. The point about control abstraction is that the flow of control is structured and individual control points do not have to be named or numbered. Thus we write

```
if X = Y then P := Q else A := B
```

and the compiler generates the gotos and labels which would have to be explicitly used in early versions of languages such as Fortran. The imperfection of early expression abstraction was repeated with control abstraction. In this case the obvious flaw was the horrid Algol 60 switch which has now been replaced by the case statement of later languages.

The third advance was 'data abstraction'. This means separating the details of the representation of data from the abstract operations defined upon the data.

Older languages take a very simple view of data types. In all cases the data is directly described in numerical terms. Thus if the data to be manipulated is not really numerical (it could be traffic light colours) then some mapping of the abstract type must be made by the programmer into a numerical type (usually integer). This mapping is purely in the mind of the programmer and does not appear in the written program except perhaps as a comment.

Pascal introduced a certain amount of data abstraction as instanced by the enumeration type. Enumeration types allow us to talk about the traffic light colours in their own terms without our having to know how they are represented in the computer. Moreover, they prevent us from making an important class of programming errors – accidentally mixing traffic lights with other abstract types such as the names of fish. When all such types are described in the program as numerical types, such errors can occur.

Another form of data abstraction concerns visibility. It has long been recognized that the traditional block structure of Algol and Pascal is not adequate. For example, it is not possible in Pascal to write two procedures to operate on some common data and make the procedures accessible without also making the data directly accessible. Many languages have provided control of visibility through separate compilation; this technique is adequate for medium-sized systems, but since the separate compilation facility usually depends upon some external system, total control of visibility is not gained. The module of Modula is an example of an appropriate construction.

Ada was probably the first practical language to bring together these various forms of data abstraction.

Another language which made an important contribution to the development of data abstraction is Simula 67 with its concept of class. This leads us into the paradigm now known as object oriented programming. There seems to be no precise definition of OOP, but its essence is a flexible form of data abstraction providing the ability to define new data abstractions in terms of old ones and allowing dynamic selection of types.

All types in Ada 83 were static and thus Ada 83 was not classed as a truly object oriented language but as an object based language. However, Ada 95, Ada 2005, and Ada 2012 include the essential functionality associated with OOP such as type extension and dynamic polymorphism.

We are, as ever, probably too close to the current scene to achieve a proper perspective. Data abstraction in Ada 83 seems to have been not quite perfect, just as Fortran expression abstraction and Algol 60 control abstraction were imperfect in their day. It remains to be seen just how well Ada now provides what we might call ‘object abstraction’. Indeed it might well be that inheritance and other aspects of OOP turn out to be unsatisfactory by obscuring the details of types although not hiding them completely; this could be argued to be an abstraction leak making the problems of program maintenance even harder.

A brief survey of how Ada relates to other languages would not be complete without mention of C and C++. These have a completely different evolutionary trail to the classic Algol–Pascal–Ada route.

The origin of C can be traced back to the CPL language devised by Strachey, Barron and others in the early 1960s. This was intended to be used on major new hardware at Cambridge and London universities but proved hard to implement. From it emerged the simple system programming language BCPL and from that B and then C. The essence of BCPL was the array and pointer model which abandoned any hope of strong typing and (with hindsight) a proper mathematical model of the mapping of the program onto a computing engine. Even the use of := for assignment was lost in this evolution which reverted to the confusing use of = as in Fortran. Having hijacked = for assignment, C uses == for equality thereby conflicting with several hundred years of mathematical usage. About the only feature of the elegant

CPL remaining in C is the unfortunate braces {} and the associated compound statement structure which was abandoned by many other languages in favour of the more reliable bracketed form originally proposed by Algol 68. It is again tragic to observe that Java has used the familiar but awful C style. The very practical problems with the C notation are briefly discussed in Chapter 2.

Of course there is a need for a low level systems language with functionality like C. It is, however, unfortunate that the interesting structural ideas in C++ have been grafted onto the fragile C foundation. As a consequence although C++ has many important capabilities for data abstraction, including inheritance and polymorphism, it is all too easy to break these abstractions and create programs that violently misbehave or are exceedingly hard to understand and maintain. Java is free from most of these flaws but persists with anarchic syntax.

The designers of Ada 95 incorporated the positive dynamic facilities of the kind found in C++ onto the firm foundation provided by Ada 83. The designers of Ada 2005 and Ada 2012 have added further appropriate good ideas from Java. But the most important step taken by Ada 2012 is to include facilities for 'programming by contract' which in a sense is the ultimate form of abstraction.

Ada thus continues to advance along the evolution of abstraction. It incorporates full object abstraction in a way that is highly reliable without incurring excessive run-time costs.

1.4 Structure and objectives of this book

Learning a programming language is a bit like learning to drive a car. Certain key things have to be learnt before any real progress is possible. Although we need not know how to use the cruise control, nevertheless we must at least be able to start the engine, select gears, steer and brake. So it is with programming languages. We do not need to know all about Ada before we can write useful programs but quite a lot must be learnt. Moreover, many virtues of Ada become apparent only when writing large programs just as many virtues of a Rolls-Royce are not apparent if we only use it to drive to the local store.

This book is not an introduction to programming but an overall description of programming in Ada. It is assumed that the reader will have significant experience of programming in some other language such as Pascal, C or Java (or earlier versions of Ada). But a specific knowledge of any particular language is not assumed.

It should also be noted that this book strives to remain neutral regarding methods of program design and should therefore prove useful whatever techniques are used.

This book is primarily about programming in Ada 2012, but in order to aid transition from earlier versions of Ada, most chapters contain a summary of where Ada 2012 differs from Ada 95 and Ada 2005 in the area concerned.

This book is in four main parts. The first part, Chapters 1 to 4, is an extensive overview of most of the language and covers enough material to enable a wide variety of programs to be written; it also lays the foundation for understanding the rest of the material.

The second part, Chapters 5 to 11, covers the traditional algorithmic parts of the language and roughly corresponds to the domain addressed by Pascal or C although

the detail is much richer. The third part, Chapters 12 to 22, covers modern and exciting material associated with data abstraction, programming in the large, OOP, contracts, and parallel processing.

Finally, the fourth part, Chapters 23 to 27, completes the story by discussing the predefined environment, interfacing to the outside world and the specialized annexes; the concluding chapter also pulls together a number of threads that are slightly dispersed in the earlier chapters and finishes with an introduction to SPARK.

There are also six complete program examples which are interspersed at various points. The first follows Chapter 4 and illustrates various aspects of OOP. Others follow Chapters 11, 13, 22, 23 and 25 and illustrate access types, data abstraction, generics and tasking, string handling, and storage pools respectively. These examples illustrate how the various components provided by Ada can be fitted together to make a complete program. The full text of these programs including additional comments and other explanatory material will be found on the associated website.

Most sections contain exercises. It is important that the reader does most, if not all, of these since they are an integral part of the discussion and later sections often use the results of earlier exercises. Solutions to key exercises will be found at the end of the book and solutions to them all will be found on the website.

Most chapters conclude with a short checklist of key points to be remembered. Although incomplete, these checklists should help to consolidate understanding.

Various appendices are provided in order to make this book reasonably self-contained. They cover matters such as reserved words, attributes, aspects, pragmas, and restrictions. There is also a glossary of terms and the complete syntax organized to correspond to the order in which the topics are introduced. The book concludes with a Bibliography and Index

This book includes occasional references to Ada Issues. These are the reports of the Ada Rapporteur Group (ARG) which analyses technical queries and drafts the new Ada standards from time to time. Since Ada 2012 became an ISO standard, a small number of improvements to the language have been made. For example, AI12-33 concerns the addition of new facilities for grouping processors. The relevant Issues are listed in the Index.

This book covers all aspects of Ada but does not explore every pathological situation. Its purpose is to teach the reader the effect of and intended use of the features of Ada. In a few areas the discussion is incomplete; these are areas such as system dependent programming, input–output, and the specialized annexes. System dependent programming (as its name implies) is so dependent upon the particular implementation that only a brief overview seems appropriate. Input–output, although important, does not introduce new concepts but is rather a mass of detail; again a simple overview is presented. And, as their name implies, the specialized annexes address the very specific needs of certain communities; to cover them in detail would make this book excessively long and so only an overview is provided.

Further details of these areas can be found in the *Ada Reference Manual* (the *ARM*) which is referred to from time to time. The appendices are mostly based upon material drawn from the *ARM*. A related document is the *Ada 2012 Rationale* which aims to explain the reasons for the upgrade from Ada 2005 to Ada 2012. Many of the examples in this book are derived from those in the *Rationale*. There is also an extended form of the reference manual known as the *Annotated Ada Reference Manual* (the *AARM*) – this includes much embedded commentary and is aimed at

the language lawyer and compiler writer. Readers transitioning from Ada 95 will also find the *Ada 2005 Rationale* of interest. These documents will all be found on the Ada website as described in the Bibliography. Traditional paper editions of most of these documents are also available as well.

1.5 References

The references given here are to the formal standardization documents describing the various versions of the Ada Programming Language. References to other documents will be found in the Bibliography.

1 United States Department of Defense. *Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A). Washington DC, 1983.

2 International Organization for Standardization. *Information technology – Programming languages – Ada. Ada Reference Manual.* ISO/IEC 8652:1995(E).

3 International Organization for Standardization. *Information technology – Programming languages – Ada. Technical Corrigendum 1.* ISO/IEC 8652:1995/COR.1:2001.

4 International Organization for Standardization. *Information technology – Programming languages – Ada. Amendment 1.* ISO/IEC 8652:1995/AMD.1:2006.

5 International Organization for Standardization. *Information technology – Programming languages – Ada. Ada Reference Manual.* ISO/IEC 8652:2012(E).

2 Simple Concepts

This is the first of three chapters covering in outline the main goals, concepts and features of Ada. Enough material is given in these chapters to enable the reader to write significant programs. It also allows the reader to create a framework in which the exercises and other fragments of program can be executed, if desired, before all the required topics are discussed in depth.

The material covered in this chapter corresponds approximately to that in simple languages such as Pascal and C.

2.1 Key goals

Ada is a large language since it addresses many important issues relevant to the programming of practical systems in the real world. It is much larger than Pascal which is really only suitable for training purposes and for small personal programs. Ada is similarly much larger than C although perhaps of the same order of size as C++. But a big difference is the stress which Ada places on integrity and readability. Some of the key issues in Ada are

- Readability – professional programs are read much more often than they are written. So it is important to avoid a cryptic notation such as in APL which, although allowing a program to be written down quickly, makes it almost impossible to be read except perhaps by the original author soon after it was written.
- Strong typing – this ensures that each object has a clearly defined set of values and prevents confusion between logically distinct concepts. As a consequence many errors are detected by the compiler which in other languages (such as C) can lead to an executable but incorrect program.

- Programming in the large – mechanisms for encapsulation, separate compilation and library management are necessary for the writing of portable and maintainable programs of any size.
- Exception handling – it is a fact of life that programs of consequence are rarely perfect. It is necessary to provide a means whereby a program can be constructed in a layered and partitioned way so that the consequences of unusual events in one part can be contained.
- Data abstraction – extra portability and maintainability can be obtained if the details of the representation of data are kept separate from the specifications of the logical operations on the data.
- Object oriented programming – in order to promote the reuse of tested code, the type flexibility associated with OOP is important. Type extension (inheritance), polymorphism and late binding are all desirable especially when achieved without loss of type integrity.
- Tasking – for many applications it is important to conceive a program as a series of parallel activities rather than just as a single sequence of actions. Building appropriate facilities into a language rather than adding them via calls to an operating system gives better portability and reliability.
- Generic units – in many cases the logic of part of a program is independent of the types of the values being manipulated. A mechanism is therefore necessary for the creation of related pieces of program from a single template. This is particularly useful for the creation of libraries.
- Communication – programs do not live in isolation and it is important to be able to communicate with systems possibly written in other languages.

An overall theme in the design of Ada was concern for the programming process as a human activity. An important aspect of this is enabling errors to be detected early in the overall process. For example a single typographical error in Ada usually results in a program that does not compile rather than a program that still compiles but does the wrong thing. We shall see some examples of this in Section 2.6.

2.2 Overall structure

An important objective of software engineering is to reuse existing pieces of program so that detailed new coding is kept to a minimum. The concept of a library of program components naturally emerges and an important aspect of a programming language is therefore its ability to access the items in a library.

Ada recognizes this situation and introduces the concept of library units. A complete Ada program is conceived as a main subprogram (itself a library unit) which calls upon the services of other library units. These library units can be thought of as forming the outermost lexical layer of the total program.

The main subprogram takes the form of a procedure of an appropriate name. The service library units can be subprograms (procedures or functions) but they are more likely to be packages. A package is a group of related items such as subprograms but may contain other entities as well.

Suppose we wish to write a program to print out the square root of some number. We can expect various library units to be available to provide us with a means of computing square roots and performing input and output. Our job is merely to write a main subprogram to use these services as we wish.

We will suppose that the square root can be obtained by calling a function in our library whose name is Sqrt. We will also suppose that our library includes a package called Simple_IO containing various simple input–output facilities. These facilities might include procedures for reading numbers, printing numbers, printing strings of characters and so on.

Our program might look like

```
with Sqrt, Simple_IO;
procedure Print_Root is
   use Simple_IO;                          -- declarations here
   X: Float;
begin
   Get(X);                                 -- statements here
   Put(Sqrt(X));
end Print_Root;
```

The program is written as a procedure called Print_Root preceded by a with clause giving the names of the library units which it wishes to use. Between **is** and **begin** we can write declarations, and between **begin** and **end** we write statements. Broadly speaking, declarations introduce the entities we wish to manipulate and statements indicate the sequential actions to be performed.

We have introduced a variable X of type Float which is a predefined language type. Values of this type are a set of floating point numbers and the declaration of X indicates that X can have values only from this set. A value is assigned to X by calling the procedure Get which is in our package Simple_IO.

Writing

```
use Simple_IO;
```

gives us immediate access to the facilities in the package Simple_IO. If we had omitted this use clause we would have had to write

```
Simple_IO.Get(X);
```

in order to indicate where Get was to be found.

The program then calls the procedure Put in the package Simple_IO with a parameter which in turn is the result of calling the function Sqrt with the parameter X.

Some small-scale details should be noted. The various statements and declarations all terminate with a semicolon; this is like C but unlike Pascal where semicolons are separators rather than terminators. The program contains various words such as **procedure**, Put and X. These fall into two categories. A few (73 in fact) such as **procedure** and **is** are used to indicate the structure of the program; they are reserved words and can be used for no other purpose. All others, such as Put and X, can be used as identifiers for whatever purpose we desire. Some of these, such as Float in the example, have a predefined meaning but we can nevertheless reuse them although it might be confusing to do so. For clarity in this book we write

the reserved words in lower case bold and use leading capitals for all others. This is purely a matter of style; the language rules do not distinguish the two cases except when we consider the manipulation of characters themselves. Note also how the underline character is used to break up long identifiers to increase readability.

Finally, observe that the name of the procedure, Print_Root, is repeated between the final **end** and the terminating semicolon. This is optional but is recommended so as to clarify the overall structure although this is obvious in a small example such as this.

Our program is very simple; it might be more useful to enable it to cater for a whole series of numbers and print out each answer on a separate line. We could stop the program somewhat arbitrarily by giving it a value of zero.

```
with Sqrt, Simple_IO;
procedure Print_Roots is
   use Simple_IO;
   X: Float;
begin
   Put("Roots of various numbers");
   New_Line(2);
   loop
      Get(X);
      exit when X = 0.0;
      Put(" Root of "); Put(X); Put(" is ");
      if X < 0.0 then
         Put("not calculable");
      else
         Put(Sqrt(X));
      end if;
      New_Line;
   end loop;
   New_Line;
   Put("Program finished");
   New_Line;
end Print_Roots;
```

The output has been enhanced by calling further procedures New_Line and Put in the package Simple_IO. A call of New_Line outputs the number of new lines specified by the parameter (of the predefined type Integer); the procedure New_Line has been written in such a way that if no parameter is supplied then a default value of 1 is assumed. There are also calls of Put with a string as argument. This is a different procedure from the one that prints the number X. The compiler distinguishes them by the different types of parameters. Having more than one subprogram with the same name is known as overloading. Note also the form of strings; this is a situation where the case of the letters does matter.

Various new control structures are also introduced. The statements between **loop** and **end loop** are repeated until the condition X = 0.0 in the **exit** statement is found to be true; when this is so the loop is finished and we immediately carry on after **end loop**. We also check that X is not negative; if it is we output the message ‘not calculable’ rather than attempting to call Sqrt. This is done by the if statement;

if the condition between **if** and **then** is true, then the statements between **then** and **else** are executed, otherwise those between **else** and **end if** are executed.

The general bracketing structure should be observed; **loop** is matched by **end loop** and **if** by **end if**. All the control structures of Ada have this closed form rather than the open form of Pascal and C which can lead to poorly structured and incorrect programs.

We will now consider in outline the possible general form of the function Sqrt and the package Simple_IO that we have been using.

The function Sqrt will have a structure similar to that of our main subprogram; the major difference will be the existence of parameters.

```
function Sqrt(F: Float) return Float is
   R: Float;
begin
   ...      -- compute value of Sqrt(F) in R
   return R;
end Sqrt;
```

We see here the description of the formal parameters (in this case only one) and the type of the result. The details of the calculation are represented by the comment which starts with a double hyphen. The return statement is the means by which the result of the function is indicated. Note the distinction between a function which returns a result and is called as part of an expression, and a procedure which does not have a result and is called as a single statement.

The package Simple_IO will be in two parts: the specification which describes its interface to the outside world, and the body which contains the details of how it is implemented. If it just contained the procedures that we have used, its specification might be

```
package Simple_IO is
   procedure Get(F: out Float);
   procedure Put(F: in Float);
   procedure Put(S: in String);
   procedure New_Line(N: in Integer := 1);
end Simple_IO;
```

The parameter of Get is an **out** parameter because the effect of a call such as Get(X); is to transmit a value out from the procedure to the actual parameter X. The other parameters are all **in** parameters because the value goes in to the procedures. The parameter mode can be omitted and is then taken as **in** by default. There is a third mode **in out** which enables the value to go both ways. Both functions and procedures can have parameters of all modes. We usually give the mode **in** for procedures explicitly but omit it for functions.

Only a part of the procedures occurs in the package specification; this part is known as the procedure specification and just gives enough information to enable the procedures to be called.

We see also the two overloaded specifications of Put, one with a parameter of type Float and the other with a parameter of type String. Finally, note how the default value of 1 for the parameter of New_Line is indicated.

The package body for Simple_IO will contain the full procedure bodies plus any other supporting material needed for their implementation and is naturally hidden from the outside user. In vague outline it might look like

```
with Ada.Text_IO;
package body Simple_IO is
   procedure Get(F: out Float) is
      ...
   end Get;
   ...      -- other procedures similarly
end Simple_IO;
```

The with clause shows that the implementation of the procedures in Simple_IO uses the more general package Ada.Text_IO. The notation indicates that Text_IO is a *child* package of the package Ada. It should be noticed how the full body of Get repeats the procedure specification which was given in the corresponding package specification. (The procedure specification is the bit up to but not including **is**.) Note that the package Ada.Text_IO really exists whereas Simple_IO is a figment of our imagination made up for the purpose of this example. We will say more about Ada.Text_IO in Chapter 4.

The example in this section has briefly revealed some of the overall structure and control statements of Ada. One purpose of this section has been to stress that the idea of packages is one of the most important concepts in Ada. A program should be conceived as a number of components which provide services to and receive services from each other.

Finally, note that there is a special package Standard which exists in every implementation and contains the declarations of all the predefined identifiers such as Float and Integer. Access to Standard is automatic and it does not have to be mentioned in a with clause. It is discussed in detail in Chapter 23.

Exercise 2.2

1 Suppose that the function Sqrt is not on its own but in a package Simple_Maths along with other mathematical functions Log, Ln, Exp, Sin and Cos. By analogy with the specification of Simple_IO, write the specification of such a package. How would the program Print_Roots need to be changed?

2.3 The scalar type model

We have said that one of the key benefits of Ada is its strong typing. This is well illustrated by the enumeration type. Consider

```
declare
   type Colour is (Red, Amber, Green);
   type Fish is (Cod, Hake, Salmon);
   X, Y: Colour;
   A, B: Fish;
begin
```

```
    X := Red;              -- ok
    A := Hake;             -- ok
    B := X;                -- illegal
    ...
end;
```

This fragment of program is a block which declares two enumeration types Colour and Fish, and two variables of each type, and then performs various assignments. The declarations of the types give the allowed values of the types. Thus the variable X can only take one of the three values Red, Amber or Green. The fundamental rule of strong typing is that we cannot assign a value of one type to a variable of a different type. So we cannot mix up colours and fishes and thus our (presumably accidental) attempt to assign the value of X to B is illegal and will be detected during compilation.

Four enumeration types are predefined in the package Standard. One is

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

which plays a key role in control flow. Thus the predefined relational operators such as < produce a result of this type and such a value follows **if** as in

```
if X < 0.0 then
```

The other predefined enumeration types are Character, Wide_Character and Wide_Wide_Character whose values are the 8-bit ISO Latin-1, the 16-bit ISO Basic Multilingual Plane and the full 32-bit ISO 10646 characters; these types naturally play an important role in input–output. Their literal values include the printable characters and these are represented by placing them in single quotes thus 'X' or 'a' or indeed '"'.

The other fundamental types are the numeric types. One way or another, all other data types are built out of enumeration types and numeric types. The two major classes of numeric types are the integer types and floating point types (there are also fixed point types which are rather obscure and deserve no further mention in this overview). The integer types are further subdivided into signed integer types (such as Integer) and unsigned or modular types. All implementations have the types Integer and Float. An implementation may also have other predefined numeric types, Long_Integer, Long_Float, Short_Float and so on. There will also be specific integer types for an implementation depending upon the supported word lengths such as Integer_16 and unsigned types such as Unsigned_16.

One of the problems of numeric types is how to obtain both portability and efficiency in the face of variation in machine architecture. In order to explain how this is done in Ada it is convenient to introduce the concept of a derived type. (We will deal with derived types in more detail in the next chapter when we come to object oriented programming.)

The simplest form of derived type introduces a new type which is almost identical to an existing type except that it is logically distinct. If we write

```
type Light is new Colour;
```

then Light will, like Colour, be an enumeration type with literals Red, Amber and Green. However, values of the two types cannot be arbitrarily mixed since they are

logically distinct. Nevertheless, in recognition of the close relationship, a value of one type can be converted to the other by explicitly using the destination type name. So we can write

```
declare
   type Light is new Colour;
   C: Colour;
   L: Light;
begin
   L := Amber;              -- the light amber, not the colour
   C := Colour(L);          -- explicit conversion
   ...
end;
```

whereas a direct assignment

```
   C := L;                  -- illegal
```

would violate the strong typing rule and this violation would be detected during compilation.

Returning now to the numeric types, if we write

```
type My_Float is new Float;
```

then My_Float will have all the operations (+, – etc.) of Float and in general can be considered as equivalent. Now suppose we transfer the program to a different computer on which the predefined type Float is not so accurate and that Long_Float is necessary. Assuming that the program has been written using My_Float rather than Float then replacing the declaration of My_Float by

```
type My_Float is new Long_Float;
```

is the only change necessary. We can actually do better than this by directly stating the precision that we require, thus

```
type My_Float is digits 7;
```

will cause My_Float to be based on the smallest predefined type with at least 7 decimal digits of accuracy.

A similar approach is possible with integer types so that rather than using the predefined types we can give the range of values required thus

```
type My_Integer is range –1000_000 .. +1000_000;
```

The point of all this is that it is not good practice to use the predefined numeric types directly when writing professional programs which may need to be portable. However, for simplicity, we will generally use the types Integer and Float in examples in most of this book. We will say no more about numeric types for the moment except that all the expected operations apply to all integer and floating types.

2.4 Arrays and records

Ada naturally enables the creation of composite array and record types. Arrays may actually be declared without giving a name to the underlying type (the type is then said to be anonymous) but records always have a type name.

As an example of the use of arrays suppose we wish to compute the successive rows of Pascal's triangle. This is usually represented as shown in Figure 2.1. The reader will recall that the rows are the coefficients in the expansion of $(1 + x)^n$ and that a neat way of computing the values is to note that each one is the sum of the two diagonal neighbours in the row above.

Suppose that we are interested in the first ten rows. We could declare an array to hold such a row by

```
Pascal: array (0 .. 10) of Integer;
```

and now assuming that the current values of the array Pascal correspond to row n–1, with the component Pascal(0) being 1, then the next row could be computed in a similar array Next by

```
Next(0) := 1;
for I in 1 .. N–1 loop
   Next(I) := Pascal(I–1) + Pascal(I);
end loop;
Next(N) := 1;
```

and then the array Next could be copied into the array Pascal.

This illustrates another form of loop statement where a controlled variable I takes successive values from a range; the variable is automatically declared to be of the type of the range which in this case is Integer. Note that the intermediate array Next could be avoided by iterating backwards over the array; we indicate this by writing **reverse** in front of the range thus

```
Pascal(N) := 1;
for I in reverse 1 .. N–1 loop
   Pascal(I) := Pascal(I–1) + Pascal(I);
end loop;
```

We can also declare arrays of several dimensions. So if we wanted to keep all the rows of the triangle we might declare

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

Figure 2.1 Pascal's triangle.

```
Pascal2: array (0 .. 10, 0 .. 10) of Integer;
```

and then the loop for computing row *n* would be

```
Pascal2(N, 0) := 1;
for I in 1 .. N–1 loop
   Pascal2(N, I) := Pascal2(N–1, I–1) + Pascal2(N–1, I);
end loop;
Pascal2(N, N) := 1;
```

We have declared the arrays without giving a name to their type. We could alternatively have written

```
type Row is array (0 .. Size) of Integer;
Pascal, Next: Row;
```

where we have given the name Row to the type and then declared the two arrays Pascal and Next. There are advantages to this approach as we shall see later. Incidentally the bounds of an array do not have to be constant, they could be any computed values such as the value of some variable Size.

We conclude this brief discussion of arrays by noting that the type String which we encountered in Section 2.2 is in fact an array whose components are of the enumeration type Character. Its declaration (in the package Standard) is

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

and this illustrates a form of type declaration which is said to be indefinite because it does not give the bounds of the array; these have to be supplied when an object is declared

```
A_Buffer: String(1 .. 80);
```

Incidentally the identifier Positive in the declaration of the type String denotes what is known as a subtype of Integer; values of the subtype Positive are the positive integers and so the bounds of all arrays of type String must also be positive – the lower bound is of course typically 1 but need not be.

A record is an object comprising a number of named components typically of different types. We always have to give a name to a record type. If we were manipulating a number of buffers then it would be convenient to declare a record type containing the buffer and an indication of the start and finish of that part of the buffer actually containing useful data.

```
type Buffer is
   record
      Data: String(1 .. 80);
      Start, Finish: Integer;
   end record;
```

An individual buffer could then be declared by

```
My_Buffer: Buffer;
```

and the components of the buffer can then be manipulated using a dotted notation to select the individual components

```
My_Buffer.Start := 1;
My_Buffer.Finish := 3;
My_Buffer.Data(1 .. 3) := "XYZ";
```

Note that the last statement assigns values to the first three components of the array My_Buffer.Data using a so-called slice.

Whole array and record values can be created using aggregates which are simply a set of values in parentheses separated by commas.

Thus we could assign appropriate values to Pascal and to My_Buffer by

```
Pascal(0 .. 4) := (1, 4, 6, 4, 1);
My_Buffer := (Data => ('X', 'Y', 'Z', others => ' '), Start => 1, Finish => 3);
```

where in the latter case we have in fact assigned all 80 values to the array My_Buffer.Data and used **others** so that after the three useful characters the remainder of the array is padded with spaces. Note the nesting of parentheses and the optional use of named notation for the record components.

This concludes our brief discussion on simple arrays and records. In the next chapter we will show how record types can be extended.

Exercise 2.4

1 Write statements to copy the array Next into the array Pascal.

2 Write a nested loop to compute all the rows of Pascal's triangle in the two-dimensional array Pascal2.

3 Declare a type Month_Name and then declare a type Date with components giving the day, month and year. Then declare a variable Today and assign Queen Victoria's date of birth to it (or your own).

2.5 Access types

The previous section showed how the scalar types (numeric and enumeration types) may be composed into arrays and records. The other vital means for creating structures is through the use of access types (the Ada name for pointer types); access types allow list processing and are typically used with record types.

The explicit manipulation of pointers or references has been an important feature of most languages since Algol 68. References rather dominated Algol 68 and caused problems and the corresponding pointer facility in Pascal is rather austere. The pointer facility in C on the other hand provides raw flexibility which is open to abuse and quite insecure and thus the cause of many wrong programs.

Ada provides both a high degree of reliability and considerable flexibility through access types. A full description will be found in Chapter 11 but the following brief description will be useful for discussing polymorphism in the next chapter.

Ada access types must explicitly indicate the type of data to which they refer. The most general form of access types can refer to any data of the type concerned but we will restrict ourselves in this overview to applications which just refer to data declared in a storage pool (the Ada term for a heap).

For example suppose we wanted to declare various buffers of the type Buffer in the previous section. We might write

```
Handle: access Buffer;
...
Handle := new Buffer;
```

This allocates a buffer in the storage pool and sets a reference to it into the variable Handle. We can then refer to the various components of the buffer indirectly using the variable Handle

```
Handle.Start := 1;
Handle.Finish := 3;
```

and we can refer to the complete record as Handle.**all**. Note that Handle.Start is strictly an abbreviation for Handle.**all**.Start.

Access types are of particular value for list processing where one record structure contains an access value to another record structure. The classic example which we will encounter in many forms is typified by

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

The type Cell is a record containing a component Next which can refer to another similar record plus a component of some type Data. An example of the use of this sort of construction will be found in the next chapter.

Sometimes it is important to give a name to an access type. We can rewrite the above example so that the component Next is of a named type by first using an incomplete type thus

```
type Cell;                                  -- incomplete declaration
type Cell_Ptr is access Cell;

type Cell is                                -- the completion
   record
      Next: Cell_Ptr;
      Value: Data;
   end record;
```

Using this two stage approach and naming the access type is necessary if we are doing our own control of storage as described in Section 25.4.

Access types often refer to record types as in these examples but can refer to any type. Access types may also be used to refer to subprograms and this is particularly important for certain repetitive operations and when communicating with programs in other languages.

2.6 Errors and exceptions

We introduce this topic by considering what would have happened in the example in Section 2.2 if we had not tested for a negative value of X and consequently called Sqrt with a negative argument. Assuming that Sqrt has itself been written in an appropriate manner then it clearly cannot deliver a value to be used as the parameter of Put. Instead an exception will be raised. The raising of an exception indicates that something unusual has happened and the normal sequence of execution is broken. In our case the exception might be Constraint_Error which is a predefined exception declared in the package Standard. If we did nothing to cope with this possibility then our program would be terminated and no doubt the Ada Run Time System will give us a rude message saying that our program has failed and why. We can, however, look out for an exception and take remedial action if it occurs. In fact we could replace the conditional statement

```
if X < 0.0 then
   Put("not calculable");
else
   Put(Sqrt(X));
end if;
```

by the block

```
begin
   Put(Sqrt(X));
exception
   when Constraint_Error =>
      Put("not calculable");
end;
```

If an exception is raised by the sequence of statements between **begin** and **exception**, then control immediately passes to the one or more statements following the handler for that exception and these are obeyed instead. If there were no handler for the exception (it might be another exception such as Storage_Error) then control passes up the nested sequence of calls until we come to an appropriate handler or fall out of the main subprogram, which then becomes terminated as we mentioned with a message from the Run Time System.

The above example is not a good illustration of the use of exceptions since the event we are guarding against can easily be tested for directly. Nevertheless it does show the general idea of how we can look out for unexpected events, and leads us into a brief consideration of errors in general.

From the linguistic viewpoint, an Ada program may be incorrect for various reasons. There are four categories according to how they are detected.

- Many errors are detected by the compiler – these include simple punctuation mistakes such as leaving out a semicolon or attempting to violate the type rules such as mixing up colours and fishes. In these cases the program is said to be illegal and will not be executed.

- Other errors are detected when the program is executed. An attempt to find the square root of a negative number or divide by zero are examples of such errors. In these cases an exception is raised as we have just seen.
- There are also certain situations where the program breaks the language rules but there is no simple way in which this violation can be detected. For example a program should not use a variable before a value is assigned to it. In cases like this the behaviour is not predictable but will nevertheless lie within certain bounds. Such errors are called bounded errors.
- In more extreme situations there are some kinds of errors which can lead to quite unpredictable behaviour. In these (quite rare) cases we say that the behaviour is erroneous.

Finally, there are situations where, for implementation reasons, the language does not prescribe the order in which things are to be done. For example, the order in which the parameters of a procedure call are evaluated is not specified. If the behaviour of a program does depend on such an order then it is not considered to be incorrect but just not portable.

We mentioned earlier that an overall theme in the design of Ada was concern for correctness and that errors should be detected early in the programming process. As a simple example consider a fragment of program controlling the crossing gates on a railroad. First we have an enumeration type describing the state of a signal

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

and then perhaps

```
if The_Signal = Clear then
   Open_Gates;
   Start_Train;
end if;
```

It is instructive to consider how this might be written in C and then to consider the consequences of various simple programming errors. Enumeration types in C are not strongly typed and essentially just provide names for integer (int) constants with values 0, 1 and 2 representing the three states. This has potential for errors because there is nothing in C that can prevent us from assigning a silly value such as 4 to a signal whereas it is not even possible to attempt such a thing when using the Ada enumeration type.

The corresponding text in C might be

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

It is interesting to consider what would happen in the two languages if we make various typographical errors. Suppose first that we accidentally type an extra semicolon at the end of the first line. The Ada program then fails to compile and the error is immediately drawn to our attention; the C program however still compiles and the condition is ignored (since it then controls no statements). The C program

consequently always opens the gates and starts the train irrespective of the state of the signal!

Another possibility is that one of the = signs might be omitted in C. The equality then becomes an assignment and also returns the result as the argument for the test. The program still compiles, the signal is set clear, the condition is true (since clear is not 0) and so the gates are always opened and the train started on its perilous journey. The corresponding error in Ada might be to write := instead of = and of course the program will then not compile.

Of course, many errors cannot be detected at compile time. For example using My_Buffer and a variable Index of type Integer, we might write

```
Index := 81;
...
My_Buffer.Data(Index) := 'x';
```

which attempts to write to the 81st component of the array which does not exist. Such assignments are checked in Ada at run time and Constraint_Error would be raised so that the integrity of the program is not violated.

The corresponding instructions in C would undoubtedly overwrite an adjacent piece of storage and probably corrupt the value in My_Buffer.Start.

It often happens that variables such as Index can only sensibly have a certain range of values; this can be indicated by introducing a subtype

```
subtype Buffer_Index is Integer range 1 .. 80;
Index: Buffer_Index;
```

or by indicating the constraint directly

```
Index: Integer range 1 .. 80;
```

Applying a constraint to Index has the advantage that the attempt to assign 81 to it is itself checked and prevented so that the error is detected even earlier.

The reader may feel that such checks will make the program slower. This is not usually the case as we shall see in Chapter 15.

2.7 Terminology

We conclude this first introductory chapter with a few remarks on terminology. Every subject has its own terminology or jargon and Ada is no exception. (Indeed in Ada an exception is a kind of error as we have seen!) A useful glossary of terms will be found in Appendix 2.

Terminology will generally be introduced as required but before starting off with the detailed description of Ada it is convenient to mention a few concepts which will occur from time to time.

The term *static* refers to things that can be determined at compilation, whereas *dynamic* refers to things determined during execution. Thus a static expression is one whose value can be determined by the compiler such as

```
2 + 3
```

and a statically constrained array is one whose bounds are known at compilation.

The term *real* comes up from time to time in the context of numeric types. The floating point types and fixed point types are collectively known as real types. (Fixed point types are rather specialized and not discussed until Chapter 17.) Literals such as 2.5 are known as real literals since they can be used to denote values of both floating and fixed point types. Other uses of the term real will occur in due course.

The terminology used with exceptions in Ada is that an exception is *raised* and then *handled*. Some languages say that an exception is thrown and then caught.

We talk about statements being *executed* and expressions being *evaluated*. Moreover, declarations can also require processing and this is called being *elaborated*.

Object oriented programming has its own rather specialized terminology and a section is devoted to this in the next chapter. One particular term that seems to be overused is *interface*. We use interface in a very general sense to mean the description of a means of communication. As we noted in Section 2.2, the specification of a package describes its interface to the outside world. But interface also has a highly technical meaning in OOP as we shall see in the next chapter. As usual in any language, the context should clarify the intended meaning.

3 Abstraction

As mentioned in Chapter 1, abstraction in various forms seems to be the key to the development of programming languages. In this chapter we survey various aspects of abstraction with particular emphasis on the object oriented paradigm.

3.1 Packages and private types

In the previous chapter we declared a type for the manipulation of a buffer as follows

```
type Buffer is
   record
      Data: String(1 .. 80);
      Start: Integer;
      Finish: Integer;
   end record;
```

in which the component Data actually holds the characters in the buffer and Start and Finish index the ends of the part of the buffer containing useful information. We also saw how the various components might be updated and read using normal assignment.

However, such direct assignment is often unwise since the user could inadvertently set inconsistent values into the components or read nonsense components of the array.

A much better approach is to create an Abstract Data Type (ADT) so that the user cannot see the internal details of the type but can only access it through various subprogram calls which define an appropriate protocol.

This can be done using a package containing a private type. Let us suppose that the protocol allows us to reload the buffer (possibly not completely full) and to read one character at a time. Consider the following

```
package Buffer_System is                    -- visible part

   type Buffer is private;
   procedure Load(B: out Buffer; S: in String);
   procedure Get(B: in out Buffer; C: out Character);

private                                     -- private part
   Max: constant Integer := 80;
   type Buffer is
      record
         Data: String(1 .. Max);
         Start: Integer := 1;
         Finish: Integer := 0;
      end record;
end Buffer_System;

package body Buffer_System is               -- package body
   procedure Load(B: out Buffer; S: in String) is
   begin
      B.Start := 1;
      B.Finish := S'Length;
      B.Data(B.Start .. B.Finish) := S;
   end Load;

   procedure Get(B: in out Buffer; C: out Character) is
   begin
      C := B.Data(B.Start);
      B.Start := B.Start + 1;
   end Get;
end Buffer_System;
```

Note how the package comes in two parts, the specification and the body. Basically the specification describes the interface to other parts of the program and the body gives implementation details.

With this formulation the client can only access the information in the visible part of the specification which is the bit before the word **private**. In this visible part the declaration of the type Buffer merely says that it is private and the full declaration then occurs in the private part. There are thus two views of the type Buffer; the external client just sees the partial view whereas within the package the code of the server subprograms can see the full view. The specifications of the server subprograms are naturally also declared in the visible part and the full bodies which give their implementation details are in the package body.

The net effect is that the user can declare and manipulate a buffer by simply writing

```
My_Buffer: Buffer;
...
```

```
Load(My_Buffer, Some_String);
...
Get(My_Buffer, A_Character);
```

but the internal structure is quite hidden. There are two advantages: one is that the user cannot inadvertently misuse the buffer and the second is that the internal structure of the private type could be rearranged if necessary and provided that the protocol is maintained the user program would not need to be changed.

This hiding of information and consequent separation of concerns is very important and illustrates the benefit of data abstraction. The design of appropriate interface protocols is the key to the development and subsequent maintenance of large programs.

The astute reader will note that we have not bothered to ensure that the buffer is not loaded when there is still unread data in it or the string is too long to fit, nor read from when it is empty. We could rectify this by declaring our own exception called perhaps Error in the visible part of the specification thus

```
Error: exception;
```

and then check within Load by for example

```
if S'Length > Max or B.Start <= B.Finish then
   raise Error;
end if;
```

This causes our own exception to be raised if an attempt is made to overwrite existing data or if the string is too long.

As a minor point note the use of the constant Max so that the literal 80 only appears in one place. Note also the attribute Length which applies to any array and gives the number of its components. The upper and lower bounds of an array S are incidentally given by S'First and S'Last.

Another point is that the parameter Buffer of Get is marked as **in out** because the procedure both reads the initial value of Buffer and updates it.

Finally, note that the components Start and Finish of the record have initial values in the declaration of the record type; these ensure that when a buffer is declared these components are assigned sensible values and thereby indicate that the buffer is empty. An alternative would of course be to provide a procedure Reset but the user might forget to call it.

Exercise 3.1

1 Extend the package Buffer_System to include the exception Error in its specification and appropriate checks in the subprogram bodies. Also add a function Is_Empty visible to the user.

3.2 Objects and inheritance

The term object oriented programming is currently in vogue. A precise definition is hard and is made worse by a complete lack of agreement on appropriate

terminology. Ada uses carefully considered terminology which is rather different from that in some languages such as C++ but avoids the ambiguities which can arise from the confusing overuse of terms such as class.

As its name suggests, object oriented programming concerns the idea of programming around objects. A good example of an object in this sense is the variable My_Buffer of the type Buffer in the previous section. We conceive of the object such as a buffer as a coordinated whole and not just as the sum of its components (corresponding to a holistic rather than a reductionist view using terminology from Quantum Mechanics). Indeed the external user cannot see the components at all but can only manipulate the buffer through the various subprograms associated with the type.

Certain operations upon a type are called the primitive operations of the type. In the case of the type Buffer they are the subprograms declared in the package specification along with the type itself and which have parameters or a result of the type. In the case of a type such as Integer, the primitive operations are those such as + and – which are predefined for the type (and indeed they are declared in Standard along with the type Integer itself and so fit the same model). Such operations are called methods in some languages.

Other important ideas in OOP are

- the ability to define one type in terms of another and especially as an extension of another; this is type extension,
- the ability for such a derived type to inherit the primitive operations of its parent and also to add to and replace such operations; this is inheritance,
- the ability to distinguish the specific type of an object at run time from among several related types and in particular to select an operation according to the specific type; this is (dynamic) polymorphism.

In the previous chapter we showed how the type Light was derived from Colour and also showed how numeric portability could be aided by deriving a numeric type such as My_Float from one of the predefined types. These were very simple forms of inheritance; the new types inherited the primitive operations of the parent; however, the types were not extended in any way and underneath were really the same type. The main benefit of such derivation is simply to provide a different name and thereby to distinguish the different uses of the same underlying type in order to prevent us from inadvertently using a Light when we meant to use a Colour.

The more general case is where we wish to extend a type in some way and also distinguish objects of different types at run time. The most natural form of type for the purposes of extension is of course a record where we can consider extension as simply the addition of further components. The other point is that if we need to distinguish the type at run time then the object must contain an indication of its type. This is provided by a hidden component called the tag. Type extension in Ada is thus naturally carried out using tagged record types.

As a simple example suppose we wish to manipulate various kinds of geometrical objects. We can imagine that the kinds of objects form a hierarchy as shown in Figure 3.1. Thus we have points, polygons, circles, and shapes; and polygons are further subdivided into quadrilaterals and pentagons.

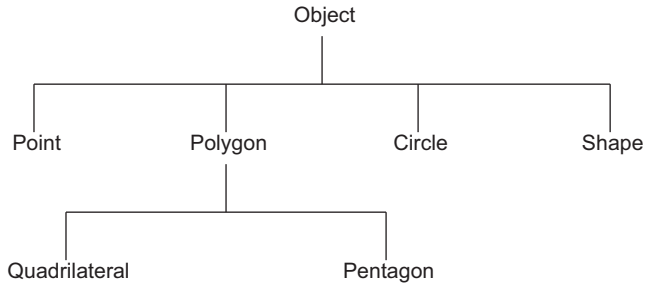


Figure 3.1 A hierarchy of geometrical objects.

All objects will have a position given by their x- and y-coordinates. So we declare the root of the hierarchy as

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

Note carefully the introduction of the reserved word **tagged**. This indicates that values of the type carry a tag at run time and that the type can be extended. The other types of geometrical objects will be derived (directly or indirectly) from this type. For example we could have

```
type Circle is new Object with
   record
      Radius: Float;
   end record;
```

and the type Circle then has the three components X_Coord, Y_Coord and Radius. It inherits the two coordinates from the type Object and the component Radius is added explicitly.

Sometimes it is convenient to derive a new type without adding any further components. For example

```
type Point is new Object with null record;
```

In this last case we have derived Point from Object but not added any new components. However, since we are dealing with tagged types we explicitly add **with null record**; to indicate that we did not want any new components. This means that it is clear from a declaration whether a type is tagged or not since it will have either **tagged** or **with** (or **interface** as we shall see later) if it is tagged.

The primitive operations of a type are those declared in the same package specification as the type and that have parameters or result of the type. On derivation these operations are inherited by the new type. They can be overridden by new versions and new operations can be added and these then become primitive operations of the new type and are themselves inherited by any further derived type.

Thus we might have declared a function giving the distance from the origin

```
function Distance(O: in Object) return Float is
begin
   return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
end Distance;
```

The type Circle would then sensibly inherit this function. If however, we were concerned with the area of an object then we might start with

```
function Area(O: in Object) return Float is
begin
   return 0.0;
end Area;
```

which returns zero since a raw object has no area. The abstract concept of an area applies also to a circle and so it is appropriate that a function Area be defined for the type Circle. However, to inherit the function from the type Object is clearly inappropriate and so we explicitly declare

```
function Area(C: in Circle) return Float is
begin
   return π * C.Radius**2;
end Area;
```

which then overrides the inherited operation. (We assume that π is an appropriate constant such as that declared in Ada.Numerics; see Section 4.3.)

We can summarize these ideas by saying that the specification is always inherited whereas the implementation may be inherited but can be replaced.

It is important to remember that new primitive operations must be declared in the same package specification as the type itself. A derived type might be declared in the same package as its parent or it might be in a different package. So in the former case the overall structure might be

```
package Geometry is

   type Object is tagged ... ;

   function Distance(O: in Object) return Float;
   function Area(O: in Object) return Float;

   type Circle is new Object with ... ;

   function Area(C: in Circle) return Float;

   type Point is new Object with null record;

end Geometry;

package body Geometry is
   ...      -- bodies of function Distance and two functions Area
end Geometry;
```

Another approach is to declare each type in a distinct library package; child packages especially have a number of advantages such as minimizing the need for

with clauses as we shall see in Program 1. Mixed strategies are also possible as illustrated in the exercise at the end of this section.

It is possible to convert a value from the type Circle to Object and vice versa. From circle to object is straightforward, we simply write

```
O: Object := (1.0, 0.5);
C: Circle := (0.0, 0.0, 34.7);
...
O := Object(C);
```

which effectively ignores the radius component. However, conversion in the other direction requires the provision of a value for the extra component and this is done by an extension aggregate thus

```
C := (O with Radius => 41.2);
```

where the expression O is extended after **with** by values for the extra components (in this case only the radius) written just as in a normal aggregate.

We conclude by noting that a private type can also be marked as tagged

```
type Shape is tagged private;
```

and the full type declaration must then (ultimately) be a tagged record

```
type Shape is tagged
   record ...
```

or derived from some tagged record type. On the other hand we might wish to make visible the fact that the type Shape is derived from Object and yet keep the additional components hidden. In this case we might write

```
with Geometry;
package Hidden_Shape is
   type Shape is new Geometry.Object with private;    -- client view
   ...
private
   type Shape is new Geometry.Object with             -- server view
      record
         ...       -- the private components
      end record;
end Hidden_Shape;
```

Note that it is not necessary for the full declaration of Shape to be derived directly from the type Object. It might be derived from Circle; all that matters is that Shape is ultimately derived from Object.

Exercise 3.2

1 Declare the type Object and the functions Distance and Area in a package Objects. Then declare a package Shapes containing the types Circle and Point and also a type Triangle with sides A, B and C and appropriate functions returning the area. (Assume that Sqrt and π are somehow directly visible.)

3.3 Classes and polymorphism

In the previous section we showed how to declare a hierarchy of types derived from the type Object. We saw that on derivation further components and operations could be added and that operations could be replaced.

However, it is very important to note that an operation cannot be taken away nor can a component be removed. As a consequence we are guaranteed that all the types derived from a common ancestor will have all the components and operations of that ancestor.

So in the case of the type Object, all types in the hierarchy derived from Object will have the common components such as their coordinates and the common operations such as Distance and Area. Since they have these common properties it is natural that we should be able to manipulate a value of any type in the hierarchy without knowing exactly which type it is provided that we only use the common properties. Such general manipulation is done through the concept of a class.

Ada carefully distinguishes between an individual type such as Object on the one hand and the set of types such as Object plus all its derivatives on the other hand. A set of such types is known as a class. Associated with each class is a type called the class wide type which for the set rooted at Object is denoted by Object'Class. The type Object is referred to as a specific type when we need to distinguish it from a class wide type.

We can of course have subclasses, for example Polygon'Class represents the set of all types derived from and including Polygon. This is a subset of the class Object'Class. All the properties of Object'Class will also apply to Polygon'Class but not vice versa. For example, although we have not shown it, the type Polygon will presumably contain a component giving the length of the sides. Such a component will belong to all types of the class Polygon'Class but not to Object'Class.

As a simple example of the use of a class wide type consider the following function

```
function Moment(OC: Object'Class) return Float is
begin
   return OC.X_Coord * Area(OC);
end Moment;
```

Those who recall their school mechanics will remember that the moment of a force about a fulcrum is the product of the force by the distance from the fulcrum. So in our example, taking the x-axis as horizontal and the y-axis as vertical, the moment of the force of gravity on an object about the origin is the x-coordinate multiplied by its weight which we can take as being proportional to its area (and for simplicity we have taken to be just its area).

This function has a formal parameter of the class wide type Object'Class. This means it can be called with an actual parameter whose type is any specific type in the class comprising the set of all types derived from Object. Thus we could write

```
C: Circle ...
M: Float;
...
M := Moment(C);
```

Within the function Moment we can naturally refer to the specific object as OC. Since we know that the object must be of a specific type in the Object class, we are guaranteed that it will have a component OC.X_Coord. Similarly we are guaranteed that the function Area will exist for the type of the object since it is a primitive operation of the type Object and will have been inherited by (and possibly overridden for) every type derived from Object. So the appropriate function Area is called and the result multiplied by the *x*-coordinate and returned as the result of the function Moment.

Note carefully that the particular function Area to be called is not known until the program executes. The choice depends upon the specific type of the parameter and this is determined by the tag of the object passed as the actual parameter; remember that the tag is a sort of hidden component of the tagged type. This selection of the particular subprogram according to the tag is known as dispatching and is a vital aspect of the dynamic behaviour provided by polymorphism.

Dispatching only occurs when the actual parameter is of a class wide type; if we call Area with an object of a specific type such as C of type Circle then the choice is made at compile time. Dispatching is often called late binding because the call is only bound to the called subprogram late in the compile–link–execute process. The binding to a call of Area with the parameter C of type Circle is called static binding because the subprogram to be called is determined at compile time.

Observe that the function Moment is not a primitive operation of any type; it is just an operation of Object'Class and it happens that a value of any specific type derived from Object is implicitly converted to the class wide type when passed as a parameter. Class wide types do not have primitive operations and so no inheritance is involved.

It is interesting to consider what would have happened if we had written

```
function Moment(O: Object) return Float is
begin
   return O.X_Coord * Area(O);
end Moment;
```

where the formal parameter is of the specific type Object. This always returns zero because the function Area for an Object always returns zero. If this function Moment were declared in the same package specification as Object then it would be a primitive operation of Object and thus inherited by the type Circle. However, the internal call would still be to the function Area for the type Object and not to the type Circle and so the answer would still be zero. This is because the binding is static and inheritance simply passes on the same code. The code mechanically works on a Circle because it only uses the Object part of the circle (we say that it sees the Object view of the Circle); but unfortunately it is not what we want. We could of course override the inherited operation by writing

```
function Moment(C: Circle) return Float is
begin
   return C.X_Coord * Area(C);
end Moment;
```

but this is both tedious and causes unnecessary duplication of similar code. The proper approach for such general situations is to use the original class wide version

with its internal dispatching; this can be shared by all types without duplication and always calls the appropriate function Area.

A major advantage of using a class wide operation such as Moment is that a system using it can be written, compiled and tested without knowing all the specific types to which it is to be applied. Moreover, we can then add further types to the system without recompilation of the existing tested system.

For example we could add a further type

```
type Pentagon is new Object with ...
function Area(P: Pentagon) return Float;
...
Star: Pentagon := ...
...
Put("Moment of star is ");
Put(Moment(Star));
```

and then the old existing tried and tested Moment will call the new Area for the Pentagon without being recompiled. (It will of course have to be relinked.)

This works because of the mechanism used for dynamic binding; the essence of the idea is that the class wide code has dynamic links into the new code and this is accessed via the tag of the type. This creates a very flexible and extensible interface ideal for building up a system from reusable components. Details of how this apparent magic might be implemented will be outlined when we discuss OOP in detail in Chapter 14.

One difficulty with the flexibility provided by class wide types is that we cannot know how much space might be occupied by an arbitrary object of the type because the type might be extended. So although we can declare an object of a class wide type it has to be initialized and thereafter that object can only be of the specific type of that initial value. Note that a formal parameter of a class wide type such as in Moment is allowed because it is similarly initialized by the actual parameter.

Another similar restriction is that we cannot have an array of class wide components (even if initialized) because the components might be of different specific types and so probably of different sizes and impossible to index efficiently.

One consequence of these necessary restrictions is that it is very natural to use access types with object oriented programming since there is no problem with pointing to objects of different sizes at different times.

Thus suppose we wanted to manipulate a series of geometrical objects; it is very natural to declare these in free storage as required. They can then be chained together on a list for processing. Consider

```
type Pointer is access Object'Class;

type Cell is
   record
      Next: access Cell;
      Element: Pointer;
   end record;
...
type List is access Cell;
A_List: List;
```

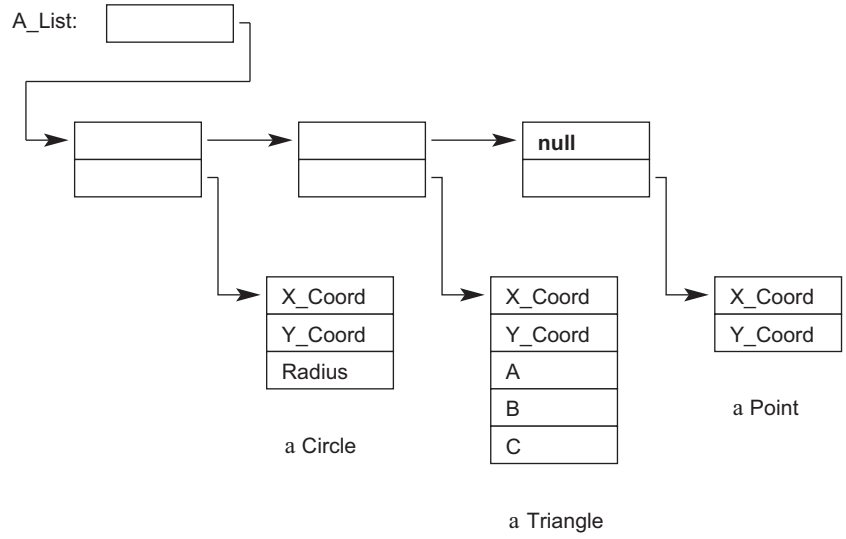


Figure 3.2 A chain of objects.

which enables us to create cells which can be linked together; each cell has an element which is a pointer to any geometrical object. We have also declared a named access type List and a variable of that type. Several objects can then be created and linked together to form a list as in Figure 3.2.

We can now easily process the objects on the list and might for example compute the total moment of the set of objects by calling the following function

```
function Total_Moment(The_List: List) return Float is
   Local: access Cell := The_List;
   Result: Float := 0.0;
begin
   loop
      if Local = null then                    -- end of list
         return Result;
      end if;
      Result := Result + Moment(Local.Element.all);
      Local := Local.Next;
   end loop;
end Total_Moment;
```

We conclude this brief survey of the OOP facilities in Ada by considering abstract types. It is sometimes the case that we would like to declare a type as the foundation for a class of types with certain common properties but without allowing objects of the original type to be declared. For example we probably would not want to declare an object of the raw type Object. If we wanted an object without any area then it would be appropriate to declare a Point. Moreover, the function Area for the

type Object is dubious since it usually has to be overridden anyway. But it is important to be able to ensure that all types derived from Object do have an Area so that the dispatching in the function Moment always works. We can achieve this by writing

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

   function Distance(O: in Object) return Float;
   function Area(O: in Object) return Float is abstract;
end Objects;
```

In this formulation the type Object and the function Area are marked as abstract. It is illegal to declare an object of an abstract type and an abstract subprogram has no body and so cannot be called. On deriving a concrete (that is, nonabstract) type from an abstract type any abstract inherited operations must be overridden by concrete operations. Note that we have declared the function Distance as not abstract; this is largely because we know that it will be appropriate anyway.

This approach has a number of advantages; we cannot declare a raw Object by mistake, we cannot inherit the silly body of Area and we cannot make the mistake of declaring the function Moment for the specific type Object (why?).

But despite the type being abstract we can declare the function Moment for the class wide type Object'Class. This always works because of the rule that we cannot declare an object of an abstract type; any actual object passed as a parameter must be of a concrete type and will have an appropriate function Area to which it can dispatch.

Note that it would be very sensible for the function Distance also to take a class wide parameter. This is because Distance is inherently the same for all types in the class and so cannot need to be overridden. So we would write

```
function Distance(OC: in Object'Class) return Float is
begin
   return Sqrt(OC.X_Coord**2 + OC.Y_Coord**2);
end Distance;
```

If an abstract type has no concrete operations at all and has no components then it can be declared as an interface. Interfaces are important for multiple inheritance where a type inherits properties from more than one ancestor. Thus if we decided that the type Object should not have any components (so that they would have to be added by each type such as Circle) then we could write

```
package Objects is
   type Object is interface;

   function Distance(OC: in Object'Class) return Float;
   function Area(O: in Object) return Float is abstract;
   function X_Coord(O: in Object) return Float is abstract;
```

```
   function Y_Coord(O: in Object) return Float is abstract;
end Objects;
```

We have also added functions to return the values of the coordinates since these are needed by the class wide function Distance.

The normal way of calling a subprogram in classical languages is to give its name and then a list of parameters. However, in OOP it is often better to consider the object as dominant and to give that first followed by the subprogram name and any other parameters. This is also possible in Ada for operations of tagged types provided always that the first parameter denotes the object. Thus we can write either of

```
X := Area(C);          -- classical notation
X := C.Area;           -- prefixed notation
```

There are a number of advantages of the prefixed notation which will be explained in detail in Chapter 14. An immediately obvious one is that it unifies access to a component and to a corresponding function. Thus we could write

```
function Moment(OC: in Object'Class) return Float is
begin
   return OC.X_Coord * OC.Area;
end Moment;
```

and this treats the function Area and the component X_Coord in the same manner. Moreover, if we later decided to use the interface style where the component value is returned by a function X_Coord then the text of the function Moment remains the same. And of course the function Distance remains the same as well.

Exercise 3.3

1 Declare a procedure Add_To_List which takes a list of type List and (an access to) any object of the type Object'Class and adds it to the list.

2 Write the body of the package Objects for the version with the abstract type.

3 How would the package Shapes of Exercise 3.2(**1**) need to be modified using the package Objects with the abstract type?

4 Why could we not declare the function Moment for the abstract type Object?

5 The moment of inertia of an object about the origin M_O is equal to its moment of inertia about its centre of gravity M_I plus MR^2 where M is its mass and R its distance from the origin. Assuming that we have functions MI for each specific type declare a class wide function MO returning the moment of inertia about the origin.

3.4 Genericity

We have seen how class wide types provide us with dynamic polymorphism. This means that we can manipulate several types with a single construction and that the specific type is determined dynamically, that is, at run time. In this

section we introduce the complementary concept of static polymorphism where again we have a single construction for the manipulation of several types but in this case the choice of type is made statically at compile time.

At the beginning of Section 2.2 we said that an important objective of software engineering is to reuse existing software components. However, the strong typing model of Ada (even with class wide types) sometimes gets in the way unless we have a method of writing software components which can be used for various different types. For example, the program to do a sort is largely independent of what it is sorting – all it needs is a rule for comparing the values to be sorted.

So we need a means of writing pieces of software that can be parameterized as required for different types. In Ada this is done by the generic mechanism. We can make a package or subprogram generic with respect to one or more parameters which can include types. Such a generic unit provides a template from which we can create genuine packages and subprograms by so-called instantiation. The full details of Ada generics are quite extensive and will be dealt with in Chapter 18. However, in the next chapter we will be discussing input–output and other aspects of the predefined library which make significant use of the generic mechanism and so a brief introduction is appropriate.

The standard package for the input and output of floating point values in text form is generic with respect to the actual floating type. This is because we want a single package to cope with all the possible floating types such as the underlying machine types Float and Long_Float as well as the portable type My_Float. Its specification is

```
generic
   type Num is digits <>;
package Float_IO is
   ...
   procedure Get(Item: out Num; ... );
   procedure Put(Item: in Num; ... );
   ...
end Float_IO;
```

where we have omitted various details relating to the format. The one generic parameter is Num and the notation **digits** <> indicates that it must be a floating point type and echoes the declaration of My_Float using **digits** 7 that we briefly mentioned in Section 2.3.

In order to create an actual package to manipulate values of the type My_Float, we write an instantiation thus

```
package My_Float_IO is new Float_IO(My_Float);
```

This creates a package with the name My_Float_IO where the formal type Num has been replaced throughout with our actual type My_Float. As a consequence, procedures Get and Put taking parameters of the type My_Float are created and we can then call these as required. But we are straying into the next chapter.

The kind of parameterization provided by genericity is similar but rather different from that provided through class wide types. In both cases the

parameterization is over a related set of types with a set of common properties. Such a related set of types is termed a category of types.

A common form of category is a derivation class where all the types are derived from a common ancestor such as the type Object. Tagged derivation classes form the basis of dynamic polymorphism as we have seen.

But there are also broader forms of categories such as the set of all floating point types which are allowed as actual parameters for the generic package Float_IO. The parameters of generic units can use these broader categories. For example, a very broad category is the set of all types having assignment. Such a category would be a suitable basis for writing a generic sort routine.

In due course we shall see that the two forms of polymorphism work together; a common form of generic package is one which takes as a parameter a type from a tagged derivation class.

3.5 Object oriented terminology

It is perhaps convenient at this point to compare the Ada terminology with that used by other object oriented languages such as Smalltalk, C++ and Java. Those not familiar with such languages could skip this section before they get confused.

At least one term in common is inheritance. Ada 83 had inheritance although not type extension. Ada 95 introduced type extension and dynamic polymorphism with tagged types. Untagged record types are called structs in some languages.

Ada actually uses the term *object* to denote variables and constants in general whereas in the OO sense an object is an instance of an Abstract Data Type (ADT).

Many languages use *class* to denote what Ada calls a specific tagged type (or more strictly an ADT consisting of a tagged type plus its primitive operations). Ada (like the functional language Haskell) uses the word class to refer to a group of related types and not to a single type. The Ada approach clarifies the distinction between the group of types and a single type and strengthens that clarification by introducing class wide types as such. Many languages use the term class for both specific types and the group of types with much resulting confusion both in terms of description but also in understanding the behaviour of the program and keeping track of the real nature of an object. (Logically the distinction is between a set and the members of the set; we confuse these in casual human language but it's best to avoid the confusion if possible and Ada does just that.)

Primitive operations of Ada tagged types are often called methods or virtual functions. The call of a primitive operation in Ada is bound statically or dynamically according to whether the actual parameter is of a specific type or a class wide type. The rules in C++ are more complex and depend upon whether the parameter is a pointer and also whether the call is prefixed by its class name and whether it is virtual or not. In Ada, an operation with class wide formal parameters is always bound statically although it applies to all types of the class.

Dispatching is the Ada term for calling a primitive operation with dynamic binding and this provides dynamic polymorphism. Subprogram calls via access to subprogram types are also a form of dynamic binding.

Abstract types correspond to abstract classes in many languages. Abstract subprograms are pure virtual member functions in C++. Note that Eiffel uses the term deferred rather than abstract.

Ancestor or parent type and descendant or derived type become superclass and subclass. The Ada concept of subtype has no correspondence in languages which do not have range checks and has no relationship to subclass. An Ada subtype can never have more values than its base type, whereas a descendant type (subclass in other languages) can never have fewer values than its parent type.

Generic units in Ada provide static polymorphism and are similar to templates in C++ and generics in C# and Java. In Ada, the contract model is stronger than in C++ so that more checking is carried out on the generic template whereas C++ leaves many checks to the instance which produces less helpful diagnostics.

Ada includes interfaces which are very similar to interfaces in Java, C#, and CORBA and avoid the complexities of multiple inheritance in C++.

Encapsulation is important. We can distinguish module encapsulation (for information hiding) and object encapsulation (for controlled access). Ada packages provide module encapsulation whereas tasks and protected objects provide object encapsulation. C++ and Java really only provide module encapsulation mainly with classes and (more weakly) with their namespaces and packages. In Ada the package has other purposes unrelated to type extension and inheritance and the private type. Note that the effect of private and protected operations in C++ is provided in Ada by a combination of private types and child packages; the latter are kinds of friends.

3.6 Tasking

No survey of abstraction in Ada would be complete without a brief mention of tasking. It is often necessary to write a program as a set of parallel activities rather than just as one sequential program.

Most programming languages do not address this issue at all. Some argue that the underlying operating system provides the necessary mechanisms and that they are therefore unnecessary in a programming language. Such arguments do not stand up to careful examination for three main reasons

- Built-in syntactic constructions provide a degree of reliability which cannot be obtained through a series of individual operating system calls.
- Portability between systems is difficult if system calls are used directly.
- General purpose operating systems do not provide the degree of control and timing required by many applications.

An Ada program can be written as a series of interacting tasks. There are two main ways in which tasks can communicate: directly by sending messages to each other and indirectly by accessing shared data.

Direct communication between Ada tasks is achieved by one task calling an entry in another task. The calling (client) task waits while the called (server) task executes an accept statement in response to the call; the two tasks are closely coupled during this interaction which is called a rendezvous.

Controlled access to shared data is vital in tasking applications if interference is to be avoided. For example, returning to the character buffer example in Section 3.1, it would be a disaster if a task started to read the buffer while another task was updating it with further information since the component B.Start could be changed and a component of the Data array read by Get before the buffer had been correctly

updated by Load. Ada prevents such interference by a construction known as a protected object.

The syntactic form of both tasks and protected objects is similar to that of a package. They have a specification part describing the interface presented to the client, a private part containing hidden details of the interface, and a body stating what they actually do.

The general client–server model can thus be expressed as

```
task Server is
   entry Some_Service(Formal: in out Data);
end;

task body Server is
begin
   ...
   accept Some_Service(Formal: in out Data) do
      ...      -- statements providing the service
   end;
   ...
end Server;

task Client;

task body Client is
   Actual: Data;
begin
   ...
   Server.Some_Service(Actual);
   ...
end Client;
```

A good example of the form of a protected object is given by the buffer example which could be rewritten as follows

```
protected type Buffer(Max: Integer) is      -- visible part
   procedure Load(S: in String);
   procedure Get(C: out Character);
private                                     -- private part
   Data: String(1 .. Max);
   Start: Integer := 1;
   Finish: Integer := 0;
end Buffer;

protected body Buffer is                    -- body
   procedure Load(S: in String) is
   begin
      Start := 1;
      Finish := S'Length;
      Data(Start .. Finish) := S;
   end Load;
```

```
   procedure Get(C: out Character) is
   begin
      C := Data(Start);
      Start := Start + 1;
   end Get;
end Buffer;
```

This construction uses a slightly different style from the package. It is a type in its own right whereas the package exported the type. Calls of the procedures use the dotted form of notation where the name of the particular protected object acts as a prefix to the call rather than being passed as a parameter. (This is similar to the prefixed notation for calling operations of tagged types mentioned in Section 3.3.) Another point is that within the bodies of the procedures, the references to the private data are naturally taken to refer to the current instance.

Note also that the type is parameterized by the discriminant Max. This enables us to give the actual size of a particular buffer when it is declared thus

```
B: Buffer(80);
```

Statements using the protected object B might then look like

```
B.Load(Some_String);
...
B.Get(A_Character);
```

The rules regarding protected objects ensure that several tasks cannot be executing subprograms Load and Get simultaneously (this is implemented by the use of so-called locks which are not accessible to the programmer). This formulation therefore prevents disastrous interference between several clients but nevertheless it does not prevent a call of Load from overwriting unread data. As before we could insert tests and raise an exception. But the proper approach is to cause the tasks to wait if circumstances are not appropriate. This can be done through the use of entries and barriers. The protected type might then be

```
protected type Buffer(Max: Integer) is
   entry Load(S: in String);
   entry Get(C: out Character);
private
   Data: String(1 .. Max);
   Start: Integer := 1;
   Finish: Integer := 0;
end Buffer;
protected body Buffer is
   entry Load(S: in String) when Start > Finish is
   begin
      Start := 1;
      Finish := S'Length;
      Data(Start .. Finish) := S;
   end Load;
```

```
    entry Get(C: out Character) when Start <= Finish is
    begin
      C := Data(Start);
      Start := Start + 1;
    end Get;
  end Buffer;
```

In this formulation, the procedures are replaced by entries and each entry body has a barrier condition. A task calling an entry is queued if the barrier is false and only allowed to proceed when the barrier becomes true. This construction is very efficient because task switching is minimized.

Tasking is discussed in detail in Chapter 20.

4 Programs and Libraries

In this final introductory chapter we consider the important topic of putting together a complete program. Such a program will inevitably use predefined material such as that for input and output and we therefore also briefly survey the structure and contents of the extensive predefined library.

4.1 The hierarchical library

A complete program is put together out of various separately compiled units. In developing a very large program it is inevitable that it will be conceived as a number of subsystems themselves each composed out of a number of separately compiled units.

We see at once the risk of name clashes between the various parts of the total system. It would be very easy for the designers of different parts to reuse popular package names such as Error_Messages or Debug_Info and so on.

In order to overcome this and other related problems, Ada has a hierarchical naming scheme at the library level. Thus a package Parent may have a child package with the name Parent.Child.

We immediately see that if our total system breaks down into a number of major parts such as acquisition, analysis and report then name clashes will be avoided if it is mapped into three corresponding library packages plus appropriate child units. There is then no risk of a clash between the names Analysis.Debug_Info and Report.Debug_Info since they are now quite distinct.

The naming is hierarchical and can continue to any depth. A good example of the use of this hierarchical naming scheme is found in the standard libraries which are provided by every implementation of Ada.

We have already mentioned the package Standard which is an intrinsic part of the language. All library units can be considered to be children of Standard and it should never (well, hardly ever) be necessary to explicitly mention Standard at all

(unless you do something crazy like redefine Integer to mean something else; the original could then still be referred to as Standard.Integer).

In order to reduce the risk of clashes with users' own names the predefined library comprises just three packages each of which has a number of children. The three packages are System, which is concerned with the control of storage and similar implementation matters; Interfaces, which is concerned with interfaces to other languages and the intrinsic hardware types; and finally Ada which contains the bulk of the predefined library.

The packages System and Interfaces are described in detail in Chapter 25. In this chapter we will briefly survey the main package Ada; a fuller description will be found in Chapters 23 and 24.

The package Ada itself (herself?) is simply

```
package Ada is
   pragma Pure(Ada);       -- as white as driven snow!
end Ada;
```

and the various predefined units are children of Ada. The pragma indicates that Ada has no variable state (this concept is important for sharing in distributed systems, a topic outside the scope of this book).

Important child packages of Ada are

Numerics – this contains the mathematical library providing the elementary functions, random number generators, facilities for complex numbers, and vector and matrix manipulation.

Characters – this contains packages for classifying and manipulating characters as well as the names of all the characters in the Latin-1 set.

Strings – this contains packages for the manipulation of strings of various kinds: fixed length, bounded and unbounded.

Containers – this contains packages for manipulating various data structures such as vectors, lists, trees, maps, and sets.

Text_IO, Sequential_IO and Direct_IO – these and other packages provide a variety of input–output facilities.

There are many other children of Ada and these will be mentioned as required.

Most library units are packages and it is easy to think that all library units must be packages; indeed only a package can have child units. But of course the main subprogram is a library subprogram and any library package can have child subprograms. A library unit can also be a generic package or subprogram and even an instantiation of a generic package or subprogram; this latter fact is often overlooked.

We have introduced the hierarchical library as simply a naming mechanism. It also has important information hiding and sharing properties which will be dealt with in detail in Chapter 13. An important example is that a child unit can access the information in the private part of its parent; but of course other units cannot see into the private part of a package. This and related facilities enable a group of units to share private information while keeping the information hidden from external clients.

It should also be noted that a child package does not need to have a with clause or use clause for its parent; this emphasizes the close relationship of the hierarchical structure and parallels the fact that we never need a with clause for Standard because all library units are children of Standard.

4.2 Input–output

The Ada language is defined in such a way that all input and output is performed in terms of other language features. There are no special intrinsic features just for input and output. In fact input–output is just a service required by a program and so is provided by one or more Ada packages. This approach runs the attendant risk that different implementations will provide different packages and program portability will be compromised. In order to avoid this, the *ARM* describes certain standard packages that will be available in all implementations. Other, more elaborate, packages may be appropriate to special circumstances and the language does not prevent this. Indeed very simple packages such as our purely illustrative Simple_IO may also be appropriate. Full consideration of input and output is deferred until Chapter 23. However, we will now briefly describe how to use some of the features so that the reader will be able to run some simple exercises. We will restrict ourselves to the input and output of simple text.

Text input–output is performed through the use of the standard package Ada.Text_IO. Unless we specify otherwise, all communication will be through two standard files, one for input and one for output, and we will assume that (as is likely for most implementations) these are such that input is from the keyboard and output is to the screen. The full details of Ada.Text_IO cannot be described here but if we restrict ourselves to just a few useful facilities it looks a bit like

```
with Ada.IO_Exceptions;
package Ada.Text_IO is
   type Count is ...          -- an integer type
   ...
   procedure New_Line(Spacing: in Count := 1);
   procedure Set_Col(To: in Count);
   function Col return Count;
   ...
   procedure Get(Item: out Character);
   procedure Get(Item: out String);
   procedure Put(Item: in Character);
   procedure Put(Item: in String);
   procedure Put_Line(Item: in String);
   ...
   ...                    -- package Float_IO as outlined in Section 3.4
   ...                    -- plus a similar package Integer_IO and so on
   ...
end Ada.Text_IO;
```

This package commences with a with clause for Ada.IO_Exceptions. This further package contains the declaration of a number of different exceptions relating to a variety of things which can go wrong with input–output. For toy programs the

most likely to arise is Data_Error which would occur for example if we tried to read in a number from the keyboard but then accidentally typed in something which was not a number at all or was in the wrong format.

Note the outline declaration of the type Count. This is an integer type having similar properties to the type Integer and almost inevitably with the same implementation (just as the type My_Integer might be based on Integer). The parameter of New_Line is of the type Count rather than plain Integer, although since the parameter will often be a literal such as 2 (or be omitted so that the default of 1 applies) this will not be particularly evident.

The procedure Set_Col and function Col are useful for tabulation. The character positions along a line of output are numbered starting at 1. So if we write (assuming **use** Ada.Text_IO;)

```
Set_Col(10);
```

then the next character output will go at position 10. A call of New_Line naturally sets the current position to 1 so that output commences at the beginning of the line. The function Col returns the current position and so

```
Set_Col(Col + 10);
```

will move the position on by 10 and thereby leave 10 spaces. Note that Col is an example of a function that has no parameters.

A single character can be output by for example

```
Put('A');
```

and a string of characters by

```
Put("This Is a string of characters");
```

There is also a useful procedure Put_Line which takes a string as parameter and has the effect of Put with the string followed by a call of New_Line.

A value of the type My_Float can be output in various formats. But first we have to instantiate the package Float_IO mentioned in Section 3.4 and which is declared inside Ada.Text_IO. Having done that we can call Put with a single parameter, the value of type My_Float to be output, in which case a standard default format is used – or we can add further parameters controlling the format. This is best illustrated by a few examples and we will suppose that the type My_Float was declared to have 7 decimal digits as in Section 2.3.

If we do not supply any format parameters then an exponent notation is used with 7 significant digits, 1 before the point and 6 after (the 7 matches the precision given in the declaration of My_Float). There is also a leading space or minus sign. The exponent consists of the letter E followed by the exponent sign (+ or –) and then a two-digit decimal exponent.

We can override the default by providing three further parameters which give respectively, the number of characters before the point, the number of characters after the point, and the number of characters after E. However, there is still always only one digit before the point and at least one after it (if we foolishly ask for zero after we get one). If we do not want exponent notation then we simply specify the last parameter as zero and we then get normal decimal notation.

The effect is shown by the following statements with the output given as a comment. For clarity the output is surrounded by quotes and s designates a space; in reality there are no quotes and spaces are spaces.

```
Put(12.34);              -- "s1.234000E+01"
Put(12.34, 3, 4, 2);     -- "ss1.2340E+1"
Put(12.34, 3, 4, 0);     -- "s12.3400"
```

The output of values of integer types follows a similar pattern. In this case we similarly instantiate the generic package Integer_IO inside Ada.Text_IO which applies to all integer types with the particular type such as My_Integer. We can then call Put with a single parameter, the value of type My_Integer, in which case a standard default field is used, or we can add a further parameter specifying the field. The default field is the smallest that will accommodate all values of the type My_Integer allowing for a leading minus sign. Thus for the range of My_Integer, the default field is 8. It should be noticed that if we specify a field which is too small then it is expanded as necessary. So

```
Put(123);                -- "sssss123"
Put(–123);               -- "ssss–123"
Put(123, 4);             -- "s123"
Put(123, 0);             -- "123"
```

Simple text input is similarly performed by a call of Get with a parameter that must be a variable of the appropriate type.

A call of Get with a floating or integer parameter will expect us to type in an appropriate number at the keyboard; this may have a decimal point only if the parameter is of a floating type. It should also be noted that leading blanks (spaces) and newlines are skipped. A call of Get with a parameter of type Character will read the very next character, and this can be neatly used for controlling the flow of an interactive program, thus

```
C: Character;
...
Put("Do you want to stop?  Answer Y if so. ");
Get(C);
if C = 'Y' then
   ...
```

For simple programs that do not have to be portable, the instantiations can be avoided if we just use the predefined types Integer and Float since the predefined library contains nongeneric versions with names Ada.Integer_Text_IO and Ada.Float_Text_IO respectively. So all we need write is

```
use Ada.Integer_Text_IO, Ada.Float_Text_IO;
```

and then we can call Put and Get without more ado.

An even simpler approach which is useful for testing is to use the attribute Image described in Appendix 1, thus to output the value of N of type Integer we simply write

```
Put(Integer'Image(N));
```

That concludes a brief introduction to input–output which has been of a rather cookbook nature. Hopefully, it has provided enough to enable the reader to drive some trial examples as well as giving some further flavour to the nature of Ada.

Exercise 4.2

1 Which of the calls of Put discussed above would produce different results if we had used the type Integer rather than My_Integer? Consider both a 16-bit and a 32-bit implementation of Integer.

4.3 Numeric library

The numeric library comprises the package Ada.Numerics plus a number of child packages. The package Ada.Numerics is as follows

```
package Ada.Numerics is
   pragma Pure(Numerics);
   Argument_Error: exception;
   Pi: constant := 3.14159_26535_89793_23846_26433_83279_
                                                50288_41971_69399_37511;
   π: constant := Pi;
   e: constant := 2.71828_18284_59045_23536_02874_71352_
                                                66249_77572_47093_69996;
end Ada.Numerics;
```

This contains the exception Argument_Error which is raised if something is wrong with the argument of a numeric function (such as attempting to take the square root of a negative number) and the useful constants Pi and e. These constants (strictly known as named numbers as explained in Chapter 6) are given to 50 decimal places. Technically of course the literals must all be on a single line but the page width of this book (unlike the *ARM*) cannot cope. Note also that we can use the Greek letter π instead of Pi. This an illustration of the fact that we can use other alphabets such as Greek and Cyrillic for identifiers if we wish.

One child package of Ada.Numerics provides the familiar elementary functions such as Sqrt and is another illustration of the use of the generic mechanism. Its specification is

```
generic
   type Float_Type is digits <>;
package Ada.Numerics.Generic_Elementary_Functions is
   function Sqrt(X: Float_Type'Base) return Float_Type'Base;
   ...      -- and so on
end;
```

Again there is a single generic parameter giving the floating type. In order to call the function Sqrt we must first instantiate the generic package much as we did for Float_IO, thus (assuming appropriate with and use clauses)

```
package My_Elementary_Functions is
      new Generic_Elementary_Functions(My_Float);
use My_Elementary_Functions;
```

and we can then write a call of Sqrt directly.

The rather long name My_Elementary_Functions follows the recommended practice in the *ARM*. In fact it should be noted that there is a nongeneric version for the type Float with the name Ada.Numerics.Elementary_Functions for the convenience of those using the predefined type Float.

A little point to note is that the parameter and result of Sqrt is written as Float_Type'Base; the reason for this is explained in Chapter 17 when we look at numeric types in more detail.

We emphasize that the exception Ada.Numerics.Argument_Error is raised if the parameter of a function such as Sqrt is unacceptable. This contrasts with our hypothetical function Sqrt introduced earlier which we assumed raised the predefined exception Constraint_Error when given a negative parameter. When we deal with exceptions in detail in Chapter 15 we shall see that it is generally better to declare and raise our own exceptions rather than use the predefined ones.

Two other child packages are those for the generation of random numbers. Each contains a function Random. One returns a value of the type Float within the range zero to one and the other returns a random value of a discrete type (an integer type or an enumeration type). We will look briefly at the latter and refer the reader to Chapter 23 for full details of both. The specification is as follows

```
generic
   type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random is
   type Generator is limited private;
   function Random(Gen: Generator) return Result_Subtype;
   ...      -- plus other facilities
end Ada.Numerics.Discrete_Random;
```

This introduces a number of new points. The most important is the form of the generic formal parameter which indicates that the actual type must be a discrete type. The pattern echoes that of an enumeration type in much the same way as that for the floating generic parameter in Generic_Elementary_Functions echoed the declaration of a floating type. Thus we see that the discrete types are another example of a class of types as discussed in Section 3.4.

A small point is that the type Generator is declared as limited. This simply means that assignment is not available for the type (or at least not for the partial view as seen by the client).

The random number generator is used as in the following fragment which illustrates the simulation of tosses of a coin

```
use Ada.Numerics;
type Coin is (Heads, Tails);
package Random_Coin is new Discrete_Random(Coin);
use Random_Coin;
G: Generator;
C: Coin;
```

```
...
loop
   C := Random(G);
   ...
end loop;
...
```

Having declared the type Coin we then instantiate the generic package. We then declare a generator and use it as the parameter of successive calls of Random. The generator technique enables us to declare several generators and thus run several independent random sequences at the same time.

This concludes our brief survey of the standard numeric packages; further details will be found in Chapter 23.

4.4 Running a program

We are now in a position to put together a complete program using the proper input–output facilities. As a first example the following program outputs the multiplication tables up to ten. Each number is printed in a column five characters wide.

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Multiplication_Tables is
begin
   for Row in 1 .. 10 loop
      for Column in 1 .. 10 loop
         Put(Row * Column, 5);
      end loop;
      New_Line;
   end loop;
end Multiplication_Tables;
```

As a further example we will rewrite the procedure Print_Roots of Section 2.2 and also use the standard mathematical library. For simplicity we will first use the predefined type Float. The program becomes

```
with Ada.Text_IO;
with Ada.Float_Text_IO;
with Ada.Numerics.Elementary_Functions;
procedure Print_Roots is
   use Ada.Text_IO;
   use Ada.Float_Text_IO;
   use Ada.Numerics.Elementary_Functions;

   X: Float;
begin
   Put("Roots of various numbers");

   ...   -- and so on as before
end Print_Roots;
```

Note that we can put the use clauses inside the procedure Print_Roots as shown or we can place them immediately after the with clauses as in the example of the multiplication tables.

If we want to write a portable version then the general approach would be

```
with Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;
procedure Print_Roots is
   type My_Float is digits 7;
   package My_Float_IO is new Ada.Text_IO.Float_IO(My_Float);
   use My_Float_IO;
   package My_Elementary_Functions is
         new Ada.Numerics.Generic_Elementary_Functions(My_Float);
   use My_Elementary_Functions;

   X: My_Float;
begin
   Put("Roots of various numbers");

   ...   -- and so on as before
end Print_Roots;
```

To have to write all that introductory stuff each time is rather a burden, so we will put it in a standard package of our own and then compile it once so that it is permanently available and can be accessed without more ado. We include the type My_Integer as well and write

```
with Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;
package Etc is
   type My_Float is digits 7;
   type My_Integer is range -1000_000 .. +1000_000;

   package My_Float_IO is new Ada.Text_IO.Float_IO(My_Float);
   package My_Integer_IO is new Ada.Text_IO.Integer_IO(My_Integer);

   package My_Elementary_Functions is
         new Ada.Numerics.Generic_Elementary_Functions(My_Float);
end Etc;
```

Having compiled Etc a typical program could look like

```
with Ada.Text_IO, Etc;
use Ada.Text_IO, Etc;
procedure Program is
   use My_Float_IO, My_Integer_IO, My_Elementary_Functions;
   ...
end Program;
```

where we would naturally only supply the internal use clauses for those packages we were actually using in the particular program.

(The author had great difficulty in identifying an appropriate and short name for the package Etc and hopes that he is forgiven for the pun on etcetera.)

An alternative approach, rather than declaring everything in the one package Etc, is to first compile a tiny package just containing the types My_Float and My_Integer and then to compile the various instantiations as individual library packages (remember from Section 4.1 that a library unit can be just an instantiation).

We might then have the following four library units

```
package My_Numerics is
   type My_Float is digits 7;
   type My_Integer is range –1000_000 .. +1000_000;
end My_Numerics;
```

```
with My_Numerics;  use My_Numerics;
with Ada.Text_IO;
package My_Float_IO is new Ada.Text_IO.Float_IO(My_Float);
```

```
with My_Numerics;  use My_Numerics;
with Ada.Text_IO;
package My_Integer_IO is new Ada.Text_IO.Integer_IO(My_Integer);
```

```
with My_Numerics;  use My_Numerics;
with Ada.Numerics.Generic_Elementary_Functions;
package My_Elementary_Functions is
      new Ada.Numerics.Generic_Elementary_Functions(My_Float);
```

With this approach we only need to include (via with clauses) the particular packages as required and our program is thus likely to be smaller if we do not need them all. We could even arrange the packages as an appropriate hierarchy.

The reader should now be in a position to write complete simple programs. The examples in this book have mostly been written as fragments rather than complete programs for two reasons; one is that complete programs take up a lot of space (and can be repetitive) and the other is that Ada is really all about software components anyway. On the other hand complete programs show how the components fit together and perhaps also act as templates for the reader to follow. Accordingly this book includes a small number of complete programs the first of which is immediately after this chapter.

One important matter remains to be addressed and that is how to compile and build a complete program. A major benefit of Ada is that consistency is maintained between separately compiled units so that the integrity of strong typing is preserved across compilation unit boundaries. It is therefore illegal to build a program out of inconsistent units; the means whereby this is prevented will depend upon the implementation.

A related issue is the order in which units are compiled. This is dictated by the idea of dependency. There are three main causes of dependency

- a body depends upon the corresponding specification,
- a child depends upon the specification of its parent,
- a unit depends upon the specifications of those it mentions in a with clause.

The key rule is that a unit can only be compiled if all those on which it depends are present in the library environment. (A discussion of what is meant by the program library environment will be found in Chapter 13.)

Thus in the example above, the package My_Float_IO cannot be compiled until after the package My_Numerics has been written and entered into the library. But there is no dependency between My_Float_IO and My_Integer_IO.

An important consequence of the rules is that the user A of a package B is quite independent of the body of B. The body of B can be changed and recompiled without having to recompile the user A. But of course the body of B cannot be compiled without the specification of B being present.

Many compilation systems permit several library units to be compiled together with the text all in a single file but of course their order within the file must be such that each unit follows all those on which it depends. A common special case is that a package specification and body can be compiled together provided the text of the specification precedes that of the body.

Complete simple programs might be presented in a single file. Thus if we wrote a program using the package Shapes of Exercise 3.2(**1**) and all of the text were contained in one file then the outline structure of the file might be

```
package Objects is ...        -- spec and body of Objects

with Objects; use Objects;
package Shapes is ...         -- spec and body of Shapes

with Shapes; use Shapes;
procedure Main is ...         -- the main subprogram
```

The dependency rules also need to be considered if a unit is modified and then recompiled; typically all those units which depend on it (directly or indirectly) will also have to be recompiled in order to preserve consistency of the total program (a good system will do such recompilations automatically).

Unfortunately it is not possible to explain how to manipulate the library, call the Ada compiler and then build a complete program or indeed how to call our Ada program because this depends upon the implementation and so we must leave the reader to find out how to do these last vital steps from the documentation for the implementation concerned.

This brings us to the end of our brief survey of the main features of Ada. We have in fact encountered most of the main concepts although very skimpily in some cases. We have for example said very little about the use of subtypes which impose constraints on types.

Hopefully the reader will have grasped the key structural concepts and especially the forms of abstraction discussed in Chapter 3. The remainder of this book takes us through various topics in considerable detail starting with the small-scale aspects. In looking at these aspects we should not lose sight of the big picture to which we will return in Part 3. Moreover, we will generally use the types Integer and Float for simplicity but the reader will no doubt remember that they are not portable.

Exercise 4.4

1 Write a program to output a table of square roots of numbers from 1 up to some limit specified by the user in response to a suitable question. Print the numbers as integers and the square roots to 6 decimal places in two columns. Use

Set_Col to set the second column position so that the program can be easily modified. Note that a value N of type Integer can be converted to the corresponding My_Float value by writing My_Float(N). Use the package Etc.

2 Declare the packages My_Float_IO, My_Integer_IO and My_Elementary_Functions as child packages of My_Numerics. Avoid unnecessary with and use clauses.

3 Write a program to read in the lengths of the three sides of a triangle and then print out its area. Use the package Shapes of Exercise 3.2(**1**).

4 Write a program to generate 100 random days of the week and count how many of them are Sundays. Output the answer in a suitable format.

5 Write a program to print out Pascal's triangle described in Section 2.4. Read in the value of an Integer variable Size which gives the final row number. Ensure that the triangle is properly aligned and has a suitably aligned caption. Print each number in a field of width 4; this will neatly accommodate values of Size up to 12. Note that a value N of type Integer can be converted to type Count by writing Count(N).

Program 1

Magic Moments

This first example of a complete program pulls together various aspects of the type Object and its descendants which were discussed in Chapter 3. This opening description is followed by the text of the program and then some notes on specific points.

The program is in two phases. It first reads in the dimensions of a number of objects and then computes and prints out a table of their properties.

The root package Geometry contains the abstract type Object together with various abstract operations. The function MI is the moment of inertia of the object about its centre (see Exercise 3.3(**5**)). The function Name returns a string describing the type of the object. Each of the concrete types is then declared in its own child package.

The child package Geometry.Circles declares the type Circle which is derived from the abstract type Object and has one additional component, Radius. It also declares concrete functions Area, MI and Name which implement the corresponding abstract operations of the root type Object.

The child packages Geometry.Points, Geometry.Triangles, and Geometry.Squares then similarly declare types Point, Triangle, and Square with appropriate concrete functions.

There are also a number of other child packages containing related material. The package Geometry.Magic contains the class wide functions Moment and MO; Moment gives the vertical moment about the origin whereas MO gives the moment of inertia about the origin. The package Geometry.Lists contains entities for defining and manipulating lists of objects. The package Geometry.IO contains functions for reading the properties of the various types of objects; each such function returns a pointer to a newly created object.

There are then two library subprograms to do the two major phases of activities, namely Build_List and Tabulate_Properties. Finally the main subprogram Magic_Moments essentially just calls these two other library subprograms.

```
package Geometry is
   type Object is abstract tagged
      record
         X_Coord: Float;
         Y_Coord: Float;
      end record;
   function Distance(O: Object) return Float;
   function Area(O: Object) return Float
                                        is abstract;
   function MI(O: Object) return Float is abstract;
   function Name(O: Object) return String
                                        is abstract;
end;
```

```
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;
package body Geometry is
   function Distance(O: Object) return Float is
   begin
      return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
   end Distance;
end Geometry;
```

```
package Geometry.Magic is
   function Moment(OC: Object'Class)
                                        return Float;
   function MO(OC: Object'Class) return Float;
end;
```

```
package body Geometry.Magic is
   function Moment(OC: Object'Class)
                                        return Float is
   begin
      return OC.X_Coord * OC.Area;
   end Moment;
   function MO(OC: Object'Class) return Float is
   begin
      return OC.MI + OC.Area * OC.Distance**2;
   end MO;
end Geometry.Magic;
```

```
package Geometry.Circles is
   type Circle is new Object with
      record
         Radius: Float;
      end record;
   function Area(C: Circle) return Float;
   function MI(C: Circle) return Float;
   function Name(C: Circle) return String;
end;
with Ada.Numerics;
package body Geometry.Circles is
   function Area(C: Circle) return Float is
   begin
      return Ada.Numerics.Pi * C.Radius**2;
   end Area;
   function MI(C: Circle) return Float is
   begin
      return 0.5 * C.Area * C.Radius**2;
   end MI;
   function Name(C: Circle) return String is
   begin
      return "Circle";
   end Name;

end Geometry.Circles;
```

```
package Geometry.Points is
   type Point is new Object with null record;
   function Area(P: Point) return Float;
   function MI(P: Point) return Float;
   function Name(P: Point) return String;
end;
package body Geometry.Points is
   function Area(P: Point) return Float is
   begin
      return 0.0;
   end Area;
   function MI(P: Point) return Float is
   begin
      return 0.0;
   end MI;
   function Name(P: Point) return String is
   begin
      return "Point";
   end Name;

end Geometry.Points;
```

```
package Geometry.Triangles is
   type Triangle is new Object with
      record
         A, B, C: Float;    -- lengths of sides
      end record;
   function Area(T: Triangle) return Float;
   function MI(T: Triangle) return Float;
   function Name(T: Triangle) return String;
end;
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;
package body Geometry.Triangles is
   function Area(T: Triangle) return Float is
      S: constant Float := 0.5 * (T.A + T.B + T.C);
   begin
      return Sqrt(S * (S – T.A) * (S – T.B) * (S – T.C));
   end Area;
   function MI(T: Triangle) return Float is
   begin
      return T.Area * (T.A**2 + T.B**2 + T.C**2)
                                          / 36.0;
   end MI;
   function Name(T: Triangle) return String is
   begin
      return "Triangle";
   end Name;

end Geometry.Triangles;
```

```
package Geometry.Squares is
   type Square is new Object with
      record
         Side: Float;
      end record;
   function Area(S: Square) return Float;
   function MI(S: Square) return Float;
   function Name(S: Square) return String;
end;
package body Geometry.Squares is
   function Area(S: Square) return Float is
   begin
      return S.Side**2;
   end Area;
   function MI(S: Square) return Float is
   begin
      return S.Area * S.Side**2 / 6.0;
   end MI;
   function Name(S: Square) return String is
   begin
      return "Square";
   end Name;

end Geometry.Squares;
```

```
package Geometry.Lists is
   type Pointer is access Object'Class;

   type Cell is
      record
         Next: access Cell;
         Element: Pointer;
      end record;

   type List is access Cell;

   procedure Add_To_List(The_List: in out List;
                         Obj_Ptr: in Pointer);
end;

package body Geometry.Lists is

   procedure Add_To_List(The_List: in out List;
                         Obj_Ptr: in Pointer) is
   begin
      The_List := new Cell'(The_List, Obj_Ptr);
   end Add_To_List;

end Geometry.Lists;
```

```
with Geometry.Lists;
package Geometry.IO is
   function Get_Circle return Lists.Pointer;
   function Get_Triangle return Lists.Pointer;
   function Get_Square return Lists.Pointer;
end;

with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Float_Text_IO;  use Ada.Float_Text_IO;
with Geometry.Circles;
with Geometry.Triangles;
with Geometry.Squares;
package body Geometry.IO is

   function Get_Circle return Lists.Pointer is
      use Circles;
      X_Coord: Float;
      Y_Coord: Float;
      Radius: Float;
   begin
      Get(X_Coord);
      Get(Y_Coord);
      Get(Radius);
      return new Circle'(X_Coord, Y_Coord,
                                          Radius);
   end Get_Circle;

   function Get_Triangle return Lists.Pointer is
      use Triangles;
      X_Coord: Float;
      Y_Coord: Float;
      A, B, C: Float;
   begin
      Get(X_Coord);
      Get(Y_Coord);
      loop
         Get(A);  Get(B);  Get(C);
         -- check to ensure a valid triangle
         exit when A < B+C and B < C+A and
                                          C < A+B;
         Put("Sorry, not a triangle, " &
             "enter sides again please");
         New_Line;
      end loop;

      return new Triangle'(X_Coord, Y_Coord,
                                          A, B, C);
   end Get_Triangle;

   function Get_Square return Lists.Pointer is
      use Squares;
      X_Coord: Float;
      Y_Coord: Float;
      Side: Float;
   begin
      Get(X_Coord);
      Get(Y_Coord);
      Get(Side);
      return new Square'(X_Coord, Y_Coord,
                                          Side);
   end Get_Square;

end Geometry.IO;
```

```
with Geometry.Lists;  use Geometry;
with Geometry.IO;  use Geometry.IO;
with Ada.Text_IO;  use Ada.Text_IO;
procedure Build_List(The_List: in out Lists.List) is
   Code_Letter: Character;
   Object_Ptr: Lists.Pointer;
begin
   loop

      loop              -- loop to skip leading spaces
         Get(Code_Letter);
         exit when Code_Letter /= ' ';
      end loop;

      case Code_Letter is
         when 'C' | 'c' =>         -- expect a circle
            Object_Ptr := Get_Circle;
         when 'T' | 't' =>         -- expect a triangle
            Object_Ptr := Get_Triangle;
         when 'S' | 's' =>         -- expect a square
            Object_Ptr := Get_Square;
         when others =>
            exit;
      end case;

      Lists.Add_To_List(The_List, Object_Ptr);
   end loop;
end Build_List;
```

```
with Geometry.Lists;  use Geometry.Lists;
with Geometry.Magic;  use Geometry.Magic;
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Float_Text_IO;  use Ada.Float_Text_IO;
procedure Tabulate_Properties(The_List: List) is
   Local: access Cell := The_List;
   This_One: Pointer;
begin
   New_Line;
   Put("           X       Y       Area     " &
       "     MI              MO          Moment");
   New_Line;
   while Local /= null loop
      This_One := Local.Element;
      New_Line;
      Put(This_One.Name);  Set_Col(10);
      Put(This_One.X_Coord, 4, 2, 0);  Put('  ');
      Put(This_One.Y_Coord, 4, 2, 0);  Put('  ');
      Put(This_One.Area, 6, 2, 0);  Put('  ');
      Put(This_One.MI, 6, 2, 0);  Put('  ');
      Put(MO(This_One.all), 6, 2, 0);  Put('  ');
      Put(Moment(This_One.all), 6, 2, 0);
      Local := Local.Next;
   end loop;
end Tabulate_Properties;
```

```
with Build_List;
with Tabulate_Properties;
with Geometry.Lists;  use Geometry.Lists;
with Ada.Text_IO;
with Ada.Float_Text_IO;
use Ada;
procedure Magic_Moments is
   The_List: List := null;
begin
   Text_IO.Put("Welcome to Magic Moments");
   Text_IO.New_Line(2);
   Text_IO.Put("Enter C, T or S followed by " &
                  "coords and dimensions");
   Text_IO.New_Line;
   Text_IO.Put("Terminate list with any other letter");
   Text_IO.New_Line(2);
   Build_List(The_List);
   Tabulate_Properties(The_List);
   Text_IO.New_Line(2);
   Text_IO.Put("Finished");
   Text_IO.New_Line;
   Text_IO.Skip_Line(2);
end Magic_Moments;
```

The overall structure is always a matter for debate. It could be argued that the main subprogram and the two other library subprograms should all be children of Geometry. Doing so would reduce the number of with clauses required.

The application of use clauses is also a matter of eternal debate. In some units they have been applied liberally whereas in others only the top level (Ada or Geometry) has been given in a use clause. This has been done largely to illustrate the possibilities. Remember that a child unit never needs a with or use clause for its parent. But a child does need a with clause in order to access a sibling.

There are a number of different approaches to reading in the details of the objects in Build_List and that shown is very straightforward. Note the use of the case statement whose structure and purpose is hopefully fairly evident. Each object is described by a letter giving its type followed by the required dimensions; any other letter terminates the list of objects. There is also a certain amount of repetition – for example we should perhaps read the common components X_Coord and Y_Coord for all the types in one place only. The function Get_Triangle checks that the three sides do form a triangle in accordance with Euclid, Book I, Proposition 20: '*Παντὸς τριγώνου αἱ δύο πλευραὶ τῆς λοιπῆς μείζονές εἰσι πάντη μεταλαμβανόμενει.*' – 'In any triangle two sides taken together in any manner are greater than the remaining one.' However, there are no checks that the dimensions of any of the objects are actually positive.

The procedure Tabulate_Properties outputs the various properties in a table. The while form of loop causes the loop to stop at the end of the list. The various values are output using a fixed format with a space character between them so that overlength numbers do not run together. Note also that the function Name returns a string whose length depends upon the specific type.

An interesting point is that Tabulate_Properties only interrogates those aspects that are common to all objects (that is, common components such as X_Coord or common operations such as Area). We could add another dispatching operation to the root package Geometry that printed the specific properties (such as the radius in the case of a circle) either directly or perhaps by calling subprograms in Geometry.IO.

This leads to a final point. Dispatching on a class wide value such as one of type Pointer is possible because the code knows the specific type of the object being pointed to and can choose the specific operation accordingly. This means that dispatching on output is feasible. But dispatching on input is difficult because until we have read the object we do not know its type. This is why we declared distinct subprograms Get_Circle, Get_Triangle and so on for reading each type. However, we can use an object constructor and then dispatch as explained in Section 21.6.

Part 2

Algorithmic Aspects

This second part covers the small-scale algorithmic features of the language in detail. These correspond to the areas covered by simple languages such as Pascal and C although Ada has much richer facilities in these areas.

Chapter 5 deals with the lexical detail which needs to be described but can perhaps be skimmed on a first reading and just referred to when required. It also briefly introduces aspect specifications which are an important new feature of Ada 2012.

Chapter 6 is where the story really begins and covers the type model and illustrates that model by introducing most of the scalar types. Chapter 7 then discusses the control structures which are very straightforward. Chapter 8 covers arrays in full but only the simplest forms of records; it is also convenient to introduce characters and strings in this chapter since strings in Ada are treated as arrays of characters. Chapter 8 is quite long compared with a corresponding discussion on Pascal or C largely because of the named notation for

array aggregates which is an important feature for writing readable programs.

Chapter 9 discusses the more elaborate forms of expressions introduced in Ada 2012, namely if expressions, case expressions and quantified expressions.

Subprograms are discussed in Chapter 10 and at this point we can write serious lumps of program. Again the named notation enriches the discussion and includes the mechanism for default parameters.

Chapter 11 contains a discussion on access types which correspond to pointers in Pascal and C. Although the concept of an access type is easy to understand, nevertheless the rules regarding accessibility might be found difficult on a first reading. These rules give much greater flexibility than Pascal whilst being carefully designed to prevent dangling references which can so easily cause a program to crash in C.

This second part concludes with a complete program which illustrates various algorithmic features and especially the use of access types.

Those familiar with Ada 2005 will find little changed in Chapters 5 to 8 apart from the introduction of aspect specifications at the end of Chapter 5. But Chapter 9 on expression structures is completely new. Chapter 10 has new material concerning expression functions, parameters of functions and new aliasing rules. Chapter 11 contains a number of changes regarding anonymous access types.

5 Lexical Style

In the previous chapters, we introduced some concepts of Ada and illustrated the general appearance of Ada programs with some simple examples. However, we have so far only talked around the subject. In this chapter we get down to serious detail.

Regrettably it seems best to start with some rather unexciting but essential material – the detailed construction of things such as identifiers and numbers which make up the text of a program. However, it is no good learning a human language without getting the spelling sorted out. And as far as programming languages are concerned, compilers are usually very unforgiving regarding such corresponding and apparently trivial matters.

We also take the opportunity to introduce the notation used to describe the syntax of Ada. In general we will not use this syntax notation to introduce concepts but, in some cases, it is the easiest way to do so. Moreover, if the reader wishes to consult the *ARM* then knowledge of the syntax notation is necessary. For completeness and easy reference the full syntax is given in Appendix 3.

5.1 Syntax notation

The syntax of Ada is described using a variant of Backus–Naur Form (BNF). In this notation, syntactic categories are represented by lower case names; some of these contain embedded underlines to increase readability. A category is defined in terms of other categories by a sort of equation known as a production. Some categories are atomic and cannot be decomposed further – these are known as terminal symbols. A production consists of the name being defined followed by the special symbol ::= and its defining sequence.

Other symbols used are

[ ] square brackets enclose optional items,

{ } braces enclose optional items which may be omitted, appear once or be repeated many times,

| a vertical bar separates alternatives.

Sometimes the name of a category is prefixed by a word in italics. In such cases the prefix aims to convey some semantic information and can be treated as a form of comment as far as the context free syntax is concerned. Where relevant, a production is presented in a form that shows the recommended layout.

5.2 Lexical elements

The international standard describes the language in terms of the ISO 32-bit 10646:2003 set. This is very broad and covers the Latin characters including accented forms plus the Greek and Cyrillic alphabets and various ideographs as well. It is essentially equivalent to Unicode 4.0. The general intent is that Ada has a multicultural approach appropriate to the 21st century. In this book we will use the Latin and Greek alphabets.

A line of Ada text can be thought of as a sequence of groups of characters known as lexical elements. We have already met several forms of lexical elements in previous chapters. Consider for example

```
Age := 21;      -- John's age
```

This consists of five such elements

- the identifier Age
- the compound delimiter :=
- the numeric literal 21
- the single delimiter ;
- the comment -- *John's age*

Other classes of lexical element are strings and character literals; they are dealt with in Chapter 9.

Individual lexical elements may not be split by spaces but otherwise spaces may be inserted freely in order to improve the appearance of the program. A most important example of this is the use of indentation to reveal the overall structure. Naturally enough a lexical element must fit on one line.

The *ARM* does not prescribe how one proceeds from one line of text to the next. This need not concern us. We can just imagine that we type our Ada program as a series of lines using whatever mechanism the keyboard provides for starting a new line.

The delimiters are either the following single delimiters

```
& ' ( ) + - * / < = > , . : ; |
```

or the following compound delimiters

=> for aggregates, cases, etc. .. for ranges

**	exponentiation	<>	the 'box' for types and defaults
:=	assignment	/=	not equals
>=	greater than or equals	<=	less than or equals
<<	label bracket	>>	the other label bracket

Special care should be taken that the compound delimiters do not contain spaces.

However, spaces may occur in strings and character literals where they stand for themselves, and also in comments. Note that adjacent words and numbers must be separated from each other by spaces otherwise they would be confused. Thus we must write **end loop** rather than **endloop**.

5.3 Identifiers

We mentioned above that Ada is defined in terms of the full ISO 32-bit character set. As a consequence we can write identifiers using many different alphabets. The basic rule is that an identifier consists of a letter possibly followed by one or more letters or digits with embedded isolated underlines. Either case of letter can be used. Moreover, the meaning attributed to an identifier does not depend upon the case of the letters. So identifiers which differ only in the case of corresponding letters are considered to be the same. But, on the other hand, the underline characters are considered to be significant.

In some alphabets the correspondence between upper and lower case letters is not straightforward. Thus in Greek there are two lower case letters equivalent to s (σ and ς, the latter being used only at the end of a word) but only one corresponding upper case letter Σ. These three characters are all the same so far as Ada identifiers are concerned. Accented characters such as é and ç may be used and they are of course distinct from the unaccented forms. Similarly, ligatures such as Æ and æ may also be used.

Ada does not impose any limit on the number of characters in an identifier. Moreover all are significant. There may however be a practical limit since an identifier must fit onto a single line and an implementation is likely to impose some maximum line length. However, this maximum must be at least 200 and so identifiers of up to 200 characters will always be accepted. Programmers are encouraged to use meaningful names such as Time_Of_Day rather than cryptic meaningless names such as T. Long names may seem tedious when first writing a program but in the course of its lifetime a program is read much more often than it is written and clarity aids subsequent understanding both by the original author and by others who may be called upon to maintain the program. Of course, in short mathematical or abstract subprograms, simple identifiers such as X and Y may be appropriate.

Identifiers are used to name all the various entities in a program. However, some words are reserved for special syntactic significance and may not be used as identifiers. We have already encountered several of these such as **if**, **procedure** and **end**. There are 73 reserved words; they are listed in Appendix 1. For readability they are printed in boldface in this book, but that is not important. In program text they could, like identifiers, be in either case or indeed in a mixture of cases – procedure, PROCEDURE and Procedure are all acceptable. Nevertheless, some

discipline aids understanding, and a good convention is to use lower case for the reserved words and leading capitals for all others. But this is a matter of taste.

There are minor exceptions regarding the reserved words **access**, **delta**, **digits**, **mod** and **range**. As will be seen later, they are also used as attributes Access, Delta, Digits, Mod, and Range. However, when so used they are always preceded by an apostrophe and so there is no confusion.

Some identifiers such as Integer and True have a predefined meaning from the package Standard. These are not reserved and can be reused although to do so is usually unwise since the program could become very confusing.

Exercise 5.3

1 Which of the following are not legal identifiers and why?

(a) Ada	(d) Ευρεκα	(g) X_
(b) fish&chips	(e) Time__Lag	(h) tax rate
(c) RATE–OF–FLOW	(f) 77E2	(i) goto

5.4 Numbers

Numbers (or numeric literals to use the proper jargon) take two forms according to whether they denote an integer (an exact whole number) or a real (an approximate and not usually whole number). This is a good opportunity to illustrate the use of the syntax.

```
numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [. numeral] [exponent]
numeral ::= digit {[underline] digit}
exponent ::= E [+] numeral | E –numeral
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A subtle point is that we can use Arabic and other forms of numerals in identifiers but not for numeric literals.

The important distinguishing feature of a real literal is that real literals always contain a decimal point whereas integer literals never do. Real literals can be used as values of any real type which we recall covers both floating point and fixed point types.

Ada is strict on mixing up types. It is illegal to use an integer literal where the context demands a real literal and vice versa. Thus

```
Age: Integer := 21.0;
```

and

```
Weight: Float := 150;
```

are both illegal.

The simplest form of integer literal is just a sequence of decimal digits. If the literal is very long it should be split up into groups of digits by inserting isolated underlines thus

```
123_456_789
```

In contrast to identifiers such underlines are, of course, of no significance other than to make the literal easier to read.

The simplest form of real literal is a sequence of decimal digits containing a decimal point. Note that there must be at least one digit on either side of the decimal point. Again, isolated underlines may be inserted to improve legibility provided they are not adjacent to the decimal point; thus

```
3.14159_26536
```

Unlike most languages both integer and real literals can have an exponent. This takes the form of the letter E (either case) followed by a signed or unsigned decimal integer. This exponent indicates the power of ten by which the preceding simple literal is to be multiplied. The exponent cannot be negative in the case of an integer literal – otherwise it might not be a whole number. (As a trivial point an exponent of −0 is not allowed for an integer literal but it is for a real literal.)

Thus the real literal 98.4 could be written with an exponent in any of the following ways

```
9.84E1      98.4e0      984.0e−1      0.984E+2
```

Note that 984e−1 would not be allowed.

Similarly, the integer literal 1900 could also be written as

```
19E2     190e+1     1900E+0
```

but not as 19000e−1 nor as 1900E−0 since these have negative exponents.

The exponent may itself contain underlines if it consists of two or more digits but it is unlikely that the exponent would be so large as to make this necessary. However, the exponent may not itself contain an exponent!

A final facility is the ability to express a literal in a base other than 10. This is done by enclosing the digits between # characters and preceding the result by the base. Thus

```
2#111#
```

is a based integer literal of value 4 + 2 + 1 = 7.

Any base from 2 to 16 inclusive can be used and, of course, base 10 can always be expressed explicitly. For bases above 10 the letters A to F are used to represent the extended digits 10 to 15. Thus

```
14#ABC#
```

equals $10 \times 14^2 + 11 \times 14 + 12 = 2126$.

A based literal can also have an exponent. But note carefully that the exponent gives the power of the base by which the simple literal is to be multiplied and not a

power of 10 – unless, of course, the base happens to be 10. The exponent itself, like the base, is always expressed in normal decimal notation. Thus

```
16#A#E2
```

equals $10 \times 16^2 = 2560$ and

```
2#11#E11
```

equals $3 \times 2^{11} = 6144$.

A based literal can be real. The distinguishing mark is again the point. (We can hardly say 'decimal point' if we are not using a decimal base! A better term is radix point.) So

```
2#101.11#
```

equals $4 + 1 + {}^1/_2 + {}^1/_4 = 5.75$ and

```
7#3.0#e-1
```

equals ${}^3/_7 = 0.\dot{4}2857\dot{1}$.

The reader may have felt that the possible forms of based literal are unduly elaborate. This is not really so. Based literals are useful – especially for fixed point types since they enable programmers to represent values in the form in which they think about them. Obviously bases 2, 8 and 16 will be the most useful. But the notation is applicable to any base and the compiler can compute to any base, so why not?

Finally note that a numeric literal cannot be negative. A form such as `-3` consists of a literal preceded by the unary minus operator.

Exercise 5.4

1 Which of the following are not legal literals and why? For those that are legal, state whether they are integer or real literals.

(a) `38.6`
(b) `.5`
(c) `32e2`
(d) `32e-2`
(e) `2#1011`
(f) `2.71828_18285`
(g) `12#ABC#`
(h) `E+6`
(i) `16#FfF#`
(j) `1_0#1_0#E1_0`
(k) `27.4e_2`
(l) `2#11#e-1`

2 What are the values of the following?

(a) `16#E#E1`
(b) `2#11#E11`
(c) `16#F.FF#E+2`
(d) `2#1.1111_1111_111#E11`

3 How many different ways can you express the following as an integer literal?

(a) the integer 41
(b) the integer 150

(Forget underlines, distinction between `E` and `e`, nonsignificant leading zeros and optional `+` in an exponent.)

5.5 Comments

It is important to add appropriate comments to a program to aid its understanding by someone else or yourself at a later date.

A comment in Ada is written as an arbitrary piece of text following two hyphens (or minus signs – the same thing). Thus

```
-- this is a comment
```

A comment can include any characters. So we might annotate a mathematical program with

```
-- the variable D_Alpha represents ∂α/∂t
```

A comment extends to the end of the line. The comment may be the only thing on the line or it may follow some other Ada text. A long comment needing several lines is merely written as successive comments.

```
-- this comment
-- is spread over
-- several lines.
```

It is important that the leading hyphens be adjacent and not separated by spaces. Note that in this book comments are written in italics. This is just to make it look pretty.

5.6 Pragmas and aspects

We conclude this chapter with a few remarks about pragmas even though they are hardly lexical elements in the sense of identifiers and numbers.

Pragmas originated in Ada 83 as a sort of compiler directive and as a convenient way of making a parenthetic remark to the compiler. As an example we can indicate that we wish the compiler to optimize our program with emphasis on saving space by writing

```
pragma Optimize(Space);
```

inside the region of program to which it is to apply. Alternatively we could write

```
pragma Optimize(Time);
```

which indicates that speed of execution is the primary criterion. Or even

```
pragma Optimize(Off);
```

to indicate that optional optimization should be turned off.

However, as Ada evolved, pragmas were also used to apply extra detail to many entities. As an example, we can use the pragma Inline to indicate that all calls of a certain procedure are to be expanded inline so that the code is faster although more space might be occupied. Thus given some procedure Do_It we might write

```
pragma Inline(Do_It);
```

Properties such as Inline are known as aspects. So we say that the aspect Inline is applied to the procedure Do_It by the pragma Inline. However, in Ada 2012 there is a much neater way of applying such aspects. We can write

```
procedure Do_It( ... )
  with Inline;
```

so that the aspect is given with the specification of the procedure. The structure **with** Inline is known as an aspect specification. There are important advantages to this approach one being that the aspect specification is lexically tied to the procedure specification whereas a pragma might be somewhat later in the text. Accordingly, many pragmas which are valid in Ada 2005 are considered obsolete in Ada 2012.

Generally a pragma can appear anywhere that a declaration or statement can appear and in some other contexts also. Sometimes there may be special rules regarding the position of a particular pragma. For fuller details on pragmas and aspect specifications see Appendix 1.

Checklist 5

The case of a letter is immaterial in all contexts except strings and character literals.

Underlines are significant in identifiers but not in numeric literals.

Spaces are not allowed in lexical elements, except in strings, character literals and comments.

The presence of a point distinguishes real and integer literals.

An integer may not have a negative exponent.

Numeric literals cannot be signed.

Leading zeros are allowed in all parts of a numeric literal.

Ada 2005 has three more reserved words than Ada 95, namely **interface**, **overriding** and **synchronized**.

Ada 2005 allows identifiers to use many more characters including the Greek and Cyrillic alphabets.

New in Ada 2012

Ada 2012 has one more reserved word than Ada 2005, namely **some**.

Ada 2012 has aspect specifications and many pragmas are obsolete.

6 Scalar Types

This chapter lays the foundations for the small-scale aspects of Ada. We start by considering the declaration of objects and the assignment of values to them and briefly discuss the ideas of scope and visibility. We then introduce the important concepts of type, subtype and constraints. As examples of types, the remainder of the chapter discusses the numeric types Integer and Float, enumeration types in general, the type Boolean in particular, and the operations on them.

6.1 Object declarations and assignments

Values can be stored in objects which are declared to be of a specific type. Objects are either variables, in which case their value may change (or vary) as the program executes, or they may be constants, in which case they keep their same initial value throughout their life.

A variable is introduced into the program by a declaration which consists of the name (that is, the identifier) of the variable followed by a colon and then the name of the type. This can then optionally be followed by the := symbol and an expression giving the initial value. The declaration terminates with a semicolon. Thus we might write

```
J: Integer;
P: Integer := 38;
```

The first declaration introduces the variable J of type Integer but gives it no particular initial value. The second introduces the variable P and gives it the specific initial value of 38.

We can introduce several variables at the same time in one declaration by separating them by commas thus

```
I, J, K: Integer;
P, Q, R: Integer := 38;
```

In the second case all of P, Q and R are given the initial value of 38.

If a variable is declared and not given an initial value then great care must be taken not to use the undefined value of the variable until one has been properly given to it. If a program does use the undefined value in an uninitialized variable, its behaviour will be unpredictable; the program is said to have a bounded error as described in Section 2.6.

A common way to give a value to a variable is by using an assignment statement. In this, the name of the variable is followed by := and then some expression giving the new value. The statement terminates with a semicolon. Thus

```
J := 36;
```

and

```
P := Q + R;
```

are both valid assignment statements and place new values in J and P, thereby overwriting their previous values.

Note that := can be followed by any expression provided that it produces a value of the type of the variable being assigned to. We will discuss all the rules about expressions later, but it suffices to say at this point that they can consist of variables and constants with operations such as + and round brackets (parentheses) and so on just like an ordinary mathematical expression.

There is a lot of similarity between a declaration containing an initial value and an assignment statement. Both use := before the expression and the expression can be of arbitrary complexity.

An important difference, however, is that although several variables can be declared and given the same initial value together, it is not possible for an assignment statement to give the same value to several variables. This may seem odd but in practice the need to give the same value to several variables usually only arises with initial values anyway.

We should remark at this stage that strictly speaking a multiple declaration such as

```
A, B: Integer := E;
```

is really a shorthand for

```
A: Integer := E;
B: Integer := E;
```

This means that in principle the expression E is evaluated for each variable. This is a subtle point and does not usually matter, but we will encounter some examples later where the effect is important.

A constant is declared in a similar way to a variable by inserting the reserved word **constant** after the colon. Of course, a constant must be initialized in its declaration otherwise it would be useless. Why?

An example might be

```
G: constant Float := 6.7E-11;          -- gravitational constant G
```

In the case of numeric types, and only numeric types, it is possible to omit the type from the declaration of a constant thus

```
γ: constant := 0.57721_56649;          -- Euler's constant
```

It is then technically known as a number declaration and merely provides a name for the number. The distinction between integer and real named numbers is made by the form of the initial value. In this case it is real because of the presence of the decimal point. It is usually good practice to omit the type when declaring numeric constants for reasons which will appear later (this especially applies in the case of true numeric constants such as π, but is not so clear for physical constants such as *c* and *G* whose value depends upon the units chosen). But note that the type cannot be omitted in numeric variable declarations even when an initial value is provided.

There is an important distinction between the allowed forms of initial values in constant declarations (with a type) and number declarations (without a type). In the former case the initial value may be any expression and is evaluated when the declaration is encountered at run time whereas in the latter case it must be static and so evaluated at compile time. Full details are deferred until Chapter 17.

Exercise 6.1

1 Write a declaration of a floating point variable F giving it an initial value of one.

2 Write appropriate declarations of constants Zero and One of type Float.

3 What is wrong with the following declarations and statements?

(a) `var I: Integer;`
(b) `g: constant := 981`
(c) `P, Q: constant Integer;`
(d) `P := Q := 7;`
(e) `MN: constant Integer := M*N;`
(f) `2Pi: constant := 2.0*Pi;`

6.2 Blocks and scopes

Ada carefully distinguishes between declarations which introduce new identifiers and statements which do not. It is clearly sensible that the declarations which introduce new identifiers should precede the statements that manipulate them. Accordingly, declarations and statements occur in distinct places in the program text. The simplest fragment of text which includes declarations and statements is a block.

A block commences with the reserved word **declare**, some declarations, **begin**, some statements and concludes with the reserved word **end** and the terminating semicolon. A trivial example is

```
declare
   J: Integer := 0;          -- declarations here
begin
   J := J + 1;               -- statements here
end;
```

A block is itself an example of a statement and so one of the statements in its body could be another block. This nesting of blocks can continue indefinitely.

Since a block is a statement it can be executed like any other statement. When this happens the declarations in its declarative part (the bit between **declare** and **begin**) are elaborated in order, and then the statements in the body (between **begin** and **end**) are executed in the usual way. Note the terminology: we elaborate declarations and execute statements. All that the elaboration of a declaration does is make the thing being declared come into existence and then evaluate and assign any initial value to it. At the **end** of the block all the things which were declared in the block automatically cease to exist.

The above simple example of a block is clearly rather foolish; it introduces J, adds 1 to it but then loses it before use is made of the resulting value.

Like other block structured languages, Ada has the idea of hiding. Consider

```
declare
   J, K: Integer;
begin
   ...                       -- here J is the outer one
   declare
      J: Integer;
   begin
      ...                    -- here J is the inner one
   end;
   ...                       -- here J is the outer one
end;
```

The variable J is declared in an outer block and then redeclared in an inner block. This redeclaration does not cause the outer J to cease to exist but merely makes it temporarily hidden. In the inner block J refers to the new J, but as soon as we leave the inner block, this new J ceases to exist and the outer one again becomes directly visible.

Another point to note is that the objects used in an initial value must, of course, already exist. They could be declared in the same declarative part but the declarations must precede their use. For example

```
declare
   J: Integer := 0;
   K: Integer := J;
begin
```

is allowed. This idea of elaborating declarations in order is important; the jargon is 'linear elaboration of declarations'. It follows from the more general principle that the meaning of a program should be clear by reading the text in order without having to look ahead. One exception to this general rule concerns aspect specifications where looking ahead is vital, see Section 16.1.

We distinguish the terms 'scope' and 'visibility'. The scope is the region of text where an entity has some effect. In the case of a block, the scope of an entity extends from the start of its declaration until the end of the block. We say it is visible at a given point if its name can be used to refer to it at that point. A fuller discussion is deferred until Chapter 10 when we distinguish visibility and direct visibility but the following simple rules will suffice for the moment

- an object is never visible in its own declaration,
- an object is hidden by the declaration of a new object with the same identifier from the start of the declaration.

Thus the inner declaration of J in the example of nesting could never be

```
J: Integer := J;                -- illegal
```

because we cannot refer to the inner J in its own declaration and the outer J is already hidden by the inner J.

Exercise 6.2

1 How many errors can you see in the following?

```
declare
   I: Integer := 7;
   J, K: Integer
begin
   J := I+K;
   declare
      P: Integer = I;
      I: Integer := Q;
   begin
      K := P+I;
   end;
   Put(K);          -- output value of K
end;
```

6.3 Types

The Ada Reference Manual (section 3.2) says 'A type is characterized by a set of values and a set of primitive operations ...'.

In the case of the built-in type Integer, the set of values is represented by

```
..., –3, –2, –1, 0, 1, 2, 3, ...
```

and the primitive operations include

```
+, –, * and so on.
```

With certain exceptions to be discussed later (access types, arrays, tasks and protected objects) every type has a name which is introduced in a type declaration. (The built-in types such as Integer are considered to be declared in the package Standard.) Moreover, every type declaration introduces a new type completely distinct from any other type.

The set of values belonging to two distinct types are themselves quite distinct, although in some cases the actual lexical form of the values may be identical – which one is meant at any point is determined by the context. The idea of one lexical form representing two or more different things is known as overloading.

Values of one type cannot be assigned to variables of another type. This is the fundamental rule of strong typing. Strong typing, correctly used, is an enormous aid to the rapid development of *correct* programs since it ensures that many errors are detected at compile time. (Overused, it can tie one in knots; we will discuss this thought in Chapter 27.)

A type declaration uses a somewhat different syntax from an object declaration in order to emphasize the conceptual difference. It consists of the reserved word **type**, the identifier to be associated with the type, the reserved word **is** and then the definition of the type followed by the terminating semicolon.

The package Standard contains type declarations such as

```
type Integer is ... ;
```

The type definition between **is** and ; gives in some way the set of values belonging to the type. As a concrete example consider the following

```
type Colour is (Red, Amber, Green);
```

(This is an example of an enumeration type and will be dealt with in more detail in a later section in this chapter.)

This introduces a new type called Colour. Moreover, it states that there are only three values of this type and they are denoted by the identifiers Red, Amber and Green.

Objects of this type can then be declared in the usual way

```
C: Colour;
```

An initial value can be supplied

```
C: Colour := Red;
```

or a constant can be declared

```
Default: constant Colour := Red;
```

We have stated that values of one type cannot be assigned to variables of another type. Therefore one cannot mix colours and integers and so

```
I: Integer;
C: Colour;
...
I := C;
```

is illegal. In some languages it is often necessary to implement concepts such as enumeration types by primitive types such as integers and give values such as 0, 1 and 2 to variables Red, Amber and Green. Thus in Java one could write

```
static final int Red = 0,  Amber = 1,  Green = 2;
```

and then use Red, Amber and Green as if they were literal values. Obviously the program would be easier to understand than if the code values 0, 1 and 2 had been used directly. But, on the other hand, the compiler could not detect the accidental assignment of a notional colour to a variable which was, in the mind of the programmer, just an ordinary integer. In Ada this is detected during compilation thus making a potentially tricky error quite trivial to discover.

6.4 Subtypes

We now introduce subtypes and constraints. A subtype characterizes a set of values which is just a subset of the values of some existing type. The subset is defined by means of a constraint or, in the case of access types, possibly also by a so-called null exclusion. (In Ada 2012, we can also use a subtype predicate as explained in Section 16.4.) Constraints take various forms according to the category of the type. As is usual with subsets, the subset may be the complete set. There is, however, no way of restricting the set of operations of the type. The subtype takes all the operations; subsetting applies only to the values.

(In Ada 83, the type of the subtype was known as the base type and we will sometimes use that term in an informal way. In later versions of Ada, the type name is strictly considered to denote a subtype of the type. Such a subtype is then called the first subtype. One advantage of this pedantic approach is that the *ARM* can use the term subtype rather than having to say type or subtype all the time.)

As an example suppose we wish to manipulate dates; we know that the day of the month must lie in the range 1 .. 31 so we declare a subtype thus

```
subtype Day_Number is Integer range 1 .. 31;
```

where the reserved word **range** followed by 1 .. 31 is known as a range constraint. We can then declare variables and constants using the subtype identifier in exactly the same way as a type identifier.

```
D: Day_Number;
```

We are then assured that the variable D can only be assigned integer values from 1 to 31 inclusive. The compiler will insert run-time checks if necessary to ensure that this is so; if a check fails then Constraint_Error is raised.

It is important to realize that a subtype declaration does not introduce a new distinct type. An object such as D is of type Integer, and so the following is perfectly legal from the syntactic point of view.

```
D: Day_Number;
I: Integer;
...
D := I;
```

On execution, the value of I may or may not lie in the range 1 .. 31. If it does, then all is well; if not then Constraint_Error will be raised. Assignment in the other direction

```
I := D;
```

will, of course, always work.

It is not always necessary to introduce a subtype explicitly in order to impose a constraint. We could equally have written

```
D: Integer range 1 .. 31;
```

Furthermore a subtype need not impose a constraint. It is perfectly legal to write

```
subtype Day_Number is Integer;
```

although in this instance it is not of much value.

A subtype (explicit or not) may be defined in terms of a previous subtype

```
subtype Feb_Day is Day_Number range 1 .. 29;
```

Any additional constraint must of course satisfy existing constraints

```
Day_Number range 0 .. 10
```

would be incorrect and cause Constraint_Error to be raised.

The above examples have shown constraints with static bounds. This is not always the case; in general the bounds can be given by arbitrary expressions and so the set of values of a subtype need not be static, that is known at compile time. However, it is an important fact that the bounds of a type are always static.

In conclusion then, a subtype does not introduce a new type but is merely a shorthand for an existing type or subtype with an optional constraint. However, in later chapters we will encounter several contexts in which an explicit constraint is not allowed; a subtype has to be introduced for these cases. We refer to a subtype name as a subtype mark and to the form consisting of a subtype mark with an optional constraint or null exclusion (see Chapter 11) as a subtype indication as shown by the syntax

subtype_mark ::= *subtype*_name

subtype_indication ::= [null exclusion] subtype_mark [constraint]

Thus we can restate the previous remark as saying that there are situations where a subtype mark has to be used whereas, as we have seen here, the more general subtype indication (which includes a subtype mark on its own) is allowed in object declarations.

The sensible use of subtypes has two advantages. It can ensure that programming errors are detected earlier by preventing variables from being

assigned inappropriate values. It can also increase the execution efficiency of a program. This particularly applies to array indexes (subscripts) as we shall see later.

We conclude this section by summarizing the assignment statement and the rules of strong typing. Assignment has the form

```
variable := expression;
```

and the two rules are

- both sides must have the same type,
- the expression must satisfy any constraints on the variable; if it does not, the assignment does not take place, and Constraint_Error is raised instead.

Note carefully the general principle that type errors (violations of the first rule) are detected during compilation whereas subtype errors (violations of the second rule) are detected during execution by the raising of Constraint_Error. (A clever compiler might give a warning during compilation.)

We have now introduced the basic concepts of types and subtypes. The remaining sections of this chapter illustrate these concepts further by considering in more detail the properties of the simple types of Ada.

It should be noted that subtype constraints restrict values to lie in a contiguous range; sometimes we wish to impose more flexible restrictions such as just odd values. This can be done in Ada 2012 using subtype predicates which are discussed in Section 16.4.

Exercise 6.4

1 Given the following declarations

```
I, J: Integer range 1 .. 10;
K: Integer range 1 .. 20;
```

which of the following assignment statements could raise Constraint_Error?

(a) `I := J;` (b) `K := J;` (c) `J := K;`

6.5 Simple numeric types

Perhaps surprisingly, a full description of the numeric types of Ada is deferred until much later in this book. The problems of numerical analysis (error estimates and so on) are complex and Ada is correspondingly rich in this area so that it can cope in a reasonably complete way with the needs of the numerical specialist. For our immediate purposes such complexity can be ignored. Accordingly, in this section, we merely consolidate a simple understanding of the two predefined numeric types Integer and Float.

First a reminder. We recall from Section 2.3 that the predefined numeric types are not portable and should be used with caution. However, all the operations described here apply to all integer and floating point types, both the predefined types such as Integer and Float and those defined by the user such as My_Integer and My_Float.

As we have seen, a constraint may be imposed on the type Integer by using the reserved word **range**. This is then followed by two expressions separated by two dots which, of course, must produce values of integer type. These expressions need not be literal constants. One could have

```
P: Integer range 1 .. I+J;
```

A range can be null as would happen in the above case if I+J turned out to be zero. Null ranges may seem pretty useless but they often automatically occur in limiting cases, and to exclude them would mean taking special action in such cases.

The minimum value of the type Integer is given by Integer'First and the maximum value by Integer'Last. These are our first examples of attributes. Ada contains various attributes denoted by a single quote (strictly, an apostrophe) followed by an identifier. We say 'Integer tick First'.

The value of Integer'First will depend on the implementation but will always be negative. On a two's complement machine it will be –Integer'Last–1 whereas on a one's complement machine it will be –Integer'Last. So on a minimal 16-bit two's complement implementation we will have

```
Integer'First = –32768
Integer'Last = +32767
```

Of course, we should always write Integer'Last rather than +32767 if that is what we logically want. Otherwise program portability could suffer.

Two useful subtypes are

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

These are so useful that they are declared for us in the package Standard.

The attributes First and Last also apply to subtypes so

```
Positive'First = 1
Natural'Last = Integer'Last
```

We turn now to a brief consideration of the type Float. It is possible to apply a constraint to the type Float in order to reduce the range such as

```
subtype Chance is Float range 0.0 .. 1.0;
```

where the range of course is given using expressions of the type Float. There are also attributes Float'First and Float'Last. It is not really necessary to say any more at this point.

The other predefined operations that can be performed on the types Integer and Float are much as one would expect in any programming language. They are summarized below.

+, – These are either unary operators (that is, taking a single operand) or binary operators taking two operands.

In the case of a unary operator, the operand can be either Integer or Float; the result will be of the same type. Unary + effectively does nothing. Unary – changes the sign.

In the case of a binary operator, both operands must be of the same type; the result will be of that type. Normal addition or subtraction is performed.

`*` Multiplication; both operands must be of the same type; again the result is of the same type.

`/` Division; both operands must be of the same type; again the result is of the same type. Integer division truncates towards zero.

rem Remainder; in this case both operands must be Integer and the result is Integer. It is the remainder on division.

mod Modulo; again both operands must be Integer and the result is Integer. This is the mathematical modulo operation.

abs Absolute value; this is a unary operator and the single operand may be Integer or Float. The result is again of the same type and is the absolute value. That is, if the operand is positive (or zero), the result is the same but if it is negative, the result is the corresponding positive value.

`**` Exponentiation; this raises the first operand to the power of the second. If the first operand is of type Integer, the second must be a positive integer or zero. If the first operand is of type Float, the second can be any integer. The result is of the same type as the first operand.

In addition, we can perform the operations `=`, `/=`, `<`, `<=`, `>` and `>=` which return a Boolean result True or False. Again both operands must be of the same type. Note the form of the not equals operator `/=`.

Although the above operations are mostly straightforward a few points are worth noting.

It is a general rule that mixed mode arithmetic is not allowed. One cannot, for example, add an integer value to a floating point value; both must be of the same type. A change of type from Integer to Float or vice versa can be done by using a type conversion which consists of the desired type name (or indeed subtype name) followed by the expression to be converted in parentheses.
So given

```
I: Integer := 3;
F: Float := 5.6;
```

we cannot write

```
I + F
```

but we must write

```
Float(I) + F
```

which uses floating point addition to give the floating point value 8.6, or

```
I + Integer(F)
```

which uses integer addition to give the integer value 9.

Conversion from Float to Integer always rounds rather than truncates, thus

```
1.4     becomes     1
1.6     becomes     2
```

and a value midway between two integers, such as 1.5, is always rounded away from zero so that 1.5 always becomes 2 and –1.5 becomes –2. Rounding can also be performed using various attributes as described in Section 17.4.

There is a subtle distinction between **rem** and **mod**. The **rem** operation gives the remainder corresponding to the integer division operation /. Integer division truncates towards zero; this means that the absolute value of the result is always the same as that obtained by dividing the absolute values of the operands. So

```
  7 /  3 =  2          7 rem  3 =  1
(–7) /  3 = –2        (–7) rem  3 = –1
  7 / (–3) = –2          7 rem (–3) =  1
(–7) / (–3) =  2       (–7) rem (–3) = –1
```

The remainder and quotient are always related by

```
(I/J) * J + I rem J = I
```

and it will also be noted that the sign of the remainder is always equal to the sign of the first operand I (the dividend).

However, **rem** is not always satisfactory. If we plot the values of I **rem** J for a fixed value of J (say 5) for both positive and negative values of I we get the pattern shown in Figure 6.1. As we can see, the pattern is symmetric about zero and consequently changes its incremental behaviour as we pass through zero. The **mod** operation, on the other hand, does have uniform incremental behaviour as shown in Figure 6.2.

The **mod** operation enables us to do normal modulo arithmetic. For example

```
(A+B) mod n = (A mod n + B mod n) mod n
```

for all values of A and B both positive and negative. For positive n, A **mod** n is always in the range 0 .. n–1, whereas for negative n, A **mod** n is always in the range n+1 .. 0. Of course, modulo arithmetic is only usually performed with a positive value for n. But the **mod** operator gives consistent and sensible behaviour for negative values of n also.

We can look upon **mod** as giving the remainder corresponding to division with truncation towards minus infinity. So

```
  7 mod  3 =  1
(–7) mod  3 =  2
  7 mod (–3) = –2
(–7) mod (–3) = –1
```

In the case of **mod** the sign of the result is always equal to the sign of the second operand whereas with **rem** it is the sign of the first operand.

The reader may have felt that this discussion has been somewhat protracted. In summary, it is perhaps worth saying that integer division with negative operands is rare. The operators **rem** and **mod** only differ when just one operand is negative. It will be found that in such cases it is almost always **mod** that is wanted. Moreover,

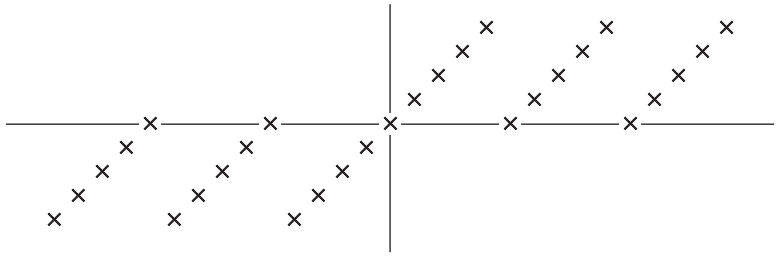


Figure 6.1 Behaviour of I **rem** 5 around zero.

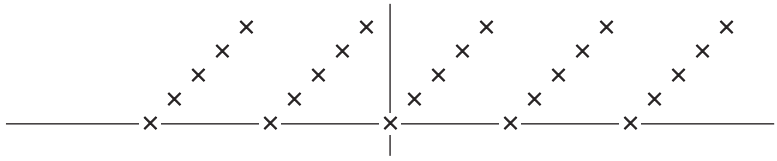


Figure 6.2 Behaviour of I **mod** 5 around zero.

note that if we are performing modular arithmetic in general then we should perhaps be using modular types; these are unsigned integer types with automatic range wraparound and are discussed in detail in Chapter 17.

Finally some notes on the exponentiation operator **. For a positive second operand, the operation corresponds to repeated multiplication. So

```
  3**4 = 3*3*3*3 = 81
3.0**4 = 3.0*3.0*3.0*3.0 = 81.0
```

A very subtle point is that the repeated multiplications can be performed by appropriate squarings as an optimization although as we shall see in Section 26.5 there can be slight differences in accuracy in the floating point case.

The second operand can be 0 and, of course, the result is then always the value one

```
  3**0 = 1
3.0**0 = 1.0
  0**0 = 1
0.0**0 = 1.0
```

The second operand cannot be negative if the first operand is an integer, because the result might not be a whole number. In fact, the exception Constraint_Error would be raised in such a case. But it is allowed for a floating point first operand and produces the corresponding reciprocal

```
3.0**(–4) = 1.0/81.0 = 0.0123456780123...
```

We conclude this section with a brief discussion on combining operators in an expression. As is usual, the operators have different precedence levels and the

natural precedence can be overruled by the use of parentheses. Operators of the same precedence are applied in order from left to right. A subexpression in parentheses obviously has to be evaluated before it can be used. But note that the order of evaluation of the two operands of a binary operator is not specified. The precedence levels of the operators we have met so far are shown below in increasing order of precedence

```
=   /=   <   <=   >   >=
+   –                      (binary)
+   –                      (unary)
*   /   mod   rem
**   abs
```

Thus

```
A/B*C          means        (A/B)*C
A+B*C+D        means        A+(B*C)+D
A*B+C*D        means        (A*B)+(C*D)
A*B**C         means        A*(B**C)
```

In general, as stated above, several operations of the same precedence can be applied from left to right and parentheses are not necessary. However, the syntax rules forbid multiple instances of the exponentiation operator without parentheses. Thus we cannot write

```
A**B**C
```

but must explicitly write either

```
(A**B)**C      or      A**(B**C)
```

This restriction avoids the risk of accidentally writing the wrong thing. Note however that the well established forms

```
A–B–C      and      A/B/C
```

are allowed. The syntax rules similarly prevent the mixed use of **abs** and ** without parentheses.

The precedence of unary minus needs care

```
–A**B      means      –(A**B)      rather than      (–A)**B
```

as in some languages. Also

```
A**–B      and      A*–B
```

are illegal. Parentheses are necessary.

Note finally that the precedence of **abs** is, confusingly, not the same as that of unary minus. As a consequence we can write

```
– abs X      but not      abs – X
```

since the latter requires parentheses as in **abs** (–X).

Exercise 6.5

1 Evaluate the expressions below given the following

```
I: Integer := 7;
J: Integer := –5;
K: Integer := 3;
```

(a) I*J*K
(b) I/J*K
(c) I/J/K
(d) J + 2 **mod** I
(e) J + 2 **rem** I
(f) K**K**K
(g) –J **mod** 3
(h) –J **rem** 3

2 Rewrite the following mathematical expressions in Ada. Use suitable identifiers of appropriate type.

(a) Mr^2 – moment of inertia of black hole
(b) $b^2–4ac$ – discriminant of quadratic
(c) $^4/_3\pi r^3$ – volume of sphere
(d) $p\pi a^4/8l\eta$ – viscous flowrate through tube

6.6 Enumeration types

Here are some examples of declarations of enumeration types starting with Colour which we introduced when discussing types in general.

```
type Colour is (Red, Amber, Green);
type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Stone is (Amber, Beryl, Quartz);
type Groom is (Tinker, Tailor, Soldier, Sailor,
               Rich_Man, Poor_Man, Beggar_Man, Thief);
type Solo is (Alone);
```

This introduces an example of overloading. The enumeration literal Amber can represent a Colour or a Stone. Both meanings are visible together and the second declaration does not hide the first whether they are declared in the same declarative part or one is in an inner declarative part. We can usually tell which is meant from the context, but in cases when we cannot we can use a qualification which consists of the literal in parentheses and preceded by a subtype mark (that is its type name or a relevant subtype name) and a single quote. Thus

```
Colour'(Amber)
Stone'(Amber)
```

Examples where this is necessary will occur later.

Although we can use Amber as an enumeration literal in two distinct enumeration types, we cannot use it as an enumeration literal and the identifier of a variable at the same time. The declaration of one would hide the other and they could not both be declared in the same declarative part. Later we will see that an enumeration literal can be overloaded with a subprogram.

There is no upper limit on the number of values in an enumeration type but there must be at least one. An empty enumeration type is not allowed.

Range constraints on enumeration types and subtypes are much as for integers. The constraint has the form

```
range lower_bound_expression .. upper_bound_expression
```

and this indicates the set of values from the lower bound to the upper bound inclusive. So we can write

```
subtype Weekday is Day range Mon .. Fri;
D: Weekday;
```

or

```
D: Day range Mon .. Fri;
```

and then we know that D cannot be Sat or Sun.

If the lower bound is above the upper bound then we get a null range, thus

```
subtype Colourless is Colour range Amber .. Red;
```

Note the curious fact that we cannot have a null subtype of a type such as Solo (since it only has one value).

The attributes First and Last apply to enumeration types and subtypes, so

```
Colour'First = Red
Weekday'Last = Fri
```

There are built-in functional attributes to give the successor or predecessor of an enumeration value. These consist of Succ or Pred following the type name and a single quote. Thus

```
Colour'Succ(Amber) = Green
Stone'Succ(Amber) = Beryl
Day'Pred(Fri) = Thu
```

Of course, the thing in parentheses can be an arbitrary expression of the appropriate type. If we try to take the predecessor of the first value or the successor of the last then the exception Constraint_Error is raised. In the absence of this exception we have, for any type T and any value X,

```
T'Succ(T'Pred(X)) = X
```

and vice versa.

Another functional attribute is Pos. This gives the position number of the enumeration value, that is the position in the declaration with the first one having a position number of zero. So

```
Colour'Pos(Red) = 0
Colour'Pos(Amber) = 1
Colour'Pos(Green) = 2
```

The opposite to Pos is Val. This takes the position number and returns the corresponding enumeration value. So

```
Colour'Val(0) = Red
Day'Val(6) = Sun
```

If we give a position value outside the range, as for example

```
Solo'Val(1)
```

then Constraint_Error is raised.

Clearly we always have

```
T'Val(T'Pos(X)) = X
```

and vice versa. We also note that

```
T'Succ(X) = T'Val(T'Pos(X) + 1)
```

they either both give the same value or both raise an exception.

It should be noted that these four attributes Succ, Pred, Pos and Val may also be applied to subtypes but are then identical to the same attributes of the corresponding base type.

It is probably rather bad practice to mess about with Pos and Val when it can be avoided. To do so encourages the programmer to think in terms of numbers rather than the enumeration values and hence destroys the abstraction.

The operators =, /=, <, <=, > and >= also apply to enumeration types. The result is defined by the order of the values in the type declaration. So

```
Red < Green     is   True
Wed >= Thu      is   False
```

The same result would be obtained by comparing the position values. So

```
T'Pos(X) < T'Pos(Y)      and      X < Y
```

are always equivalent (except that X < Y might be ambiguous).

Exercise 6.6

1 Evaluate

(a) Day'Succ(Weekday'Last)
(b) Weekday'Succ(Weekday'Last)
(c) Stone'Pos(Quartz)

2 Write suitable declarations of enumeration types for

(a) the colours of the rainbow,
(b) typical fruits.

3 Write an expression that delivers one's predicted bridegroom after eating a portion of pie containing N stones. Use the type Groom declared at the beginning of this section.

4 If the first of the month is in D where D is of type Day, then write an assignment replacing D by the day of the week of the Nth day of the month.

5 Why might X < Y be ambiguous?

6.7 The type Boolean

The type Boolean is a predefined enumeration type whose declaration in the package Standard is

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

Boolean values are used in constructions such as the if statement which we briefly met in Chapter 2. Boolean values are produced by the operators =, /=, <, <=, > and >= which have their expected meaning and apply to many types. So we can write constructions such as

```
if Today = Sun then
   Tomorrow := Mon;
else
   Tomorrow := Day'Succ(Today);
end if;
```

The Boolean type (we capitalize the name in memory of the English mathematician George Boole) has all the normal properties of an enumeration type, so, for instance

```
False < True = True !!
Boolean'Pos(True) = 1
```

We could even write the curious

```
subtype Always is Boolean range True .. True;
```

The type Boolean also has other operators which are as follows

not This is a unary operator and changes True to False and vice versa. It has the same precedence as **abs**.

and This is a binary operator. The result is True if both operands are True, and False otherwise.

or This is a binary operator. The result is True if one or other or both operands are True, and False only if they are both False.

xor This is also a binary operator. The result is True if one or other operand but not both are True. (Hence the name – eXclusive OR.) Another way of looking at it is to note that the result is True if and only if the operands are different. (The operator is known as 'not equivalent' in some languages.)

The effects of **and**, **or** and **xor** are summarized in the usual operator tables shown in Figure 6.3. The precedences of **and**, **or** and **xor** are equal to each other but lower than that of any other operator. In particular they are of lower precedence than the relational operators =, /=, <, <=, > and >=. As a consequence, parentheses are not needed in expressions such as

```
P < Q and I = J
```

and	F	T
F	F	F
T	F	T

or	F	T
F	F	T
T	T	T

xor	F	T
F	F	T
T	T	F

Figure 6.3 Operator tables for **and**, **or** and **xor**.

However, although the precedences are equal, **and**, **or** and **xor** cannot be mixed up in an expression without using parentheses (unlike + and – for instance). So

B **and** C **or** D is illegal

whereas

I + J – K is legal

We have to write

B **and** (C **or** D) or (B **and** C) **or** D

in order to emphasize which meaning is required.

The reader familiar with other programming languages will remember that **and** and **or** usually have a different precedence. The problem with this is that the programmer often gets confused and writes the wrong thing. It is to prevent this that Ada makes them the same precedence and insists on parentheses. Of course, successive applications of the same operator are permitted so

B **and** C **and** D is legal

and, as usual, evaluation goes from left to right although it does not matter since **and**, **or** and **xor** are associative. Thus, even (A **xor** B) **xor** C and A **xor** (B **xor** C) are always the same.

Take care with **not**. Its precedence is higher than **and**, **or** and **xor** as in other languages and so

not A **or** B means (**not** A) **or** B rather than **not** (A **or** B)

Boolean variables and constants can be declared and manipulated in the usual way

```
Danger: Boolean;
Signal: Colour;
...
Danger := Signal = Red;
```

The variable Danger is then True if the signal is Red. We can then write

```
if Danger then
   Stop_Train;
end if;
```

Note that we do not have to write

```
if Danger = True then
```

although this is perfectly legal; it just misses the point that Danger is already of type Boolean and so can be used directly as the condition.

A worse sin is to write

```
if Signal = Red then
   Danger := True;
else
   Danger := False;
end if;
```

rather than

```
Danger := Signal = Red;
```

The literals True and False could be overloaded by declaring for example

```
type Answer is (False, Dont_Know, True);
```

but to do so might make the program rather confusing.

Finally, it should be noted that it is often clearer to introduce our own two-valued enumeration type rather than use the type Boolean. Thus instead of

```
Wheels_OK: Boolean;
...
if Wheels_OK then
```

it is much better (and safer!) to write

```
type Wheel_State is (Up, Down);
Wheel_Position: Wheel_State;
...
if Wheel_Position = Up then
```

since whether the wheels are OK or not depends upon the situation. OK for landing is different from being OK for cruising. The enumeration type removes any doubt as to which is meant.

Exercise 6.7

1 Write declarations of constants T and F having the values True and False.

2 Using T and F from the previous exercise, evaluate

(a) T **and** F **and** T
(b) **not** T **or** T
(c) F = F = F = F
(d) (F = F) = (F = F)
(e) T < T < T < T

3 Evaluate

```
(A /= B) = (A xor B)
```

for all combinations of values of Boolean variables A and B.

6.8 Categories of types

At this point we pause to consolidate the material given in this chapter so far. The types in Ada can be grouped into various categories as shown in Figure 6.4. The broad grouping is into elementary types and composite types. Ultimately everything is essentially made up of elementary types.

Elementary types are divided into access types which are dealt with in Chapter 11 and scalar types some of which we have discussed in this chapter. Arrays and simple records are dealt with in Chapter 8. Tagged record types and interfaces are dealt with in Chapter 14 when we discuss object oriented programming. The other composite types, task and protected types which concern concurrent programming, are dealt with in Chapter 20. Record, task and protected types may also be parameterized with so-called discriminants which are discussed in Chapter 18.

The scalar types themselves can be subdivided into real types and discrete types. The real types are subdivided into floating point types and fixed point types. Our sole example of a real type has been the floating point type Float – the other real types are discussed in Chapter 17. The other types discussed in this chapter, Integer, Boolean and enumeration types in general, are discrete types – the only other kinds of discrete types to be introduced are other integer types again dealt with in Chapter 17, and character types which are in fact a form of enumeration type and are dealt with in Chapter 8. Note that the integer types are subdivided into signed integer types such as Integer and the modular types.

The key distinction between the discrete types and the real types is that the former have a clear-cut set of distinct separate (that is, discrete) values. The type Float, on the other hand, should be thought of as having a continuous set of values – we know in practice that a finite digital computer must implement a real type as actually a set of distinct values but this is really an implementation detail, the underlying mathematical concept is of a continuous set of values.

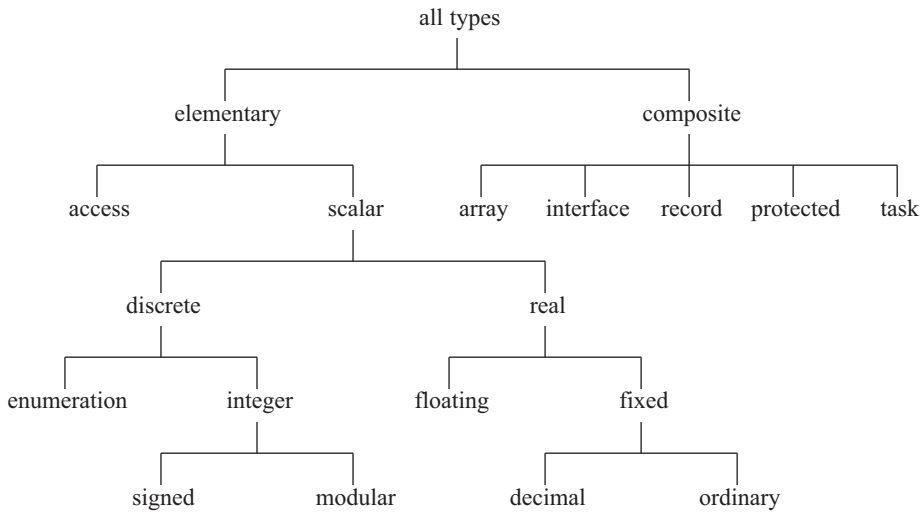


Figure 6.4 Ada type hierarchy.

The attributes Pos, Val, Succ and Pred apply to all discrete types (and subtypes) because the operations reflect the discrete nature of the values. We explained their meaning with enumeration types in Section 6.6. In the case of the type Integer the position number is simply the number itself so

```
Integer'Pos(N) = N
Integer'Val(N) = N
Integer'Succ(N) = N+1
Integer'Pred(N) = N−1
```

The application of these attributes to integers does at first sight seem pretty futile, but when we come to the concept of generic units in Chapter 17 we will see that it is convenient to allow them to apply to all discrete types.

Furthermore, despite our previous remarks about the continuous nature of real types, the attributes Pred and Succ (but not Pos and Val) also apply to all real types. They return the adjacent implemented number.

The attributes First and Last also apply to all scalar types and subtypes including real types. The attribute Range provides a useful shorthand since S'Range is equivalent to S'First .. S'Last. The attributes First_Valid and Last_Valid apply to static scalar subtypes and are of value with static predicates, see Section 16.4.

There are two other functional attributes for all scalar types and subtypes. These are Max and Min which take two parameters and return the maximum or minimum value respectively. Thus

```
Integer'Min(5, 10) = 5
```

Again we emphasize that Max, Min, Pos, Val, Succ and Pred for a subtype are identical to the corresponding operations on the base type, whereas in the case of First, Last and Range this is not so.

If an object is declared without an initial value then it is undefined unless a default initial value is given for the subtype. This can be done using the aspect Default_Value thus

```
type Colour is (Red, Amber, Green)
   with Default_Value => Red;
...
The_Signal: Colour;
```

and then The_Signal will automatically have the value Red. We cannot do this for the predefined types such as Integer and Float, but we can for our own types such as My_Integer and My_Float mentioned in Section 2.3. This aspect does not apply to access types which automatically have the default value of **null** as explained in Section 11.2. For composite types see Sections 8.2 and 8.6.

Finally, note the difference between type conversion and type qualification.

```
Float(I)          -- conversion
Integer'(I)       -- qualification
```

In the case of a conversion we are changing the type, whereas in the case of a qualification we are just stating it (usually to overcome an ambiguity). As a mnemonic aid *q*ualification uses a *q*uote.

In both cases we can use a subtype name and Constraint_Error could consequently arise. Thus

```
Positive(F)
```

converts the value of F to integer and then checks that it is positive, whereas

```
Positive'(I)
```

just checks that I is positive. In both cases the result is the checked value which is then used in an overall expression; these checks cannot just stand alone.

Exercise 6.8

1 Evaluate

(a) Boolean'Min(True, False) (b) Weekday'Max(Tue, Sat)

6.9 Expression summary

All the operators introduced so far are shown in Table 6.1 grouped by precedence level. In all the cases of binary operators except for **, the two operands must be of the same type.

We have actually now introduced all the operators of Ada except for one (&) although as we shall see there are further possible meanings to be added.

There are also two membership tests which apply to all scalar types (among others). These are **in** and **not in**. They are technically not operators although their precedence is the same as that of the relational operators =, /= and so on. They enable us to test whether a value lies within a specified range (including the end values) or satisfies a constraint implied by a subtype. The first operand is therefore a scalar expression, the second is a range or a subtype mark and the result is, of course, of type Boolean. Examples are

```
I not in 1 .. 10
I in Positive
Today in Weekday
```

Note that this is one of the situations where we have to use a subtype mark rather than a subtype indication. We could not replace the last example by

```
Today in Day range Mon .. Fri
```

although we could write

```
Today in Mon .. Fri
```

Ada seems a bit curious here!

The test **not in** is equivalent to using **in** and then applying **not** to the result, but **not in** is usually more readable. So the first expression above could be written as

```
not (I in 1 .. 10)
```

where the parentheses are necessary.

Table 6.1 Simple scalar operators.

Operator	*Operation*	*Operand(s)*	*Result*
and	conjunction	Boolean	Boolean
or	inclusive or	Boolean	Boolean
xor	exclusive or	Boolean	Boolean
=	equality	any	Boolean
/=	inequality	any	Boolean
<	less than	scalar	Boolean
<=	less than or equals	scalar	Boolean
>	greater than	scalar	Boolean
>=	greater than or equals	scalar	Boolean
+	addition	numeric	same
–	subtraction	numeric	same
+	identity	numeric	same
–	negation	numeric	same
*	multiplication	Integer	Integer
		Float	Float
/	division	Integer	Integer
		Float	Float
mod	modulo	Integer	Integer
rem	remainder	Integer	Integer
**	exponentiation	Integer, Natural	Integer
		Float, Integer	Float
not	negation	Boolean	Boolean
abs	absolute value	numeric	same

The reason that **in** and **not in** are not technically operators is that they cannot be overloaded as is explained in Chapter 10 when we deal with subprograms in detail.

Ada 2012 introduces other forms of membership tests and these are described in Section 9.4.

There are also two short circuit control forms **and then** and **or else** which like **in** and **not in** are not technically classified as operators.

The form **and then** is closely related to the operator **and**, whereas **or else** is closely related to the operator **or**. They may occur in expressions and have the same precedence as **and**, **or** and **xor**. The difference lies in the rules regarding the evaluation of their operands.

In the case of **and** and **or**, both operands are always evaluated but the order is not specified. In the case of **and then** and **or else** the left hand operand is always evaluated first and the right hand operand is only evaluated if it is necessary in order to determine the result.

So in

```
X and then Y
```

X is evaluated first. If X is false, the answer is false whatever the value of Y so Y is not evaluated. If X is true, Y has to be evaluated and the value of Y is the answer.

Similarly in

```
X or else Y
```

X is evaluated first. If X is true, the answer is true whatever the value of Y so Y is not evaluated. If X is false, Y has to be evaluated and the value of Y is the answer.

The forms **and then** and **or else** should be used in cases where the order of evaluation matters. A common circumstance is where the first condition protects against the evaluation of the second condition in circumstances that could raise an exception.

Suppose we need to test

```
I/J > K
```

and we wish to avoid the risk that J is zero. In such a case we could write

```
J /= 0 and then I/J > K
```

and we would then know that if J is zero there is no risk of an attempt to divide by zero. The observant reader will realize that this is not a very good example because one could usually write I>K*J (assuming J positive) – but even here we could get overflow. Better examples occur with arrays and access types and will be mentioned in due course.

Like **and** and **or**, the forms **and then** and **or else** cannot be mixed without using parentheses.

We now summarize the primary components of an expression (that is the things upon which the operators operate) that we have met so far. They are

- identifiers such as Colour
- literals such as 4.6, 2#101#
- type conversions such as Integer(F)
- qualified expressions such as Colour'(Amber)
- attributes such as Integer'Last
- function calls such as Day'Succ(Today)

A full consideration of functions will be found in Chapter 10. However, it is worth noting at this point that a function with one parameter is called by following its name by the parameter in parentheses. The parameter can be any expression of the appropriate type and could include further function calls. We will assume for the moment that we have available a simple mathematical library containing familiar functions such as

Sqrt	square root
Log	logarithm to base 10
Ln	natural logarithm
Exp	exponential function
Sin	sine
Cos	cosine

In each case they take a parameter of type Float and return a result of type Float.

We are now in a position to write statements such as

```
Root := (-B+Sqrt(B**2-4.0*A*C)) / (2.0*A);
Sin2x := 2.0*Sin(X)*Cos(X);
```

Finally a note on errors. The reader will have noticed that whenever anything could go wrong we have usually stated that the exception Constraint_Error will be raised. This is a general exception which applies to all sorts of violations of ranges. Constraint_Error is also raised if something goes wrong with the evaluation of an arithmetic expression itself before an attempt is made to store the result. An obvious example is an attempt to divide by zero.

As well as exceptions (which are discussed in detail in Chapter 15) there are erroneous constructs and bounded errors as mentioned in Chapter 2. The use of a variable before a value has been assigned to it (either explicitly or through the aspect Default_Value) is an important example of a bounded error. In addition there are situations where the order of evaluation is not defined and which could give rise to a nonportable program. Two cases which we have encountered so far where the order is not defined are

- The destination variable in an assignment statement may be evaluated before or after the expression to be assigned.
- The order of evaluation of the two operands of a binary operator is not defined.

(In the first case it should be realized that the destination variable could be an array component such as A(I+J) and so the expression I+J has to be evaluated as part of evaluating the destination variable; we will deal with arrays in Chapter 9.) Examples where these orders matter cannot be given until we deal with functions in Chapter 10.

This is perhaps a good moment to note that the use of Default_Value does have risks. Although it will prevent a variable from having a junk value, it can hide errors where we have inadvertently read an object before assigning a value to it. We can of course always make the default some easily recognized value such as 999 which will then show up in a debugger.

We conclude by mentioning that Ada 2012 has other forms of expressions including if expressions, case expressions, and quantified expressions. These are described in Chapter 9 in which we also discuss the new forms of membership tests.

Exercise 6.9

1 Rewrite the following mathematical expressions in Ada.

(a) $2\pi\sqrt{l/g}$ – period of a pendulum

(b) $\dfrac{m_0}{\sqrt{1-v^2/c^2}}$ – mass of relativistic particle

(c) $\sqrt{(2\pi n)}.n^n.e^{-n}$ – Stirling's approximation for $n!$ (integral n)

2 Rewrite **1**(c) replacing n by the real value x.

Checklist 6

Declarations and statements are terminated by a semicolon.
Initialization, like assignment, uses :=.
Any initial value is evaluated for each object in a declaration.
Elaboration of declarations is linear.
The identifier of an object may not be used in its own declaration.
Each type definition introduces a quite distinct type.
A subtype is not a new type but merely a shorthand for a type with a possible constraint.
A type is always static, a subtype need not be.
There is no mixed mode arithmetic.
Distinguish **rem** and **mod** for negative operands.
Exponentiation with a negative exponent only applies to real types.
Take care with the precedence of the unary operators.
A scalar type cannot be empty, a subtype can.
Max, Min, Pos, Val, Succ and Pred on subtypes are the same as on the base type.
First and Last are different for subtypes.
Qualification uses a quote.
Order of evaluation of binary operands is not defined.
Distinguish **and**, **or** and **and then**, **or else**.
Subtypes can be defined by a null exclusion in Ada 2005 as well as by a constraint.

New in Ada 2012

The aspect Default_Value for scalar subtypes is new in Ada 2012.
Membership tests are much more flexible in Ada 2012 and are described in Section 9.4.
Ada 2012 also has if expressions, case expressions and quantified expressions. These are described in Chapter 9.
Ada 2012 has new forms of subtypes using subtype predicates. These are described in Section 16.4.

7 Control Structures

This chapter describes the three bracketed sequential control structures of Ada. These are the if statement, the case statement and the loop statement. These three statements enable programs to be written with a clear flow of control without using goto statements and labels. However, for pragmatic reasons, Ada does have a goto statement and this is also described in this chapter.

The three control structures exhibit a similar bracketing style. There is an opening reserved word **if**, **case** or **loop** and this is matched at the end of the structure by the same reserved word preceded by **end**. The whole is, as usual, terminated by a semicolon. So we have

```
if                  case                loop
   ...                 ...                 ...
end if;             end case;           end loop;
```

In the case of the loop statement the word **loop** can be preceded by an iteration scheme commencing with **for** or **while**.

Corresponding structures for expressions such as if expressions and case expressions which are introduced in Ada 2012 are described in Chapter 9.

7.1 If statements

The simplest form of if statement starts with the reserved word **if** followed by a Boolean expression and the reserved word **then**. This is then followed by a sequence of statements which will be executed if the Boolean expression turns out to be True. The end of the sequence is indicated by the closing **end if**. The Boolean expression can, of course, be of arbitrary complexity and the sequence of statements can be of arbitrary length.

A simple example is

```
if Hungry then
   Cook;
   Eat;
   Wash_Up;
end if;
```

In this, Hungry is a Boolean variable and Cook, Eat and Wash_Up are various subprograms describing the details of the activities. The statement Eat; merely calls the corresponding subprogram (subprograms are dealt with in detail in Chapter 10).

The effect of this if statement is that if variable Hungry is True then we call the subprograms Cook, Eat and Wash_Up in sequence and otherwise we do nothing. In either case we then obey the statement following the if statement.

Note how we indent the statements to show the flow structure of the program. This is most important since it enables the program to be understood so much more easily. The **end if** should be underneath the corresponding **if** and the **then** is best placed on the same line as the **if**.

Sometimes, if the whole statement is very short it can all go on one line

```
if X < 0.0 then X := –X; end if;
```

Note that **end if** will always be preceded by a semicolon. This is because the semicolons terminate statements rather than separate them as in some other languages.

Often we will want to do alternative actions according to the value of the condition. In this case we add **else** followed by the alternative sequence to be obeyed if the condition is False. We saw an example of this in the previous chapter

```
if Today = Sun then
   Tomorrow := Mon;
else
   Tomorrow := Day'Succ(Today);
end if;
```

In Ada 2012 this can be rewritten using an if expression thus

```
Tomorrow :=
      (if Today = Sun then Mon else Day'Succ(Today));      -- in 2012
```

as will be explained in detail in Chapter 9.

The statements in the sequences in an if statement after **then** and **else** can be quite arbitrary and so could be further nested if statements.

Suppose we have to solve the quadratic equation

$$ax^2 + bx + c = 0$$

The first thing to check is a. If $a = 0$ then the equation degenerates into a linear equation with a single root $–c/b$. (Mathematicians will understand that the other root

has slipped off to infinity.) If a is not zero then we test the discriminant $b^2 - 4ac$ to see whether the roots are real or complex. We could program this as

```
if A = 0.0 then
        -- linear case
else
   if B**2 - 4.0*A*C >= 0.0 then
        -- real roots
   else
        -- complex roots
   end if;
end if;
```

Observe the repetition of **end if**. This is rather ugly and occurs sufficiently frequently to justify an additional construction. This uses the reserved word **elsif** as follows

```
if A = 0.0 then
        -- linear case
elsif B**2 - 4.0*A*C >= 0.0 then
        -- real roots
else
        -- complex roots
end if;
```

This construction emphasizes the essentially equal status of the three cases and also the sequential nature of the tests.

The **elsif** part can be repeated an arbitrary number of times and the final **else** part is optional. The behaviour is simply that each condition is evaluated in turn until one that is True is encountered; the corresponding sequence is then obeyed. If none of the conditions turns out to be True then the else part, if any, is taken; if there is no else part then none of the sequences is obeyed.

Note the spelling of **elsif**. It is the only reserved word of Ada that is not an English word (apart from operators such as **xor**). Note also the layout – we align **elsif** and **else** with the **if** and **end if** and all the sequences are indented equally.

As a further example, suppose we are drilling soldiers and they can obey four different orders described by

```
type Move is (Left, Right, Back, On);
```

and that their response to these orders is described by calling subprograms Turn_Left, Turn_Right and Turn_Back or by doing nothing at all respectively. Suppose that the variable Order of type Move contains the order to be obeyed. We could then write the following

```
if Order = Left then
   Turn_Left;
else
   if Order = Right then
      Turn_Right;
```

```
      else
         if Order = Back then
            Turn_Back;
         end if;
      end if;
   end if;
```

But it is far clearer and neater to write

```
if Order = Left then
   Turn_Left;
elsif Order = Right then
   Turn_Right;
elsif Order = Back then
   Turn_Back;
end if;
```

This illustrates a situation where there is no **else** part. However, although better than using nested if statements, this is still a bad solution because it obscures the symmetry and mutual exclusion of the four cases ('mutual exclusion' means that by their very nature only one can apply). We have been forced to impose an ordering on the tests which is quite arbitrary and not the essence of the problem. The proper solution is to use the case statement as we shall see in the next section.

There is no directly corresponding contraction for **then if**. Instead we can often use the short circuit control form **and then** which was discussed in Section 6.9.

So, rather than writing

```
if J > 0 then
   if I/J > K then
      Action;
   end if;
end if;
```

we can instead write

```
if J > 0 and then I/J > K then
   Action;
end if;
```

and this avoids the repetition of **end if**.

Exercise 7.1

1 The variables Day, Month and Year contain today's date. They are declared as

```
Day: Integer range 1 .. 31;
Month: Month_Name;
Year: Integer range 1901 .. 2399;
```

where

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

Write statements to update the variables to contain tomorrow's date. What happens if today is 31 Dec 2399?

2 X and Y are two variables of type Float. Write statements to swap their values, if necessary, to ensure that the larger value is in X. Use a block to declare a temporary variable T.

7.2 Case statements

A case statement allows us to choose one of several sequences of statements according to the value of an expression. For instance, the example of the drilling soldiers should be written as

```
case Order is
   when Left => Turn_Left;
   when Right => Turn_Right;
   when Back => Turn_Back;
   when On => null;
end case;
```

All possible values of the expression must be provided for in order to guard against accidental omissions. If, as in this example, no action is required for one or more values then the null statement has to be used. The null statement, written as just **null**; does absolutely nothing but its presence indicates that we truly want to do nothing. The sequence of statements here, as in the if statement, must contain at least one statement. (There is no empty statement as in Pascal and C.)

It often happens that the same action is desired for several values of the expression. Consider the following

```
case Today is
   when Mon | Tue | Wed | Thu => Work;
   when Fri                   => Work; Party;
   when Sat | Sun             => null;
end case;
```

This expresses the idea that on Monday to Thursday we go to work. On Friday we also go to a party. At the weekend we do nothing. The alternatives are separated by the vertical bar character. Note again the use of a null statement.

If several successive values have the same action then it is more convenient to use a range

```
when Mon .. Thu => Work;
```

We can also express the idea of a default action to be taken by all values not explicitly stated by using the reserved word **others**. So we can write

```
case Today is
   when Mon .. Thu => Work;
   when Fri        => Work; Party;
   when others     => null;
end case;
```

It is possible to have ranges as alternatives. The various possibilities can best be seen from the formal syntax (note that in the production for discrete_choice_list, the vertical bar stands for itself and is not a metasymbol).

```
case_statement ::=
      case selecting_expression is
         case_statement_alternative
        {case_statement_alternative}
      end case;

case_statement_alternative ::=
      when discrete_choice_list => sequence_of_statements

discrete_choice_list ::= discrete_choice { | discrete_choice}

discrete_choice ::= choice_expression | discrete_subtype_indication
                        | range | others

subtype_indication ::= [null exclusion] subtype_mark [constraint]

subtype_mark ::= subtype_name

range ::= range_attribute_reference
              | simple_expression .. simple_expression
```

We see that **when** is followed by one or more discrete choices separated by vertical bars and that a discrete choice may be a choice expression, a discrete subtype indication, a range or **others**. A choice expression, of course, just gives a single value – Fri being a trivial example (a choice expression is an expression excluding membership tests which could be confusing because they might include vertical bars as well). A subtype indication is a subtype mark with optionally a null exclusion (which is not possible in this context) or an appropriate constraint which in this context has to be a range constraint which is just the reserved word **range** followed by the syntactic form range. A discrete choice can also be just a range on its own which is typically two simple expressions separated by two dots – Mon .. Thu being a simple example; a range can also be given by a range attribute which we will meet in the next chapter. Finally, a discrete choice can be just **others**.

There is considerable flexibility and we can write any of

```
Mon .. Thu
Day range Mon .. Thu
Weekday range Mon .. Thu
```

which all mean the same. Note that there is not much point in using the subtype name followed by a constraint since the constraint alone will do. However, it might be useful to use a subtype name alone when that exactly corresponds to the range required. So we could rewrite the example as

```
case Today is
   when Weekday => Work;
                   if Today = Fri then Party; end if;
   when others  => null;
end case;
```

although this solution feels untidy.

There are various other restrictions that the syntax does not tell us. One is that if we use **others** then it must appear alone and as the last alternative. As stated earlier it covers all values not explicitly covered by the previous alternatives (one can write **others** even if there are no other cases left).

Another very important restriction is that all the choices must be static so that they can be evaluated at compile time. Thus all expressions in discrete choices must be static – in practice they will usually be literals as in our examples. Similarly, if a choice is just a subtype such as Weekday then it too must be static.

At the beginning of this section we remarked that all possible values of the expression after **case** must be provided for. This usually means all values of the type of the expression. However, if the expression is of a simple form and belongs to a static subtype (that is one whose constraints are static expressions and so can be determined at compile time) then only values of that subtype need be provided for. In other words, if the compiler can tell that only a subset of values is possible then only that subset need and must be covered. The simple forms allowed for the expression are the name of an object of the static subtype (and that includes a function call whose result is of the static subtype) or a qualified or converted expression whose subtype mark is that of the static subtype.

Thus in the example, since Today is declared to be of type Day without any constraints, all values of the type Day must be provided for. However, if Today were of the static subtype Weekday then only the values Mon .. Fri would be possible and so only these could and need be covered. Even if Today is not constrained we can still write the expression as the qualified expression Weekday'(Today) and then again only Mon .. Fri are possible. So we could write

```
case Weekday'(Today) is
   when Mon .. Thu => Work;
   when Fri        => Work; Party;
end case;
```

but, of course, if Today happens to take a value not in the subtype Weekday (that is, Sat or Sun) then Constraint_Error will be raised. Mere qualification cannot prevent Today from being Sat or Sun. So this is not really a solution to our original problem.

As further examples, suppose we had variables

```
I: Integer range 1 .. 10;
J: Integer range 1 .. N;
```

where N is not static. Then we know that I belongs to a static subtype (albeit anonymous) whereas we cannot say the same about J. If I is used as an expression in a case statement then only the values 1 .. 10 have to be catered for, whereas if J is so used then the full range of values of type Integer (Integer'First .. Integer'Last) have to be catered for.

The above discussion on the case statement has no doubt given the reader the impression of considerable complexity. It therefore seems wise to summarize the key points which will in practice need to be remembered

- Every possible value of the expression after **case** must be covered once and once only.
- All values and ranges after **when** must be static.
- If **others** is used it must be last and on its own.

Finally, note that the use of **others** will be necessary if the expression after **case** is of the type *universal_integer*. This type will be discussed in Chapter 17 when we consider numeric types in detail. The main use of the type occurs with integer literals. However, certain attributes such as Pos are of the type *universal_integer* and so if the case expression is of the form Character'Pos(C) then **others** will be required even if the values it covers could never arise.

Exercise 7.2

1 Rewrite Exercise 7.1(**1**) to use a case statement to set the correct value in End_Of_Month.

2 A vegetable gardener digs in winter, sows seed in spring, tends the growing plants in summer and harvests the crop in the autumn or fall. Write a case statement to call the appropriate subprogram Dig, Sow, Tend or Harvest according to the month M. Declare appropriate subtypes if desired.

3 An improvident man is paid on the first of each month. For the first ten days he gorges himself, for the next ten he subsists and for the remainder he starves. Call subprograms Gorge, Subsist and Starve according to the day D. Assume the variable End_Of_Month has been set and that D is declared as

```
D: Integer range 1 .. End_Of_Month;
```

7.3 Loop statements

Loop statements can take various forms including special forms for iterating over containers. The simplest form of loop statement is

```
loop
   sequence_of_statements
end loop;
```

The statements of the sequence are then repeated indefinitely unless one of them terminates the loop by some means. So immortality could be represented by

```
loop
   Work;
   Eat;
   Sleep;
end loop;
```

As a more concrete example consider the problem of computing the base e of natural logarithms from the infinite series

$$e = 1 + 1/1! + 1/2! + 1/3! + 1/4! + ...$$

where

$$n! = n \times (n–1) \times (n–2) \ldots 3 \times 2 \times 1$$

A possible solution is

```
declare
   E: Float := 1.0;
   I: Integer := 0;
   Term: Float := 1.0;
begin
   loop
      I := I + 1;
      Term := Term / Float(I);
      E := E + Term;
   end loop;
   ...
```

Each time around the loop a new term is computed by dividing the previous term by I. The new term is then added to the sum so far which is accumulated in E. The term number I is an integer because it is logically a counter and so we have to write Float(I) as the divisor. The series is started by setting values in E, I and Term which correspond to the first term (that for which I = 0).

The computation then goes on for ever with E becoming a closer and closer approximation to *e*. In practice, because of the finite accuracy of the computer, Term will become zero and continued computation will be pointless. But in any event we presumably want to stop at some point so that we can do something with our computed result. We can do this with the statement

```
exit;
```

If this is obeyed inside a loop then the loop terminates at once and control passes to the point immediately after **end loop**.

Suppose we decide to stop after *N* terms of the series – that is when I = N. We can do this by writing the loop as

```
loop
   if I = N then exit; end if;
   I := I + 1;
   Term := Term / Float(I);
   E := E + Term;
end loop;
```

The construction

```
if condition then exit; end if;
```

is so common that a special shorthand is provided

```
exit when condition;
```

So we now have

```
loop
   exit when I = N;
   I := I + 1;
   Term := Term / Float(I);
   E := E + Term;
end loop;
```

Although an exit statement can appear anywhere inside a loop – it could be in the middle or near the end – a special form of loop is provided for the frequent case where we want to test a condition at the start of each iteration. This is the while statement and uses the reserved word **while** followed by the condition for the loop to be continued. So we could write

```
while I /= N loop
   I := I + 1;
   Term := Term / Float(I);
   E := E + Term;
end loop;
```

The condition is naturally evaluated each time around the loop.

The final form of loop is the for statement which allows a specific number of iterations with a loop parameter taking in turn all the values of a range of a discrete type. Our example could be recast as

```
for I in 1 .. N loop
   Term := Term / Float(I);
   E := E + Term;
end loop;
```

where I takes the values 1, 2, 3, ..., N.

The loop parameter I is implicitly declared by its appearance after **for** and does not have to be declared outside. It takes its type from the range and within the loop behaves as a constant so that it cannot be changed except by the loop mechanism itself. When we leave the loop (by whatever means) I ceases to exist (because it was implicitly declared by the loop) and so we cannot read its final value from outside.

We could leave the loop by an exit statement – if we wanted to know the final value we could copy the value of I into a variable declared outside the loop thus

```
if condition_to_exit then
   Last_I := I;
   exit;
end if;
```

The values of the range are normally taken in ascending order. Descending order can be specified by writing

```
for I in reverse 1 .. N loop
```

but the range itself is always written in ascending order.

It is not possible to specify a numeric step size of other than 1. This should not be a problem since the vast majority of loops go up by steps of 1 and almost all the rest go down by steps of 1. The very few which do behave otherwise can be explicitly programmed using the while form of loop.

The range can be null (as for instance if N happened to be zero or negative in our example) in which case the sequence of statements will not be obeyed at all. Of course, the range itself is evaluated only once and cannot be changed inside the loop.

Thus

```
N := 4;
for I in 1 .. N loop
   ...
   N := 10;
end loop;
```

results in the loop being executed just four times despite the fact that N is changed to ten.

Our examples have all shown the lower bound of the range being 1. This, of course, need not be the case. Both bounds can be arbitrary dynamically evaluated expressions. Furthermore, the loop parameter need not be of integer type. It can be of any discrete type, as determined by the range.

We could, for instance, simulate a week's activity by

```
for Today in Mon .. Sun loop
   case Today is
      ...
   end case;
end loop;
```

This implicitly declares Today to be of type Day and obeys the loop with the values Mon, Tue, ..., Sun in turn.

The other forms of discrete range (using a type or subtype name) are of advantage here. The essence of Mon .. Sun is that it embraces all the values of the type Day. It is therefore better to write the loop using a form of discrete range that conveys the idea of completeness

```
for Today in Day loop
   ...
end loop;
```

And again since we know that we do nothing at weekends anyway we could write

```
for Today in Day range Mon .. Fri loop
```

or better

```
for Today in Weekday loop
```

It is interesting to note a difference regarding the determination of types in the case statement and for statement. In the case statement, the type of a range after **when** is determined from the type of the expression after **case**. In the for statement,

the type of the loop parameter is determined from the type of the range after **in**. The dependency is the other way round.

It is therefore necessary for the type of the range to be unambiguous in the for statement. This is usually the case but if we had two enumeration types with two overloaded literals such as

```
type Planet is (Mercury, Venus, Earth, Mars, Jupiter,
                Saturn, Uranus, Neptune, Pluto);
type Roman_God is (Janus, Mars, Jupiter, Juno, Vesta,
                   Vulcan, Saturn, Mercury, Minerva);
```

then

```
for X in Mars .. Saturn loop                    -- illegal
```

would be ambiguous and the compiler would not compile our program. We could resolve the problem by qualifying one of the expressions

```
for X in Planet'(Mars) .. Saturn loop
```

or (probably better) by using a form of range giving the type explicitly

```
for X in Planet range Mars .. Saturn loop
```

On the other hand

```
for X in Mars .. Vulcan loop
```

is not ambiguous because Vulcan can only be a Roman god and not a planet.

When we have dealt with numerics in more detail we will realize that the range 1 .. 10 is not necessarily of type Integer (it might be Long_Integer). A general application of the rule that the type must not be ambiguous in a for statement would lead us to have to write

```
for I in Integer range 1 .. 10 loop
```

However, this would be very tedious and so in such cases the type Integer can be omitted and is then implied by default. We can therefore conveniently write

```
for I in 1 .. 10 loop
```

More general expressions are also allowed so that we could also write

```
for I in -1 .. 10 loop
```

although we recall that -1 is not a literal as explained in Section 5.4. We will return to this topic in Section 17.1.

Finally, we reconsider the exit statement. The simple form encountered earlier always transfers control to immediately after the innermost embracing loop. But of course loops may be nested and sometimes we may wish to exit from a nested construction. As an example suppose we are searching in two dimensions

```
for I in 1 .. N loop
   for J in 1 .. M loop
      -- if values of I and J satisfy
      -- some condition then leave nested loop
   end loop;
end loop;
```

A simple exit statement in the inner loop would merely take us to the end of that loop and we would have to recheck the condition and exit again. This can be avoided by naming the outer loop and using the name in the exit statement thus

```
Search:
for I in 1 .. N loop
   for J in 1 .. M loop
      if condition_OK then
         I_Value := I;
         J_Value := J;
         exit Search;
      end if;
   end loop;
end loop Search;
-- control passes here
```

A loop is named by preceding it with an identifier and colon. (It looks remarkably like a label in other languages but it is not and cannot be 'gone to'.) The identifier must be repeated between the corresponding **end loop** and the semicolon. The conditional form of exit can also refer to a loop by name; so

```
exit Search when condition;
```

transfers control to the end of the loop named Search if the condition is True.

In Ada 2012, other forms of for loop can be used with arrays (see Section 8.1) and with containers (see Chapter 24).

Exercise 7.3

1 The statement Get(I); reads the next value from the input file into the integer variable I. Write statements to read and add together a series of numbers. The end of the series is indicated by a dummy negative value.

2 Write statements to determine the power of 2 in the factorization of N. Compute the result in Count but do not alter N.

3 Compute

$$g = \sum_{i=1}^{n} 1/i - \log n$$

(As $n \to \infty$, $g \to \gamma = 0.577215665...$, Euler's constant.)

7.4 Goto statements and labels

Many will be surprised that a modern programming language should contain a goto statement at all. It is now considered to be extremely bad practice to use goto statements because of the resulting difficulty in proving correctness of the program, maintenance and so on.

So why provide a goto statement? The main reason concerns automatically generated programs. If we try to transliterate (by hand or machine) a program from some other language into Ada then the goto will probably be useful. Another example might be where the program is generated automatically from some high level specification. Finally there may be cases where the goto is the neatest way – perhaps as a way out of some deeply nested structure – but the alternative of raising an exception (see Chapter 15) could also be considered.

In order to put us off using gotos and labels (and perhaps so that our manager can spot them if we do) the notation for a label is unusual and stands out like a sore thumb. A label is an identifier enclosed in double angled brackets thus

```
<<The_Devil>>
```

and a goto statement takes the expected form of the reserved word **goto** followed by the label identifier and semicolon

```
goto The_Devil;
```

Perhaps the most common use of a goto statement is to transfer control to just before the end of a loop in the case when an iteration proves useless and we wish to try the next iteration. Thus we might write

```
for I in ... loop
   ...                                        -- various statements
   if not OK then goto End_Of_Loop; end if;
   ...                                        -- other statements
<<End_Of_Loop>>                               -- now try another iteration
end loop;
```

It is usually the case that a label always immediately precedes a statement. But in Ada 2012 it can also be at the end of a sequence of statements as here. In earlier versions of Ada we had to write an explicit null statement at the end of the loop.

A goto statement cannot be used to transfer control into an if, case or loop statement nor between the arms of an if or case statement. But it can transfer control out of a loop or out of a block.

7.5 Statement classification

The various statements in Ada can be classified as shown in Figure 7.1. All statements can have one or more labels. The simple statements cannot be decomposed lexically into other statements whereas the compound statements can be so decomposed and can therefore be nested. Statements are obeyed sequentially unless one of them is a control statement (or an exception is raised). The return statement has both simple and compound forms.

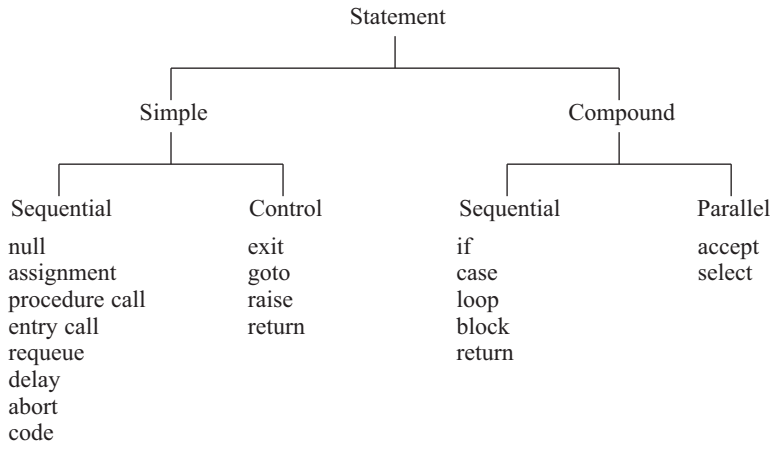


Figure 7.1 Classification of statements.

Further detail on the assignment statement is in the next chapter when we discuss composite types. Procedure calls and return statements are discussed in Chapter 10 and the raise statement which is concerned with exceptions is discussed in Chapter 15. The code statement is mentioned in Chapter 25. The remaining statements (entry call, requeue, delay, abort, accept and select) concern tasking and are dealt with in Chapter 20.

Checklist 7

Statement brackets must match correctly.

Use **elsif** where appropriate.

The choices in a case statement must be static.

All possibilities in a case statement must be catered for.

If **others** is used it must be last and on its own.

The expression after **case** can be qualified in order to reduce the alternatives.

A loop parameter behaves as a constant.

A named loop must have the name at both ends. Avoid gotos.

Use the recommended layout.

In Ada 2005 the return statement has both simple and compound forms.

New in Ada 2012

Labels are permitted at the end of a sequence of statements.

There are other forms of for loops for arrays and containers.

8 Arrays and Records

This chapter describes the main composite types which are arrays and records. It also completes our discussion of enumeration types by introducing characters and strings. At this stage we discuss arrays fairly completely but consider only the simplest forms of records. Tagged records and interfaces which permit extension and polymorphism are discussed in Chapter 14 and discriminated records which include variant records are deferred until Chapter 18. Tasks and protected types which are also classified as composite are discussed in Chapter 20.

8.1 Arrays

An array is a composite object consisting of a number of components all of the same type (strictly, subtype). An array can be of one, two or more dimensions. A typical array declaration might be

```
A: array (Integer range 1 .. 6) of Float;
```

This declares A to be a variable object which has six components, each of which is of type Float. The individual components are referred to by following the array name with an expression in parentheses giving an integer value in the discrete range 1 .. 6. If this expression, known as the index value, has a value outside the bounds of the range, then the exception Constraint_Error will be raised. We could set zero in each component of A by writing

```
for I in 1 .. 6 loop
   A(I) := 0.0;
end loop;
```

An array can be of several dimensions, in which case a separate range is given for each dimension. So

```
AA: array (Integer range 0 .. 2, Integer range 0 .. 3) of Float;
```

is an array of 12 components in total, each of which is referred to by two integer index values, the first in the range 0 .. 2 and the second in the range 0 .. 3. The components of this two-dimensional array could all be set to zero by a nested loop thus

```
for I in 0 .. 2 loop
   for J in 0 .. 3 loop
      AA(I, J) := 0.0;
   end loop;
end loop;
```

The discrete ranges do not have to be static; one could have

```
N: Integer := ... ;
B: array (Integer range 1 .. N) of Boolean;
```

and the value of N at the point when the declaration of B is elaborated would determine the number of components in B. Of course, the declaration of B might be elaborated many times during the course of a program – it might be inside a loop for example – and each elaboration will give rise to a new life of a new array and the value of N could be different each time. Like other declared objects, the array B ceases to exist once we pass the end of the block containing its declaration. Because of 'linear elaboration of declarations' both N and B could be declared in the same declarative part but the declaration of N would have to precede that of B.

The discrete range in an array index follows similar rules to that in a for statement. An important one is that a range such as 1 .. 6 implies type Integer so we could have written

```
A: array (1 .. 6) of Float;
```

However, an array index subtype could be a subtype of any discrete type. We could for example have

```
Hours_Worked: array (Day) of Float;
```

This array has seven components denoted by Hours_Worked(Mon), ..., Hours_Worked(Sun). We could set suitable values in these variables by

```
for D in Weekday loop
   Hours_Worked(D) := 8.0;
end loop;
Hours_Worked(Sat) := 0.0;
Hours_Worked(Sun) := 0.0;
```

If we only wanted to declare the array Hours_Worked to have components corresponding to Mon .. Fri then we could write

```
Hours_Worked: array (Day range Mon .. Fri) of Float;
```

or (better)

```
Hours_Worked: array (Weekday) of Float;
```

Arrays have various attributes relating to their indices. A'First and A'Last give the lower and upper bound of the first (or only) index of A. So using our last declaration of Hours_Worked

```
Hours_Worked'First = Mon
Hours_Worked'Last = Fri
```

A'Length gives the number of values of the first (or only) index. So

```
Hours_Worked'Length = 5
```

A'Range is short for A'First .. A'Last. So

```
Hours_Worked'Range    is    Mon .. Fri
```

The same attributes can be applied to the various dimensions of a multi-dimensional array by adding the dimension number in parentheses. It has to be a static expression. So, in the case of our two-dimensional array AA we have

```
AA'First(1)  = 0                AA'First(2)  = 0
AA'Last(1)  = 2                 AA'Last(2)  = 3
AA'Length(1)  = 3               AA'Length(2)  = 4
```

and

```
AA'Range(1)    is    0 .. 2     AA'Range(2)    is    0 .. 3
```

The first dimension is assumed if (1) is omitted. It is perhaps better practice to specifically state (1) for multidimensional arrays and omit it for one-dimensional arrays.

It is always best to use the attributes where possible in order to reflect relationships among entities in a program because it generally means that if the program is modified, the modifications are localized.

The Range attribute is particularly useful with loops. Our earlier examples are better written as

```
for I in A'Range loop
   A(I) := 0.0;
end loop;
```

```
for I in AA'Range(1) loop
   for J in AA'Range(2) loop
      AA(I, J) := 0.0;
   end loop;
end loop;
```

The Range attribute can also be used in a declaration. Thus

```
J: Integer range A'Range;
```

is equivalent to

```
J: Integer range 1 .. 6;
```

If a variable is to be used to index an array as in A(J) it is usually best if the variable has the same constraints as the discrete range in the array declaration. This will usually minimize the run-time checks necessary. It has been found that in such circumstances it is usually the case that the index variable J is assigned to less frequently than the array component A(J) is accessed. We will return to this topic in Section 15.3.

The array components we have seen are just variables in the ordinary way. They can therefore be assigned to and used in expressions.

Like other variable objects, arrays can be given an initial value. This will often be denoted by an aggregate which is the literal form for an array value. The simplest form of aggregate (a positional aggregate) consists of a list of expressions giving the values of the components in order, separated by commas and enclosed in parentheses. So we could initialize the array A by

```
A: array (1 .. 6) of Float := (0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
```

In the case of a multidimensional array the aggregate is written in a nested form

```
AA: array (0 .. 2, 0 .. 3) of Float := ((0.0, 0.0, 0.0, 0.0),
                                        (0.0, 0.0, 0.0, 0.0),
                                        (0.0, 0.0, 0.0, 0.0));
```

and this illustrates that the first index is the 'outer' one. Or thinking in terms of rows and columns, the first index is the row number.

An aggregate must be complete. If we initialize any component of an array, we must initialize them all.

The initial values for the individual components need not be literals, they can be any expressions. These expressions are evaluated when the declaration is elaborated but the order of evaluation of the expressions in the aggregate is not specified.

An array can be declared as constant in which case an initial value is mandatory as explained in Section 6.1. Constant arrays are of frequent value as look-up tables. The following array can be used to determine whether a particular day is a working day or not

```
Work_Day: constant array (Day) of Boolean :=
                                  (True, True, True, True, True, False, False);
```

An interesting example would be an array enabling tomorrow to be determined without worrying about the end of the week.

```
Tomorrow: constant array (Day) of Day :=
                                  (Tue, Wed, Thu, Fri, Sat, Sun, Mon);
```

For any day D, Tomorrow (D) is the following day.

It should be noted that the array components can be of any definite type or subtype. Also the dimensions of a multidimensional array can be of different discrete types. An extreme example would be

```
Strange: array (Colour, 2 .. 7, Weekday range Tue .. Thu)
                                  of Planet range Mars .. Saturn;
```

Note we said that the component type must be definite; the distinction between definite and indefinite types is explained in the next section.

Finally, Ada 2012 provides a much neater way of iterating over an array when similar actions are to be performed on each component of the array. So rather than

```
for I in A'Range loop
   A(I) := A(I) + 1.0;
end loop;
```

we can instead write

```
for E of A loop
   E := E + 1.0;
end loop;
```

The loop parameter E thus takes the identity of each component in turn. In the case of the two-dimensional array AA, rather than the double loop shown earlier we can set each component to zero by simply writing

```
for E of AA loop
   E := 0.0;
end loop;
```

and it iterates over the array in the expected manner. The index of the last dimension varies fastest matching the behaviour in the traditional version. Note the use of **of** rather then **in** in the loop specification.

This alternative iteration mechanism was introduced into Ada 2012 largely for improving iteration over containers as will be explained in Chapter 21 but it also applies to arrays for convenience and uniformity.

Exercise 8.1

1 Declare an array F of integers with index running from 0 to N. Write statements to set the components of F equal to the Fibonacci numbers given by

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2} \qquad i > 1$$

2 Write statements to find the index values I, J of the maximum component of

```
A: array (1 .. N, 1 .. M) of Float;
```

3 Declare an array Days_In_Month giving the number of days in each month. See Exercise 7.1(**1**). Use it to rewrite that example. See also Exercise 7.2(**1**).

4 Declare an array Yesterday analogous to the example Tomorrow above.

5 Declare a constant array Bor such that

```
Bor(P, Q) = P or Q
```

6 Declare a constant unit matrix Unit of order 3. A unit matrix is one for which all components are zero except those whose indices are equal which have value one.

8.2 Array types

The arrays we introduced in the previous section did not have an explicit type name. They were in fact of various anonymous types. This is one of the few cases in Ada where an object can be declared without naming the type – the other cases are access types, tasks and protected objects.

Reconsidering the first example in the previous section, we could write

```
type Vector_6 is array (1 .. 6) of Float;
A: Vector_6;
```

where we have declared A using the type name in the usual way.

An advantage of using a type name is that it enables us to assign whole arrays that have been declared separately. If we also have

```
B: Vector_6;
```

then we can write

```
B := A;
```

which has the effect of

```
B(1) := A(1);  B(2) := A(2);  ... B(6) := A(6);
```

although the order of assigning the components is not relevant.

On the other hand if we had written

```
C: array (1 .. 6) of Float;
D: array (1 .. 6) of Float;
```

then D := C; is illegal because C and D are not of the same type. They are of different types both of which are anonymous. The underlying rule of type equivalence is that every type definition introduces a new type and in this case the syntax tells us that an array type definition is the piece of text from **array** up to (but not including) the semicolon.

Moreover, even if we had written

```
C, D: array (1 .. 6) of Float;
```

then D := C; would still have been illegal. This is because of the rule mentioned in Section 6.1 that such a multiple declaration is only a shorthand for the two declarations above. There are therefore still two distinct type definitions even though they are not explicit.

Whether or not we introduce a type name for particular arrays depends very much on the abstract view of each situation. If we are thinking of the array as a complete object in its own right then we should use a type name. If, on the other hand, we are thinking of the array as merely an indexable conglomerate not related as a whole to other arrays then it should probably be of an anonymous type.

Arrays like Tomorrow and Work_Day of the previous section are good examples of arrays which are of the anonymous category. To be forced to introduce a type

name for such arrays would introduce unnecessary clutter and a possibly false sense of abstraction.

On the other hand, if we are manipulating lots of arrays of type Float of length 6 then there is a common underlying abstract type and so it should be named. The reader might also like to reconsider the example of Pascal's triangle in Section 2.4.

The model for array types introduced so far is still not satisfactory. It does not allow us to represent an abstract view that embraces the commonality between arrays which have different bounds but are otherwise of the same type. In particular, it would not allow the writing of subprograms which could take an array of arbitrary bounds as an actual parameter. So the concept of an unconstrained array type is introduced in which the constraints for the indices are not given. Consider

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

(The compound symbol <> is read as 'box'.)

This says that Vector is the name of a type which is a one-dimensional array of Float components with an Integer index. But the lower and upper bounds are not given; **range** <> is meant to convey the notion of information to be added later.

When we declare objects of type Vector we must supply the bounds. We can do this in various ways. We can introduce an intermediate subtype and then declare the objects.

```
subtype Vector_5 is Vector(1 .. 5);
V: Vector_5;
```

Or we can declare the objects directly

```
V: Vector(1 .. 5);
```

In either case the bounds are given by an index constraint which takes the form of a discrete range in parentheses. All the usual forms of discrete range can be used.

The index can also be given by a subtype name, thus

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

in which case the actual bounds of any declared object must lie within the range implied by the index subtype Positive (unless the object has a null range such as 1 .. 0). Note that the index subtype must be given by a subtype mark and not by a subtype indication; this avoids the horrid double use of **range** which could otherwise occur as in

```
type Nasty is array (Integer range 1 .. 100 range <>) of ... ;
```

We can now see that when we wrote

```
type Vector_6 is array (1 .. 6) of Float;
```

this was effectively a shorthand for

```
subtype index is Integer range 1 .. 6;
type anon is array (index range <>) of Float;
subtype Vector_6 is anon(1 .. 6);
```

Another useful array type declaration is

```
type Matrix is array (Integer range <>, Integer range <>) of Float;
```

And again we could introduce subtypes thus

```
subtype Matrix_3 is Matrix(1 .. 3, 1 .. 3);
M: Matrix_3;
```

or we could declare the objects directly

```
M: Matrix(1 .. 3, 1 .. 3);
```

An important point to notice is that an array subtype must give all the bounds or none at all. It would be perfectly legal to introduce an alternative name for Matrix by

```
subtype Mat is Matrix;
```

in which no bounds are given, but we could not have a subtype that just gave the bounds for one dimension but not the other.

In all of the cases we have been discussing, the ranges need not have static bounds. The bounds could be any expressions and are evaluated when the index constraint is encountered. We could have

```
M: Matrix(1 .. N, 1 .. N);
```

and then the upper bounds of M would be the value of N when M is declared. A range could even be null as would happen in the above case if N turned out to be zero. In this case the matrix M would have no components at all.

There is a further way in which the bounds of an array can be supplied; they can be taken from an initial value. Remember that all constants must have an initial value and variables may have one. The bounds can then be taken from the initial value if they are not supplied directly.

The initial value can be any expression of the appropriate type but will often be an aggregate as shown in the previous section. The form of aggregate shown there consisted of a list of expressions in parentheses. Such an aggregate is known as a positional aggregate since the values are given in position order. In the case of a positional aggregate used as an initial value and supplying the bounds, the lower bound is S'First where S is the subtype of the index. The upper bound is deduced from the number of components. (The bounds of positional aggregates in other contexts will be discussed in the next section.)

Suppose we had

```
type W is array (Weekday range <>) of Day;
Next_Work_Day: constant W := (Tue, Wed, Thu, Fri, Mon);
```

then the lower bound of the array is Weekday'First = Mon and the upper bound is Fri. It would not have mattered whether we had written Day or Weekday in the declaration of W because Day'First and Weekday'First are the same.

Note that we can also use the box notation with anonymous array types. So we can also write

```
Next_Work_Day: array (Weekday range <>) of Day :=
                                (Tue, Wed, Thu, Fri, Mon);
```

and there is no need to declare the intermediate type W. Again the bounds are deduced from the aggregate and we have also chosen not to make the array a constant.

Using initial values to supply the bounds needs care. Consider

```
Unit_2: constant Matrix := ((1.0, 0.0), (0.0, 1.0));
```

intended to declare a 2 2 unit matrix with Unit_2(1, 1) = Unit_2(2, 2) = 1.0 and Unit_2(1, 2) = Unit_2(2, 1) = 0.0.

But disaster! We have actually declared an array whose lower bounds are Integer'First which might be –32768 (2^{15}) or perhaps –2147483648 (2^{31}) or some such number, but is most certainly not 1.

If we declared the type Matrix as

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

then all would have been well since Positive'First = 1. So beware that array bounds deduced from an initial value may lead to nasty surprises.

We continue by returning to the topic of whole array assignment. In order to perform such assignment it is necessary that the array expression and the destination array variable have the same type and that the components can be matched. This does not mean that the bounds have to be equal, but merely that the number of components in corresponding dimensions is the same. In other words so that one array can be slid onto the other, giving rise to the term ‘sliding semantics’. So we can write

```
V: Vector(1 .. 5);
W: Vector(0 .. 4);
...
V := W;
```

Both V and W are of type Vector and both have five components.

Sliding may occur in several dimensions so the following is also valid

```
P: Matrix(0 .. 1, 0 .. 1);
Q: Matrix(6 .. 7, N .. N+1);
...
P := Q;
```

Equality and inequality of arrays follow similar sliding rules to assignment. Two arrays may only be compared if they are of the same type. They are equal if corresponding dimensions have the same number of components and the matching components are themselves equal. Note, however, that if the dimensions of the two arrays are not of the same length then equality will return False whereas an attempt to assign one array to the other will naturally cause Constraint_Error.

Although assignment and equality can only occur if the arrays are of the same type, nevertheless an array value of one type can be converted to another type if the index types are convertible and the component subtypes are statically the same. The usual notation for type conversion is used. So if we have

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

V: Vector(1 .. 5);
R: Row(0 .. 4);
```

then

```
R := Row(V);
```

is valid. In fact, since Row is unconstrained, the bounds of Row(V) are those of V. The normal assignment rules then apply. However, if the conversion uses a constrained type or subtype then the bounds are those of the type or subtype and the number of components in corresponding dimensions must be the same. Array type conversion is of particular value when subprograms from different libraries are used together as we shall see in Section 10.3.

Note that the component subtypes in a conversion must be statically the same – the technical term is that they must statically match. Remember that the component subtype could be constrained as in the array Strange at the end of the previous section. Static matching means that either the constraints are static and have the same bounds or else that the component subtypes come from the same elaboration. In this latter case they need not be static. So if we had

```
subtype S is Integer range 1 .. N;          -- N is some variable
type A1 is array (Index) of S;
type A2 is array (Index) of S;
type B1 is array (Index) of Integer range 1 .. N;
type B2 is array (Index) of Integer range 1 .. N;
```

then types A1 and A2 can be converted to each other but types B1 and B2 cannot. On the other hand, if we replaced N in all the declarations by a static expression such as 10 then all the subtypes would statically match and so all the array types could be converted to each other. The key point is that the matching has to be done statically (that is, at compile time) and not that the subtypes are necessarily static.

This is a good moment to explain the terms definite and indefinite. A definite subtype is one for which we can declare an object without an explicit constraint or initial value. Thus the subtype Vector_5 is definite. A type such as Vector on the other hand is indefinite since we cannot declare an object of the type without supplying the bounds either from an explicit constraint or from an initial value. Scalar types such as Integer are also definite. Other forms of indefinite type will be encountered in due course.

It will be recalled from Section 6.8 that a scalar subtype can have the aspect Default_Value so that objects of the type not explicitly initialized when declared will take that value rather than be undefined. The aspect Default_Component_Value can similarly be given for an array type whose components are of a scalar subtype.

Thus we might have

```
type Vector is array (Integer range <>) of Float
   with Default_Component_Value => 999.999;

subtype Vec is Vector
   with Default_Component_Value => 0.0;
```

The components of any object of type Vector are then 999.999 by default whereas in the case of the subtype Vec we have overridden the default to be zero.

We conclude this section by observing that the attributes First, Last, Length and Range, as well as applying to array objects, may also be applied to array types and subtypes provided they are constrained and so are definite. Hence Vector_6'Length is permitted and has the value 6 but Vector'Length is illegal.

Exercise 8.2

1 Declare an array type Bbb corresponding to the array Bor of Exercise 8.1(**5**).

2 Declare a two-dimensional array type suitable for operator tables on values of

```
subtype Ring5 is Integer range 0 .. 4;
```

Then declare addition and multiplication tables for modulo 5 arithmetic. Use the tables to write the expression (A + B) * C using modulo 5 arithmetic and assign the result to D where A, B, C and D have been appropriately declared. See Section 6.5.

8.3 Array aggregates

In the previous sections we introduced the idea of a positional aggregate. There is another form of aggregate known as a named aggregate in which the component values are preceded by the corresponding index value and =>. (The symbol => is akin to the 'pointing hand' sign used for indicating directions.) A simple example would be

```
(1 => 0.0, 2 => 0.0, 3 => 0.0, 4 => 0.0, 5 => 0.0, 6 => 0.0)
```

with the expected extension to several dimensions. The bounds of such an aggregate are self-evident and so our problem with the unit 2 2 matrix of the previous section could be overcome by writing

```
Unit_2: constant Matrix := (1 => (1 => 1.0, 2 => 0.0),
                            2 => (1 => 0.0, 2 => 1.0));
```

The rules for named aggregates are very similar to the rules for the alternatives in a case statement. Each choice can be given as a series of alternatives each of which can be a single value or a discrete range. We could therefore rewrite some previous examples as follows

```
A: array (1 .. 6) of Float := (1 .. 6 => 0.0);
```

```
Work_Day: constant array (Day) of Boolean :=
                                (Mon .. Fri => True, Sat | Sun => False);
```

In contrast to a positional aggregate, the index values need not appear in order. We could equally have written

```
(Sat | Sun => False, Mon .. Fri => True)
```

We can also use **others** but then as for the case statement it must be last and on its own (and there do not have to be any more values).

Array aggregates may not mix positional and named notation except that **others** may be used at the end of a positional aggregate.

It should also be realized that although we have been showing aggregates as initial values, they can be used quite generally in any place where an expression of an array type is required. They can also be the argument of type qualification.

The rules for deducing the bounds of an aggregate depend upon the form of the aggregate and its context. There are quite a lot of cases to consider and this makes the rules seem complicated although they are quite natural. We will first give the rules and then some examples of the consequences of the rules.

There are three kinds of aggregates to be considered

- Named without **others**, these have self-evident bounds.
- Positional without **others**, the number of elements is known but the actual bounds are not.
- Named or positional with **others**, neither the bounds nor the number of elements is known.

There are two main contexts to be considered according to whether the target type is constrained or unconstrained. In addition, the rules for qualification are special. So we have

- Unconstrained, gives no bounds.
- Constrained generally, gives the bounds, sliding usually permitted.
- Constrained qualification, gives the bounds, sliding never permitted.

The philosophy regarding sliding is that it is generally useful and so should be allowed and that aggregates should be no different from other array values in this respect; we saw some examples of sliding in the context of assignment in the previous section. Other contexts that behave like assignment will be met in due course when we discuss subprogram parameters and results in Chapter 9 and generic parameters in Chapter 17.

However, qualification is more in the nature of an assertion and so sliding is forbidden since it would be wrong to change the value in any way. If the bounds are not exactly the same then Constraint_Error is raised.

We will now consider the three kinds of aggregates in turn; the various combinations are summarized in Table 8.1.

A named aggregate without **others** has known bounds and will slide if permitted and necessary.

Table 8.1 Array aggregates and contexts.

Context	*Named*	*Positional*	*With* **others**
Unconstrained	OK bounds from aggregate	lower bound is S'First	illegal
Constrained *like* *assignment*	length must be same, could slide	length same, bounds from target	bounds from target
Constrained *qualification*	bounds must exactly match	length same, bounds from target	bounds from target

If a positional aggregate without **others** is used in a context which does not give the bounds then the lower bound is by default taken to be S'First where S is the index subtype and the upper bound is then deduced from the number of components. They never need to slide.

Aggregates with **others** are particularly awkward since we cannot deduce either the bounds or the number of elements. They can therefore only be used in a context that gives the bounds and consequently can never slide. Given the bounds, the components covered by **others** follow on from those given explicitly in the positional case and are simply those not given explicitly in the named case.

One way of supplying the bounds for an aggregate with **others** is to use qualification as we did to distinguish between overloaded enumeration literals. In order to do this we must have an appropriate (constrained) type or subtype name. So we might introduce

```
type Schedule is array (Day) of Boolean;
```

and can then write an expression such as

```
Schedule'(Mon .. Fri => True, others => False)
```

Note that when qualifying an aggregate we do not, as for an expression, need to put it in parentheses because it already has parentheses.

We have already considered the use of an aggregate as an initial value and providing the bounds in the example of Unit_2; this of course was an unconstrained context. An aggregate with **others** is thus not allowed. In the previous section we saw the effect of using a positional aggregate in such a context since it gave surprising bounds. In this section we saw how a named aggregate was more appropriate.

We will now consider the context of assignment which behaves much the same as a declaration with an initial value where the initial value is not being used to supply the bounds. These are constrained contexts and can therefore supply the bounds of an aggregate if necessary.

So both of the following are permitted

```
Work_Day: constant array (Day) of Boolean :=
            (Mon .. Fri => True, others => False);

Work_Day: constant array (Day) of Boolean :=
            (True, True, True, True, True, others => False);
```

Further insight might be obtained by another example. Consider

```
type Vector is array (Integer range <>) of Float;
V: Vector(1 .. 5) := (3 .. 5 => 1.0, 6 | 7 => 2.0);
```

which shows a named aggregate as the initial value of V. The bounds of the named aggregate are self-evident being 3 and 7 and so sliding occurs and the net result is that components V(1) .. V(3) have the value 1.0 and V(4) and V(5) have the value 2.0.

On the other hand, writing

```
V := (3 .. 5 => 1.0, others => 2.0);
```

has the rather different effect of setting V(3) .. V(5) to 1.0 and V(1) and V(2) to 2.0.

The point is that the bounds of the aggregate are taken from the context and there is no sliding. Aggregates with **others** never slide.

Similarly, no sliding occurs in

```
V := (1.0, 1.0, 1.0, others => 2.0);
```

and this results in setting V(1) .. V(3) to 1.0 and V(4) and V(5) to 2.0. It is clear that care is necessary when using **others**.

Array aggregates really are rather complicated and we still have a few points to make. The first is that in a named aggregate all the ranges and values before => must be static (as in a case statement) except for one situation. This is where there is only one alternative consisting of a single choice – it could then be a dynamic range or even a single dynamic value. An example might be

```
A: array (1 .. N) of Integer := (1 .. N => 0);
```

This is valid even if N is zero (or negative) and then gives a null array and a null aggregate. The following example illustrates a general rule that the expression after => is evaluated once for each corresponding index value; of course it usually makes no difference but consider

```
A: array (1 .. N) of Integer := (1 .. N => 1/N);
```

If N is zero then there are no values and so 1/N is not evaluated and Constraint_Error cannot occur. The reader will recall from Section 6.1 that a similar multiple evaluation also occurs when several objects are declared and initialized together.

In order to avoid awkward problems with null aggregates, a null choice is only allowed if it is the only choice. Foolish aggregates such as

```
(7 .. 6 | 1 .. 0 => 0)
```

are thus forbidden, and there is no question of the lower bound of such an aggregate.

Another point is that although we cannot mix named and positional notation within an aggregate, we can, however, use different forms for the different

components and levels of a multidimensional aggregate. So the initial value of the matrix Unit_2 could also be written as

```
(1 => (1.0, 0.0),      or      ((1 => 1.0, 2 => 0.0),
 2 => (0.0, 1.0))                (1 => 0.0, 2 => 1.0))
```

or even as

```
(1 => (1 => 1.0, 2 => 0.0),
 2 => (      0.0,       1.0))
```

and so on.

Note also that the Range attribute stands for a range and therefore can be used as one of the choices in a named aggregate. However, we cannot use the range attribute of an object in its own initial value. Thus

```
A: array (1 .. N) of Integer := (A'Range => 0);    -- illegal
```

is not allowed because an object is not visible until the end of its own declaration. However, we could write

```
A: array (1 .. N) of Integer := (others => 0);
```

and this is probably better than repeating 1 .. N because it localizes the dependency on N.

A further point is that a positional aggregate cannot contain just one component because otherwise it would be ambiguous – we could not distinguish it from a scalar value that happened to be in parentheses. An aggregate of one component must therefore always use the named notation. So

```
A: array (1 .. 1) of Integer := (99);          -- illegal
A: array (1 .. 1) of Integer := (1 => 99);     -- legal
```

Note also the following obscure case

```
A: array (N .. N) of Integer := (N => 99);
```

which shows a single choice and a single dynamic value being that choice.

The final feature is the ability to use <> as the expression with named notation in which case it means the default value. Thus we can write

```
P: array (1 .. 1000) of Integer := (1 => 2, 2 .. 1000 => <>);
```

The array P has its first component set to 2 and the others are undefined. It might be that P is to be used to hold the first 1000 prime numbers and the algorithm requires the first prime to be provided. The box cannot be used with positional notation but it is allowed with **others**. So we could write

```
(2, others => <>)
```

The box notation is not particularly useful with the types we have seen so far but it is very important with limited types which we will meet in Chapter 11.

The reader will by now have concluded that arrays in Ada are somewhat complicated. That is a fair judgement, but in practice there should be few

difficulties. There is always the safeguard that if we do something wrong, the compiler will inevitably tell us. In cases of ambiguity, qualification solves the problems provided we have an appropriate type or subtype name to use.

We conclude this section by pointing out that the named aggregate notation can greatly increase program legibility. It is especially valuable in initializing large constant arrays and guards against the accidental misplacement of individual values. Consider

```
type Event is (Birth, Accession, Death);
type Monarch is (William_I, William_II, Henry_I, ...,
                 Victoria, Edward_VII, George_V, ... );
...
Royal_Events: constant array (Monarch, Event) of Integer :=
    (William_I    => (1027, 1066, 1087),
     William_II   => (1056, 1087, 1100),
     Henry_I      => (1068, 1100, 1135),
     ...
     Victoria     => (1819, 1837, 1901),
     Edward_VII   => (1841, 1901, 1910),
     George_V     => (1865, 1910, 1936),
     ...                                 );
```

The accidental interchange of two lines of the aggregate causes no problems, whereas if we had just used the positional notation then an error would have been introduced and this might have been tricky to detect.

Exercise 8.3

1 Rewrite the declaration of the array Days_In_Month in Exercise 8.1(**3**) using a named aggregate for an initial value.

2 Declare a constant Matrix whose bounds are both 1 .. N where N is dynamic and whose components are all zero.

3 Declare a constant Matrix as in **2** but make it a unit matrix.

4 Declare a constant two-dimensional array which gives the numbers of each atom in a molecule of the various aliphatic alcohols. Declare appropriate enumeration types for both the atoms and the molecules. Consider methanol CH_3OH, ethanol C_2H_5OH, propanol C_3H_7OH and butanol C_4H_9OH.

8.4 Characters and strings

We now complete our discussion of enumeration types by introducing character types. In the enumeration types seen so far such as

```
type Colour is (Red, Amber, Green);
```

the values are represented by identifiers. It is also possible to have an enumeration type in which some or all of the values are represented by character literals.

A character literal is a further form of lexical element. It consists of a single character within a pair of single quotes. The character must be one of the graphic characters; this might be a space but it must not be a control character such as horizontal tabulate or newline.

This is a situation where there is a distinction between upper and lower case letters. Thus the character literals 'A' and 'a' are distinct.

So we could declare an enumeration type

```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
...
Dig: Roman_Digit := 'D';
```

All the usual properties of enumeration types apply.

```
Roman_Digit'First = 'I'
Roman_Digit'Succ('X') = 'L'
Roman_Digit'Pos('M') = 6
Dig < 'L' = False
```

There is a predefined enumeration type Character which is (naturally) a character type. We can think of its declaration as being of the form

```
type Character is (nul, ... , '0', '1', '2', ... , 'A', 'B', 'C', ... , 'a', 'b', 'c', ... , 'ÿ');
```

where the literals which are not graphic character literals (such as *nul*) are not really identifiers either (which is why they are represented here in italics). The predefined type Character represents the standard ISO 8-bit set, ISO 8859-1 commonly known as Latin-1. It describes the set of characters normally used for input and output and includes the various accented characters used in European languages, for example the last character is lower case y diaeresis; for the full declaration of type Character see Section 23.1.

It is possible to refer to the non-graphic characters as Ada.Characters.Latin_1.Nul and so on (or with a suitable use clause as Latin_1.Nul or simply Nul). We can also refer to the graphic characters (other than digits and the normal 26 upper case letters) by name; this is useful for displaying program text on output devices which do not support all the graphic characters. Finally, we can refer to those characters also in the 7-bit ASCII set as ASCII.Nul and so on but this feature is obsolescent.

There are also predefined types Wide_Character (16-bit) and Wide_Wide_Character (32-bit) corresponding to the Basic Multilingual Plane (BMP) and the full set of ISO 10646 respectively. The first 256 positions of Wide_Character correspond to those of the type Character and the first 65536 positions of Wide_Wide_Character correspond to those of the type Wide_Character.

The existence of these predefined types results in overloading of some of the literals. So an expression such as

```
'X' < 'L'
```

is ambiguous. We do not know whether it is comparing characters of the type Character, Wide_Character or Wide_Wide_Character (or even Roman_Digit). In order to resolve the ambiguity we must qualify one or both literals thus

```
Character'('X') < 'L'                    -- False
Roman_Digit'('X') < 'L'                  -- True
```

As well as the predefined type Character there is also the predefined array type String

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

This is a perfectly normal array type and obeys all the rules of the previous section. So we can write

```
S: String (1 .. 7);
```

to declare an array of range 1 .. 7. The bounds can also be deduced from the initial value thus

```
G: constant String := ('P', 'I', 'G');          -- bounds are 1 and 3
```

where the initial value takes the form of a normal positional aggregate. The lower bound of G (that is, G'First) is 1 since the index subtype of String is Positive and Positive'First is 1.

There is another notation for positional aggregates whose components are character literals. This is the string literal. So we can more conveniently write

```
G: constant String := "PIG";
```

The string literal is the last lexical element to be introduced. It consists of a sequence of printable characters and spaces enclosed in double quotes. A double quote may be represented in a string by two double quotes so that

```
('A', '"', 'B')      is the same as    "A""B"
```

The string may also have just one character or may be null. The equivalent aggregates using character literals have to be written in named notation.

```
(1 => 'A')           is the same as    "A"
(1 .. 0 => <>)       is the same as    ""
```

The box character avoids introducing an arbitrary character in the null case.

Another rule about a lexical string is that it must fit onto a single line. Moreover it cannot contain control characters such as *soh*. And, of course, as with character literals, the two cases of alphabet are distinct in strings.

In Section 8.6 we shall see how to overcome the limitations that a string must fit onto a single line and yet cannot contain control characters.

A major use for strings is for creating text to be output. A simple sequence of characters can be output by a call of the (overloaded) subprogram Put. Thus

```
Put("The Countess of Lovelace");
```

will output the text

```
The Countess of Lovelace
```

onto some appropriate file.

There are also types Wide_String and Wide_Wide_String defined as arrays of the corresponding character types. One consequence is that comparisons between literal strings are also ambiguous like comparisons between individual literals and so also have to be qualified as illustrated in Section 8.6.

However, the lexical string is not reserved just for use with the predefined types String, Wide_String and Wide_Wide_String. It can be used to represent an array of any character type. We can write

```
type Roman_Number is array (Positive range <>) of Roman_Digit;
```

and then

```
Nineteen_Eighty_Four: constant Roman_Number := "MCMLXXXIV";
```

or indeed

```
Four: array (1 .. 2) of Roman_Digit := "IV";
```

Of course the compiler knows nothing of our interpretation of Roman numbers and the reason for our choice of identifiers!

Exercise 8.4

1 Declare a constant array Roman_To_Integer which can be used for table look-up to convert a Roman_Digit to its normal integer equivalent (e.g. converts 'C' to 100).

2 Given an object R of type Roman_Number write statements to compute the equivalent integer value V. It may be assumed that R obeys the normal rules of construction of Roman numbers.

8.5 Arrays of arrays and slices

The components of an array can be of any definite subtype. Remember that a definite subtype is one for which we can declare an object (without an explicit constraint or initial value). Thus we can declare arrays of any scalar type; we can also declare arrays of arrays. So we can have

```
type Matrix_3_6 is array (1 .. 3) of Vector_6;
```

where, as in Section 8.2

```
type Vector_6 is array (1 .. 6) of Float;
```

However, we cannot declare an array type whose component subtype is indefinite such as an unconstrained array (just as we cannot declare an object which is an unconstrained array). So we cannot write

```
type Matrix_3_N is array (1 .. 3) of Vector;    -- illegal
```

On the other hand, nothing prevents us declaring an unconstrained array of constrained arrays thus

```
type Matrix_N_6 is array (Integer range <>) of Vector_6;
```

It is instructive to compare the differences between declaring an array of arrays

```
AOA: Matrix_3_6;      or      AOA: Matrix_N_6(1 .. 3);
```

and the similar multidimensional array

```
MDA: Matrix(1 .. 3, 1 .. 6);
```

Aggregates for both are completely identical, for example

```
((1.0, 2.0, 3.0, 4.0, 5.0, 6.0),
 (4.0, 4.0, 4.0, 4.0, 4.0, 4.0),
 (6.0, 5.0, 4.0, 3.0, 2.0, 1.0))
```

but component access is quite different, thus

```
AOA(I)(J)
MDA(I, J)
```

where in the case of AOA the internal structure is naturally revealed. The individual rows of AOA can be manipulated as arrays in their own right, but the structure of MDA cannot be decomposed. So we could change the middle row of AOA to zero by

```
AOA(2) := (1 .. 6 => 0.0);
```

but a similar technique cannot be applied to MDA.

Arrays of arrays are not restricted to one dimension, we can have a multi-dimensional array of arrays or an array of multidimensional arrays; the notation extends in an obvious way.

Arrays of strings are revealing. Consider

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

which is an unconstrained two-dimensional array type. We can then declare

```
Farmyard: constant String_Array := ("pig", "cat", "dog",
                                    "cow", "rat", "hen");
```

where the bounds are conveniently deduced from the aggregate. But we cannot have a ragged array where the individual strings are of different lengths such as

```
Zoo: constant String_Array := ("aardvark", "baboon",
          "camel", "dolphin", "elephant", ..., "zebra");      -- illegal
```

This is a real nuisance and means we have to pad out the strings with spaces so that they are all the same length

```
Zoo: constant String_Array := ("aardvark    ",
                               "baboon     ",
                               "camel      ",
                               ...
                               "zebra      ");
```

The next problem is that we cannot select an individual one of the strings. We might want to output a particular one and so perhaps attempt

```
Put(Farmyard(5));
```

hoping to print the text

```
rat
```

but this is not allowed since we can only select an individual component of an array and, in the case of this two-dimensional array, this is just one character.

An alternative approach is to use an array of arrays. A problem here is that the component in the array type declaration has to be constrained and so we have to decide on the length of our strings right from the beginning thus

```
type String_3_Array is array (Positive range <>) of String(1 .. 3);
```

and then

```
Farmyard: constant String_3_Array := ("pig", "cat", "dog",
                                      "cow", "rat", "hen");
```

With this formulation we can indeed select an individual string as a whole and so the statement Put(Farmyard(5)); now works. However, we still cannot declare our Zoo as a ragged array; we will return to this topic in Section 11.2 when another approach will be discussed.

We thus see that arrays of arrays and multidimensional arrays each have their own advantages and disadvantages.

A special feature of one-dimensional arrays is the ability to denote a slice of an array object. A slice is written as the name of the object (variable or constant) followed by a discrete range in parentheses.

So given

```
S: String(1 .. 10);
```

then we can write S(3 .. 8) to denote the middle six characters of S. The bounds of the slice are the bounds of the range and not those of the index subtype. We could write

```
T: constant String := S(3 .. 8);
```

and then T'First = 3, T'Last = 8.

The bounds of the slice need not be static but can be any expressions. A slice would be null if the range turned out to be null.

The use of slices emphasizes the nature of array assignment. The value of the expression to be assigned is completely evaluated before any components are assigned. No problems arise with overlapping slices. So

```
S(1 .. 4) := "BARA";
S(4 .. 7) := S(1 .. 4);
```

results in S(1 .. 7) = "BARBARA". S(4) is only updated after the expression S(1 .. 4) is safely evaluated. There is no risk of setting S(4) to 'B' and then consequently making the expression "BARB" with the final result of

```
"BARBARB"
```

The ability to use slices is another consideration in deciding between arrays of arrays and multidimensional arrays. With our second Farmyard we can write

```
Pets: String_3_Array(1 .. 2) := Farmyard(2 .. 3);
```

which uses sliding assignment so that the two components of Pets are "cat" and "dog". Moreover, if we had declared the Farmyard as a variable rather than a constant then we could also write

```
Farmyard(1)(1 .. 2) := "ho";
```

which turns the "pig" into a "hog"! We can do none of these things with the old Farmyard.

Exercise 8.5

1 Write a single assignment statement to swap the first two rows of AOA.

2 Declare the second Farmyard as a variable. Then change the cow into a sow.

3 Assume that R contains a Roman number. Write statements to see if the last digit of the corresponding decimal Arabic value is a 4; if so change it to a 6.

8.6 One-dimensional array operations

Many of the operators that we met in Chapter 6 may also be applied to one-dimensional arrays.

The operators **and**, **or**, **xor** and **not** may be applied to one-dimensional Boolean arrays. For the binary operators, the two operands must have the same length and be of the same type. The underlying scalar operation is applied to matching components and the resulting array is again of the same type. The bounds of the result are the same as the bounds of the left or only operand.

Consider the following declarations and assignments

```
type Bit_Row is array (Positive range <>) of Boolean;
A, B: Bit_Row(1 .. 4);
C, D: array (1 .. 4) of Boolean;
T: constant Boolean := True;
F: constant Boolean := False;
...
A := (T, T, F, F);
B := (T, F, T, F);

A := A and B;
B := not B;
```

The result is that A now equals (T, F, F, F), and B equals (F, T, F, T). But note that C **and** D would not be allowed since C and D are of different (and anonymous) types because of the rules regarding type equivalence mentioned in Section 8.2. This is clearly a case where it is appropriate to give a name to the array type because we are manipulating the arrays as complete objects.

Note that these operators also use sliding semantics, like assignment as explained in Section 8.2, and so only demand that the types and the number of components be the same. The bounds themselves do not have to be equal. However, if the number of components are not the same then, naturally, Constraint_Error will be raised.

Boolean arrays can be used to represent sets. Consider

```
type Primary is (R, Y, B);
type Colour is array (Primary) of Boolean;
C: Colour;
```

then there are 8 = 2 2 2 values that C can take. C is, of course, an array with three components and each of these has value True or False; the three components are

C(R), C(Y) and C(B)

The 8 possible values of the type Colour can be represented by suitably named constants as follows

```
White:  constant Colour := (F, F, F);
Red:    constant Colour := (T, F, F);
Yellow: constant Colour := (F, T, F);
Blue:   constant Colour := (F, F, T);
Green:  constant Colour := (F, T, T);
Purple: constant Colour := (T, F, T);
Orange: constant Colour := (T, T, F);
Black:  constant Colour := (T, T, T);
```

and then we can write expressions such as

```
Red or Yellow
```

which is equal to Orange and

```
not Black
```

which is White.

So the values of our type Colour are effectively the set of colours obtained by taking all combinations of the primary colours represented by R, Y and B. The empty set is the value of White and the full set is the value of Black. We are using the paint pot mixing colour model rather than light mixing. A value of True for a component means that the primary colour concerned is mixed in our pot. The murky mess we got at junior school from mixing too many colours together is our black!

The operations **or**, **and** and **xor** may be interpreted as set union, set intersection and symmetric difference. A test for set membership can be made by inspecting the value of the appropriate component of the set. Thus

```
C(R)
```

is True if R is in the set represented by C. We cannot use the predefined operation **in** for this. A literal value can be represented using the named aggregate notation, so we might denote Orange by

```
(R | Y => T, others => F)
```

A more elegant way of doing this will appear in the next chapter.

We now consider the relational operators. The equality operators = and /= apply to almost all types anyway and we gave the rules for arrays when we discussed assignment in Section 8.2.

The ordering operators <, <=, > and >= may be applied to one-dimensional arrays of a discrete type. (Note discrete.) The result of the comparison is based upon the lexicographic (that is, dictionary) order using the predefined order relation for the components. Remembering that the upper and lower case letters are distinct and the upper case ones occur earlier in the type Character, then for the type String the following strings are in lexicographic order

```
"" , "A" , "AZZ" , "CAT" , "CATERPILLAR" , "DOG" , "cat"
```

Strings are compared component by component until they differ in some position. The string with the lower component is then lower. If one string runs out of components as in CAT versus CATERPILLAR then the shorter one is lower. The null string is lowest of all.

Because of the existence of the types Wide_String and Wide_Wide_String we cannot actually write comparisons such as

```
"CAT" < "DOG"          -- illegal
"CCL" < "CCXC"         -- illegal
```

because they are ambiguous since we do not know whether we are comparing type String, Wide_String, Wide_Wide_String or even Roman_Number. We must qualify one or both of the strings. This is done in the usual way but a string, unlike an ordinary aggregate, has to be placed in extra parentheses otherwise we would get an ugly juxtaposition of a single and double quote. So

```
Wide_String'("CAT") < "DOG"            -- True
String'("CCL") < "CCXC"                -- True
Roman_Number'("CCL") < "CCXC"          -- False
```

Note that the compiler is too stupid to know about the interpretation of Roman numbers in our minds and has said that 250 < 290 is false. The only thing that matters is the order relation of the characters 'L' and 'X' in the type definition. In the next chapter we will see how we can redefine < so that it works ‘properly’ for Roman numbers.

The ordering operators also apply to general expressions and not just to literal strings

```
Nineteen_Eighty_Four < "MM"                -- True
```

The ordering operators can be applied to arrays of any discrete types. So

```
(1, 2, 3) < (2, 3)
(Jan, Jan) < (1 => Feb)
```

The predefined operators <=, > and >= are defined by analogy with <.

We finally introduce a new binary operator & which denotes concatenation of one-dimensional arrays. It has the same precedence as binary plus and minus. The two operands must be of the same type and the result is an array of the same type whose value is obtained by juxtaposing the two operands. The length of the result is thus the sum of the lengths of the operands.

The lower bound of the result depends upon whether the underlying array type is constrained or not. If it is unconstrained (considered the usual case) then the lower bound is that of the left operand as for other operators. However, if it is constrained then the lower bound is that of the array index subtype. (If the left operand is null the result is simply the right operand.)

So

```
"CAT" & "ERPILLAR" = "CATERPILLAR"
```

Concatenation can be used to construct a string which will not fit on one line

```
"This string goes " &
"on and on"
```

One or both operands of & can also be a single value of the component type. If the left operand is such a single value then the lower bound of the result is always the lower bound of the array index subtype.

```
"CAT" & 'S' = "CATS"
'S' & "CAT" = "SCAT"
'S' & 'S' = "SS"
```

This is useful for representing the control characters such as CR and LF in strings. So, using an abbreviated form rather than Ada.Characters.Latin_1.CR, we can write

```
"First line" & Latin_1.CR & Latin_1.LF & "Next line"
```

Of course, it might be neater to declare

```
CRLF: constant String := (Latin_1.CR, Latin_1.LF);
```

and then write

```
"First line" & CRLF & "Next line"
```

The operation & can be applied to any one-dimensional array type and so we can apply it to our Roman numbers. Consider

```
R: Roman_Number(1 .. 5);
S: String(1 .. 5);

R := "CCL" & "IV";
S := "CCL" & "IV";
```

This is valid. The context tells us that in the first case we apply & to two Roman numbers whereas in the second we apply it to two values of type String. There is no ambiguity as in

```
B: Boolean := "CCL" < "IV";                    -- illegal
```

where the context does not distinguish the various string types.

Exercise 8.6

1 Put the eight possible constants White ... Black of the type Colour in ascending order as determined by the operator < applied to one-dimensional arrays.

2 Evaluate

(a) Red **or** Green
(b) Black **xor** Red
(c) **not** Green

3 Show that **not** (Black **xor** C) = C is true for all values of C.

4 Why did we not write

```
(Jan, Jan) < (Feb)
```

5 Put in ascending order the following values of type String: "ABC", "123", "abc", "Abc", "abC", "aBc".

6 Given

```
C: Character;
S: String(5 .. 10);
```

What are the lower bounds of

(a) C & S (b) S & C (c) "" & S

7 Given

```
type TC is array (1 .. 10) of Integer;
type TU is array (Natural range <>) of Integer;
AC: TC;
AU: TU(1 .. 10);
```

What are the bounds of

(a) AC(6 .. 10) & AC(1 .. 5)
(b) AC(6) & AC(7 .. 10) & AC(1 .. 5)
(c) AU(6 .. 10) & AU(1 .. 5)
(d) AU(6) & AU(7 .. 10) & AU(1 .. 5)

8.7 Records

As stated at the beginning of this chapter, we are only going to consider the simplest form of record at this point. Discussions of tagged and discriminated records will be found in Chapters 14 and 18 respectively.

A record is a composite object consisting of named components which may be of different types. In contrast to arrays, we cannot have anonymous record types – they all have to be named. Consider

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

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

This declares the type Date to be a record containing three named components: Day, Month and Year.

We can declare variables and constants of record types in the usual way.

```
D: Date;
```

declares an object D which is a date. The individual components of D can be denoted by following D with a dot and the component name. Thus we could write

```
D.Day := 4;
D.Month := Jul;
D.Year := 1776;
```

in order to assign new values to the individual components.

Records can be manipulated as whole objects. Literal values can be written as aggregates much like arrays; both positional and named forms can be used. So we could write

```
D: Date := (4, Jul, 1776);
E: Date;
```

and then

```
E := D;
```

or

```
E := (Month => Jul, Day => 4, Year => 1776);
```

The reader will be relieved to know that much of the complexity of array aggregates does not apply to records. This is because the number of components is always known.

In a positional aggregate the components come in order. In a named aggregate they may be in any order.

A named aggregate cannot use a range because the components are not considered to be closely related. The vertical bar and **others** can be used but of course the expression must be appropriate for all the components covered.

An extra possibility for records is that the positional and named notations can be mixed in one aggregate. But if this is done then the positional components must come first and in order (without holes) as usual. So in other words, we can change to the named notation at any point in the aggregate but must then stick to it. The above date could therefore also be expressed as

```
(4, Jul, Year => 1776)
(4, Year => 1776, Month => Jul)
```

and so on.

It is possible to give default expressions for some or all of the components in the type declaration. Note that there is no need for an aspect corresponding to Default_Value for scalars and Default_Component_Value for arrays. In the case of records, we simply write

```
type Complex is
   record
      Re: Float := 0.0;
      Im: Float := 0.0;
   end record;
```

or more succinctly

```
type Complex is
   record
      Re, Im: Float := 0.0;
   end record;
```

declares a record type containing two components of type Float and gives a default expression of 0.0 for each. This record type represents a complex number $x + iy$ where Re and Im are the values of x and y. The default value thus represents (0, 0), the origin of the Argand plane. We can now declare

```
C1: Complex;
C2: Complex := (1.0, 0.0);
```

The object C1 will now have the values 0.0 for its components by default. In the case of C2 we have explicitly overridden the defaults. Note that even if there are default expressions, an aggregate must always be complete in order to provide so-called full coverage analysis. This ensures that if we extend a type with additional components, then we have to modify all aggregates to match and so cannot inadvertently forget some components.

In this example both components have the same type and so the following named forms are also possible

```
(Re | Im => 1.0)
(others => 1.0)
```

We can also use the box notation with record aggregates. Thus we might declare

```
C3: Complex := (Re => 1.0, Im => <>);
```

in which case the component Re takes the explicitly given value whereas the component Im takes its default value (if any). Remember that the box can only be used with named notation although since we can mix named and positional notation in record aggregates we could also write (1.0, Im => <>).

A minor point is that we can write **others** => <> even if the components covered are of different types (or there are no more components) since no expression is involved. We can also use the vertical bar with the box even if the components are of different types for the same reason.

The only operations predefined on record types are = and /= as well as assignment (unless the type is limited as discussed in Chapter 12). Other operations must be performed at the component level or be explicitly defined by a subprogram as we shall see in the next chapter.

A record type may have any number of components. It may even have none in which case we must either write the component list as **null**; or use the abbreviated form

```
type Hole is null record;
```

The aggregate for the (null) value of such a type can be written as (**null record**) or even as (**others** => <>).

As in the case of arrays, an aggregate for a record with just one component must always use named notation in order to avoid confusion with an expression in parentheses.

The components of a record type can be of any definite type; they can be other records or arrays. However, if a component is an array then it must be fully constrained and moreover it must be of a named type and not an anonymous type. And obviously a record cannot contain an instance of itself.

The components cannot be constants but the record as a whole can be. Thus

```
I: constant Complex := (0.0, 1.0);
```

is allowed and represents the square root of –1.

A more elaborate example of a record is given by

```
type Subject is (Theology, Classics, Mathematics);
type Scores is array (Subject) of Integer range 0 .. 100;

type Student is
   record
      Birth: Date;
      Finals: Scores := (others => 0);
   end record;
```

The record Student has two components, the first is another record, a Date, the second an array giving examination results in various subjects. The array has default values of zeros.

We can now write

```
Fred: Student;
Fred.Birth := (19, Aug, 1984);
Fred.Finals := (5, 15, 99);
```

and this would be equivalent to

```
Fred := (Birth => (19, Aug, 1984), Finals => (5, 15, 99));
```

The notation is as expected. It is better to use an aggregate rather than a series of assignments to individual components since it ensures that they are all given a value. If another component is added later and we fail to update an aggregate then the compiler will tell us.

For components of objects we proceed from left to right using the dot notation to select components of a record and indices in parentheses to select components of an array and ranges in parentheses to slice arrays. There is no limit. We could have an array of students

```
People: array (1 .. N) of Student;
```

and then statements such as

```
People(6).Birth.Day := 19;
People(8).Finals(Classics .. Mathematics) := (50, 50);
```

A final point concerns the evaluation of expressions in a record declaration. An expression in a constraint applied to a component is evaluated when the record type is elaborated. Suppose the type Student also has a component

```
Name: String(1 .. N) := (others => ' ');
```

then the length of the component Name will be the value of N when the type Student is elaborated. Of course, N need not be static and so if the type declaration is in a loop, for example, then each execution of the loop might give rise to a type with a different size component. However, for each elaboration of the record type declaration all objects of the type will have the same component size.

On the other hand, a default expression in a record type is only evaluated when an object of the type is declared and only then if no explicit initial value is provided. Of course, in simple cases, like our type Complex, it makes no difference but it could bring surprises. For example suppose we write the component Name as

```
Name: String(1 .. N) := (1 .. N => ' ');
```

then the length of the component Name is the value of N when the record type is declared whereas when a Student is subsequently declared without an initial value, the aggregate will be evaluated using the value of N which then applies. Of course, N may by then be different and so Constraint_Error will be raised. This is rather surprising; we do seem to have strayed into an odd backwater of Ada!

Exercise 8.7

1 Declare three variables C1, C2 and C3 of type Complex. Write one or more statements to assign (a) the sum, (b) the product, of C1 and C2 to C3.

2 Write statements to find the index of the first student of the array People born on or after 1 January 1980.

Checklist 8

Array types can be anonymous, but record types cannot.

Aggregates must always be complete.

Distinguish constrained array types (definite types) from unconstrained array types (those with <>).

The component subtype of an array must be definite.

Named and positional notations cannot be mixed for array aggregates – they can for records.

An array aggregate with **others** must have a context giving its bounds. It never slides.

A choice in an array aggregate can only be dynamic or null if it is the only choice.

The attributes First, Last, Length and Range apply to array objects and constrained array types and subtypes but not to unconstrained types and subtypes.

For array assignment to be valid, the number of components must be equal for each dimension – not the bounds.

The cases of alphabet are distinct in character literals and strings.

An aggregate with one component must use the named notation. This applies to records as well as to arrays.

A record component cannot be of an anonymous array type.

A default component expression is only evaluated when an uninitialized object is declared.

The <> in aggregates can only be used with named notation.

The types Wide_Wide_Character and Wide_Wide_String were added in Ada 2005.

The use of <> in aggregates meaning the default value was added in Ada 2005.

New in Ada 2012

Ada 2012 provides an alternative notation for iterating over arrays.

The aspect Default_Component_Value is new in Ada 2012.

9 Expression Structures

This short chapter describes additional forms of expressions introduced in Ada 2012. These are related to the three bracketed sequential control structures of Ada described in the previous chapter. Thus, corresponding to the if statement there is the if expression and corresponding to the case statement there is the case expression. There are also quantified expressions and these are related to loop statements. We also discuss the new forms of membership tests in Ada 2012.

It is perhaps very surprising that Ada did not have conditional forms of expressions right from the beginning. Even Algol 60 had conditional expressions. However, perhaps there was a feeling that conditional expressions could lead one astray by loosening the crisp distinction between statements and expressions.

The incentive to introduce conditional expressions into Ada was triggered by the addition of pre- and postconditions which are described in Chapter 16. Without adding conditional forms of expressions these pre- and postconditions would have been cumbersome and required lots of small functions. However, these additional forms of expressions are applicable in other situations as well and it seems appropriate to describe them here.

9.1 Membership tests

Membership tests were introduced in Section 6.9 where we showed how they could be used to check whether a value lies within a specified range or satisfies a constraint implied by a subtype.

Thus given I of type Integer we could write

```
if I in 1.. 10 then ...
```

and given the type Day describing all the days of the week and the subtype Weekday we could test whether a variable D of the type Day had a value within the subtype by writing

```
if D in Weekday then ...
```

These were the only forms of membership test allowed in Ada 2005, but more general forms are permitted in Ada 2012.

Suppose we have an enumeration type for the names of months and perhaps related subtypes thus

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

subtype Spring is Month_Name range March .. May;
```

and wish to check whether it is safe to eat an oyster (there has to be an R in the month). In Ada 2005 we can write

```
if M in Jan .. April or M in Sep .. Dec then
```

which means repeating M and then perhaps worrying about whether to use **or** or **or else**. Alternatively we might write

```
if M not in May .. August then
```

but this seems somewhat unnatural.

However, Ada 2012 permits a membership test to have several possible membership choices separated by the vertical bar so we can more naturally write

```
if M in Jan .. April | Sep .. Dec then
```

The individual membership choices can be single expressions, subtypes, or ranges. Thus the following are all permitted

```
if N in 6 | 28 | 496 then      -- N is small and perfect

if M in Spring | June | October .. December then
                               -- combination of subtype, single value, range

if X in 0.5 .. Z | 2.0*Z .. 10.0 then      -- not discrete or static

if Obj in Triangle | Circle then           -- with tagged types

if Letter in 'A' | 'E' | 'I' | 'O' | 'U' then      -- characters
```

Membership tests are permitted for any type and values do not have to be static. However, it should be remembered that uses of the vertical bar in case statements and aggregates do require the type to be discrete and the values to be static.

Another important point about membership tests is that the membership choices (that is the items separated by the bars) are evaluated in order. As soon as one is found such that the value of the test is known to be true (or false if **not** is present) then the test as a whole is determined and the other membership choices are not evaluated. This is therefore the same as using short circuit forms such as **or else**.

It is often convenient to use a membership test before a conversion to ensure that the conversion will succeed. This avoids raising an exception which then has to be handled. Thus we might have

```
subtype Score is Integer range 1 .. 60;
Total: Integer;
S: Score;
...                          -- compute Total somehow
if Total in Score then
   S := Score(Total);        -- reliable conversion
   ...                       -- now use S knowing that it is OK
else
   ...                       -- Total was excessive
end if;
```

If we are indexing some arrays whose range is Score then it is an advantage to use S as an index since we know it will work and no checks are needed.

There are other uses for membership tests which we will encounter in due course. For example, Ada 2012 permits the use of a membership test to check accessibility which is described in Chapter 11.

An important issue is that the whole purpose of a membership test X **in** S is to find out whether a condition is satisfied. We want the result of a test to be true or false and not to raise an exception. However, if the evaluation of S could itself raise an exception then we have done something silly; this is discussed in detail in Section 16.5.

Exercise 9.1

1 Assume that N is a positive integer. Assign true or false to B according to whether or not the value of N is a prime less than 20.

2 Assume that Letter is a value of the type Character. Write a test to see whether Letter is one of the first five or last five letters of the alphabet.

9.2 If expressions

A simple example of an if expression was illustrated in Section 7.1 when we noted that we could write

```
Tomorrow := (if Today = Sun then Mon else Day'Succ(Today));
```

rather than using an if statement.

Another situation where an if expression is useful is if we have alternative calls of the same subprogram but with just one parameter being different. Thus we might wish to call a procedure P thus

```
if X > 0 then
   P(A, B, D, E);
else
   P(A, C, D, E);
end if;
```

This is cumbersome and in Ada 2012 we can simply write

```
P(A, (if X > 0 then B else C), D, E);
```

Note that there is no closing **end if**. One reason is simply that it is logically unnecessary since there can only be a single expression after **else** and also **end if** would be obtrusively heavy. However, in order to aid clarity, an if expression is always enclosed in parentheses. If the context already has parentheses then additional ones are not necessary. Thus in the case of a procedure call with a single parameter, we can just write

```
P(if X > 0 then B else C);
```

However, if the call uses named notation (see Section 10.4) then additional parentheses are needed as in

```
P(Para => (if X > 0 then B else C));
```

As expected, a series of tests can be done using **elsif** thus

```
P(if X > 0 then B elsif X < 0 then C else D);
```

and expressions can be nested

```
P(if X > 0 then (If Y > 0 then B else C) else D);
```

Without the rule requiring enclosing parentheses this could be written as

```
P(if X > 0 then If Y > 0 then B else C else D);
```

which seems more than a little confusing.

There is a special rule if the type of the expression is Boolean. In that case a final else part can be omitted and is taken to be true by default. Thus the following are equivalent

```
Q(if C1 then C2 else True);
Q(if C1 then C2);
```

Such abbreviations occur frequently in preconditions which are discussed in Chapter 16. If we write

```
Pre => (if P1 > 0 then P2 > 0)
```

then this has the obvious meaning that the precondition requires that if P1 is positive then P2 must also be positive. However, if P1 is not positive then the precondition is true and it doesn't matter about P2.

Curiously enough the two expressions

```
(if C1 then C2)
(not C1 or C2)
```

have exactly the same value and are in fact the **imp** (implies) operator which is present in some languages. The truth table for this phantom operator is shown in Figure 9.1.

imp	F	T
F	T	F
T	T	T

Figure 9.1 Operator table for the non-existent operator **imp**.

There are important rules regarding the types of the various dependent expressions in the branches of an if expression. Basically, they have to all be of the same type or convertible to the same expected type. But there are some interesting situations.

If the expression is the argument of a type conversion then effectively the conversion is considered pushed down to the dependent expressions. Thus

```
X := Float(if P then A else B);
```

is equivalent to

```
X := (if P then Float(A) else Float(B));
```

As a consequence we can write

```
X := Float(if P then 27 else 0.3);
```

and it does not matter that 27 and 0.3 are not of the same type.

Similar situations arise with other conversions. Using the examples of Section 3.3 we might declare a variable V of a class wide type such as Object'Class. See also Section 14.2. We can initialize V with a value of a type derived from Object such as

```
V: Object'Class := A_Circle;
```

where A_Circle is an object of type Circle which is derived from the type Object.

The initial value can also be given by a conditional expression such as

```
V: Object'Class := (if B then A_Circle else A_Triangle);
```

where A_Circle and A_Triangle are objects of specific types Circle and Triangle which are themselves derived from the root type Object. Thus the individual expressions do not have to be of the same specific type provided that they are all derived from the same root type. Effectively, the implicit conversion is pushed down to each dependent expression.

If the expected type is a specific tagged type then the rules for the various branches are similar to the rules for calling a subprogram with several controlling operands as described in Section 14.4. Briefly, they all have to be dynamically tagged (that is class wide) or all have to be statically tagged; they might all be tag indeterminate in which case the conditional expression as a whole is tag indeterminate.

Some obscure situations arise. Remember that the controlling expression of an if statement can be any Boolean type. Now consider

```
type My_Boolean is new Boolean;
My_Cond: My_Boolean := ...
if (if K > 10 then X = Y else My_Cond) then                    -- illegal
   ...
end if;
```

The problem here is that X = Y is of type Boolean but My_Cond is of type My_Boolean. Moreover, the expected type for the condition in the if statement is any Boolean type so the poor compiler cannot make up its mind. This foolishness could be overcome by putting a type conversion around the if expression.

Similar rules regarding the various dependent parts of an if expression apply to staticness. If all parts are static then the expression as a whole is static.

The initial value of a named number has to be static (see Section 6.1). So if we wish to set the initial value of a named number Febdays to 29 or 28 and there is a static Boolean Leap indicating whether it is a leap year or not then we can write

```
Febdays: constant := (if Leap then 29 else 28);
```

Attempting to do this in earlier versions of Ada was awkward. One had to write something horrid like

```
Febdays: constant := Boolean'Pos(Leap)*29 + Boolean'Pos(not Leap)*28;
```

which is truly gruesome. Similar disgusting expressions might also be contrived for the call of the procedure where only one parameter was different. Thus we could write

```
P(A, Boolean'Pos(X>0)*B + Boolean'Pos(X<=0)*C, D, E);
```

Static expressions introduce situations where all parts of an expression need not be evaluated. Consider

```
X := (If B then P else Q);
```

If B, P, and Q are all static then the expression as a whole is static as mentioned above. If B is true then the answer is P and there is not any need to even look at Q. We say that Q is statically unevaluated and indeed it does not matter that if Q had been evaluated then it would have raised an exception.

If we write

```
Answer := (if Count = 0 then 0.0 else Total/Count);
```

then naturally, having evaluated Count and finding it to be zero, then we know that the answer is 0.0 and we do not have to evaluate Total/Count so there is no risk of dividing by zero. Similar situations arise with short circuit conditions described in Section 6.9.

Exercise 9.2

1 Assign to Days_In_Month the number of days in a month M. Assume that the year is in the range 1901 .. 2099 as in Exercise 7.1(**1**).

9.3 Case expressions

Case expressions have much in common with if expressions and the two are collectively known as conditional expressions.

Thus given a variable D of the familiar type Day, we can assign the number of hours in a working day to the variable Hours by

```
Hours := (case D is
            when Mon .. Thurs => 8.0,
            when Fri => 6.0,
            when Sat | Sun => 0);
```

A slightly more adventurous example taking full account of leap years and involving nested if expressions is

```
Days_In_Month := (case M is
                    when September | April | June | November => 30,
                    when February =>
                      (if Year mod 100 = 0 then
                         (if Year mod 400 = 0 then 29 else 28)
                       else
                         (if Year mod 4 = 0 then 29 else 28)),
                    when others => 31);
```

The reader is invited to improve this!

Note the similarity to the rules for if expressions. There is no closing **end case**. Case expressions are similarly always enclosed in parentheses but they can be omitted if the context already provides parentheses such as in a subprogram call with a single positional parameter.

The inner structure is just like that of a case statement. The individual choices must be static and an optional **others** clause may be last. The rules for the expression after **case** are the same as well and of course all values of the appropriate subtype must be covered.

If M and Year are static then the case expression as a whole is static. If M is static and equal to September, April, June or November then the value is statically known to be 30 so that the expression for February is not evaluated even if Year is not static. (Again we say that the expression is statically unevaluated.) Note that the various choices are evaluated in order.

The rules regarding the types of the dependent expressions are exactly as for if expressions. Thus if the case expression is the argument of a type conversion then the conversion is effectively pushed down to all the dependent expressions.

It is always worth emphasizing that an important advantage of case constructions is that they give a coverage check. Thus suppose we have an enumeration type describing various animals

```
type Animal is (Bear, Cat, Dog, Horse, Wolf);
```

Note that some of these animals might be considered to be pets and thus need feeding whereas others might be considered to be pests and should not be fed (what a difference a single s makes!).

So if A is an animal and Feed_It is a Boolean then we might write

```
Feed_It := A in Cat | Dog | Horse;
```

using the more elaborate form of membership test described in Section 9.1. However, this is unwise since if we add Rabbit to the type Animal then (presuming it is a pet and not wild) then it will not be fed unless we remember to add Rabbit to the membership test.

It is better to write

```
Feed_It := (case A is
              when Cat | Dog | Horse => True,
              when Bear | Wolf => False);
```

and then if we add Rabbit to the type Animal then we must add it to one arm of the case expression otherwise it will fail to compile.

It is interesting to look for examples of case statements to see whether they can more conveniently be rewritten using case expressions. For example, the procedure Build_List of Program 1, Magic Moments, has a case statement whose purpose is to assign a value of the type Object'Class to Object_Ptr according to a character typed by the user. However, if the character is incorrect the effect is to exit the surrounding loop. One could rewrite this as

```
Object_Ptr := (case Code_Letter is
                 when 'C' | 'c' =>          -- expect a circle
                   Get_Circle
                 when 'T' | 't' =>          -- expect a triangle
                   Get_Triangle
                 when 'S' | 's' =>          -- expect a square
                   Get_Square
                 when others =>
                   null);
if Object_Ptr = null then exit; end if;
```

This certainly captures the expected behaviour that all sensible branches assign to Object_Ptr but makes the exceptional case more awkward. We will discuss this example again in Section 15.2 when we introduce raise expressions.

Note that conditional expressions can themselves be part of a larger expression. Thus we might have an if expression within a case expression and so on.

Exercise 9.3

1 Write a statement using a case expression to assign the length of the name of Today to the integer L. Thus if today is Monday the value to be assigned is 6. Do not use any attributes.

2 A sexist and ageist company provides a pension for all staff aged 60 and over when they retire. The basic pension is 500 euros per month and is incremented by 50 euros at age 70 and 80. No pension is given to those aged over 100. It is reduced by 10% for females and increased by 5% for those who are disabled. A

special terminal bonus of 100 per month is given to all in the year when they are 100. Given variables Age, Gender and Disabled write an expression giving their monthly pension to the nearest euro and assign it to a variable Pension of type Integer. Consider whether it is best to use a case expression or if expression for each part of the calculation.

9.4 Quantified expressions

Another new form of expression in Ada 2012 is the quantified expression. Quantified expressions are closely related to for loops and might be considered to be a sort of loop expression. However, the type of a quantified expression is always a Boolean type.

As a simple example consider

```
B := (for all K in A'Range => A(K) = 0);
```

which assigns true to B if every component of the array A has value 0.

A quantified expression always starts with **for** and is then followed by a quantifier which in this case is the reserved word **all**. The other possible quantifier is **some** so we might instead have

```
B := (for some K in A'Range => A(K) = 0);
```

which assigns true to B if some component of the array A has value 0. Note that **some** is a new reserved word in Ada 2012 and in fact is the only new reserved word in Ada 2012.

The expression after the => is always a Boolean expression. It is known technically as the predicate which is simply a fancy term used in logic to mean a truth value which in Ada we refer to as a Boolean expression.

Note that the loop parameter is almost inevitably used in the predicate. A quantified expression is very much like a for statement except that we evaluate the expression after => on each iteration rather than executing one or more statements. The iteration is somewhat implicit and the words **loop** and **end loop** do not appear.

The expression is evaluated for each iteration in the appropriate order (**reverse** can be inserted of course) and the iteration stops as soon as the value of the expression is determined. Thus in the case of **for all**, as soon as one value is found to be False, the overall expression is False whereas in the case of **for some** as soon as one value is found to be True, the overall expression is True. An iteration could raise an exception which would then be propagated in the usual way.

Like conditional expressions, a quantified expression is always enclosed in parentheses which can be omitted if the context already provides them, such as in a procedure call with a single positional parameter.

The forms **for all** and **for some** are technically known as the universal quantifier and existential quantifier respectively.

Note that in mathematics we use the symbols ∀ and ∃ to mean 'for all' and 'there exists'. Thus we might write

$$\forall\ x, \exists\ y,\ \text{s.t.}\ x + y = 0$$

which means of course that for all values x, there exists a value y such that $x+y$ equals zero. In other words, every value has a corresponding negative (although it does not say it is unique).

Readers might like to contemplate whether the symbols ∀ and ∃ are the usual letters A and E inverted and reversed respectively or perhaps simply rotated. (Casting one's mind back to the days of hot lead it is clear that they are just rotated.)

The type of a quantified expression in Ada can be any Boolean type (that is the predefined type Boolean or perhaps My_Boolean derived from Boolean). The predicate must be of the same type as the expression as a whole. Thus if the predicate is of type My_Boolean then the quantified expression is also of type My_Boolean.

Quantified expressions can be nested. So we might check that all components of a two-dimensional array AA are zero by writing

```
B := (for all I in AA'Range(1) =>
        (for all J in AA'Range(2) => AA(I, J) = 0));
```

This can be done rather more neatly using the iterator form using **of** rather than **in** as mentioned in Section 8.1. We just write

```
B := (for all E of AA => E = 0);
```

which iterates over all elements of the array AA however many dimensions it has.

Of course, we cannot always use the abbreviated form. Thus suppose we wanted to see if an array A has every component equal to its index value. The quantified expression describing this might be

```
(for all I in A'Range => A(I) = I);
```

and this cannot be rewritten in the other form because the predicate involves the index value other than for indexing the array.

Quantified expressions were introduced into Ada 2012 primarily for use in preconditions and postconditions which will be discussed in Chapter 16. However, they can be used in any context requiring an expression. Thus we might test whether an integer N is prime by

```
RN := Integer(Sqrt(Float(N)));
if (for some K in 2 .. RN => N mod K = 0) then    ...   -- N not prime
```

or we might reverse the test by

```
if (for all K in 2 .. RN => N mod K /= 0) then    ...   -- N is prime
```

Beware that this is not a recommended technique if N is at all large!

We could also use a quantified expression for checking that it is safe to eat an oyster which we did using a membership test in Section 9.1. Thus we might write

```
if (for some C of Month_Name'Image(M) => C = 'R' or C = 'r') then
```

which assumes that we have given the full names January, February, and so on for the enumeration literals in the type Month_Name.

Exercise 9.4

1 The array A of integers has been sorted into ascending order. Write a quantified expression describing this. Permit duplication.

Checklist 9

Membership choices can be expressions, ranges and subtypes.
Membership choices need not be static.
If expressions, case expressions and quantified expressions are always in parentheses.
Conditional expressions do not have **end if** or **end case**.
The choices in a case expression must be static.
All possibilities in a case expression must be catered for.
If **others** is used it must be last and on its own.
The expression after **case** can be qualified in order to reduce the alternatives.
This chapter does not apply to Ada 2005.

New in Ada 2012

This chapter is all new in Ada 2012.

10 Subprograms

Subprograms are perhaps the oldest form of abstraction and existed long before the introduction of high level languages. Subprograms enable a unit of code to be encapsulated and thereby reused and also enable the code to be parameterized so that it can be written without knowing the actual data to which it is to be applied.

In Ada, subprograms fall into two categories: functions and procedures. Functions are called as components of expressions and return a value as part of the expression, whereas procedures are called as statements standing alone.

As we shall see, the actions to be performed when a subprogram is called are usually described by a subprogram body. Subprogram bodies are declared in the usual way in a declarative part which may for instance be in a block or indeed in another subprogram.

10.1 Functions

A function is a form of subprogram that can be called as part of an expression. In Chapter 6 we met examples of calls of functions such as Day'Succ and Sqrt.

We now consider the form of a function body which describes the statements to be executed when the function is called. For example the body of the function Sqrt might have the form

```
function Sqrt(X: Float) return Float is
   R: Float;
begin
   -- compute value of Sqrt(X) in R
   return R;
end Sqrt;
```

All function bodies start with the reserved word **function** and the designator of the function being defined. If the function has parameters the designator is followed by a list of parameter specifications in parentheses. Each gives the identifiers of one or more parameters, a colon and then its type or subtype. If there are several such specifications then they are separated by semicolons. Examples of parameter lists are

```
(I, J, K: Integer)
(Left: Integer; Right: Float)
```

The parameter list, if any, is then followed by **return** and the type or subtype of the result of the function. In the case of both parameters and result the type or subtype must be given by a subtype mark and not by a subtype indication with an explicit constraint; the reason will be mentioned in Section 10.7. (But we can give a null exclusion which concerns access types and will be discussed in Chapter 11.)

The part of the body we have described so far is called the function specification. It specifies the function to the outside world in the sense of providing all the information needed to call the function.

After the specification comes **is** and then the body proper which is just like a block – it has a declarative part, **begin**, a sequence of statements, and then **end**. As in the case of a block, the declarative part can be empty, but there must be at least one statement in the sequence of statements. Between **end** and the terminating semicolon we may repeat the designator of the function. This is optional but, if present, must correctly match the designator after **function**.

It is often necessary or just convenient to give the specification on its own but without the rest of the body. In such a case it is immediately followed by a semicolon thus

```
function Sqrt(X: Float) return Float;
```

and is then correctly known as a function declaration – although often still informally referred to as a specification. The uses of such declarations will be discussed in Section 10.7.

In general, the parameters of a subprogram can be of three modes, **in**, **in out** and **out**. The modes are described in Section 10.3 when we discuss procedures. Up until Ada 2005, functions could only have parameters of mode **in** and the discussion in this section assumes that all parameters are of mode **in.**

The formal parameters (of mode **in**) of a function act as local constants whose values are provided by the corresponding actual parameters. When the function is called the declarative part is elaborated in the usual way and then the statements are executed. A return statement is used to indicate the value of the function call and to return control back to the calling expression.

Thus considering our example suppose we had

```
S := Sqrt(T + 0.5);
```

then first T + 0.5 is evaluated and then Sqrt is called. Within the body the parameter X behaves as a constant with the initial value given by T + 0.5. It is rather as if we had

```
X: constant Float := T + 0.5;
```

The declaration of R is then elaborated. We then obey the sequence of statements which we assume compute the square root of X and assign it to R. The last statement is **return** R; this passes control back to the calling expression with the result of the function being the value of R. This value is then assigned to S.

The expression in a return statement can be of arbitrary complexity and must be of the same type as and satisfy any constraints implied by the subtype mark given in the function specification (and any null exclusion, see Section 11.3). If the constraints are violated then the exception Constraint_Error is raised. (A result of an array type can slide as mentioned later in this section.)

A function body may have several return statements. The execution of any one of them will terminate the function. Thus the function Sign which takes an integer value and returns +1, 0 or −1 according to whether the parameter is positive, zero or negative could be written as

```
function Sign(X: Integer) return Integer is
begin
   if X > 0 then
      return +1;
   elsif X < 0 then
      return −1;
   else
      return 0;
   end if;
end Sign;
```

So we see that the last lexical statement of the body need not be a return statement since there is one in each branch of the if statement. Any attempt to 'run' into the final end will raise the exception Program_Error. This is our first example of a situation giving rise to Program_Error; this exception is generally used for situations which would violate the run-time control structure.

Each call of a function produces a new instance of any objects declared within it (including parameters of course) and these disappear when we leave the function. It is therefore possible for a function to be called recursively without any problems. So the factorial function could be declared as

```
function Factorial(N: Positive) return Positive is
begin
   if N = 1 then
      return 1;
   else
      return N * Factorial(N−1);
   end if;
end Factorial;
```

If we write

```
F := Factorial(4);
```

then the function calls itself until, on the fourth call (with the other three calls all partly executed and waiting for the result of the call they did before doing the

multiply) we find that N is 1 and the calls then all unwind and all the multiplications are performed.

Note that there is no need to check that the parameter N is positive since the parameter is of the subtype Positive. So calling Factorial(−2) will result in Constraint_Error. Of course, Factorial(10_000) could result in the computer running out of space in which case Storage_Error would be raised. The more moderate call Factorial(50) would undoubtedly cause overflow and thus raise Constraint_Error.

A formal parameter may be of any type but in general the type must have a name. The one exception is access parameters which are discussed in Chapter 11. So a parameter cannot be of an anonymous type such as

```
array (1 .. 6) of Float
```

In any event no actual parameter (other than an aggregate) could match such a formal parameter even if it were allowed since the actual and formal parameters must have the same type and the rules of type equivalence require that it must be named. Again access parameters are slightly different.

A formal parameter can be of an unconstrained array type such as

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

In such a case the bounds of the formal parameter are taken from those of the actual parameter.

Consider

```
function Sum(A: Vector) return Float is
   Result: Float := 0.0;
begin
   for I in A'Range loop
      Result := Result + A(I);
   end loop;
   return Result;
end Sum;
```

then we can write

```
V: Vector(1 .. 4) := (1.0, 2.0, 3.0, 4.0);
S: Float;
...
S := Sum(V);
```

The formal parameter A then takes the bounds of the actual parameter V. So for this call we have

```
A'Range    is    1 .. 4
```

and the effect of the loop is to compute the sum of A(1), A(2), A(3) and A(4). The final value of Result which is returned and assigned to S is therefore 10.0.

The function Sum can therefore be used to sum the components of a vector with any bounds and in particular where the bounds are not known until the program executes.

A function could have a constrained array subtype as a formal parameter. However, remember that we cannot apply the constraint in the parameter list as in

```
function Sum_5(A: Vector(1 .. 5)) return Float      -- illegal
```

but must use the name of a constrained array type or subtype thus

```
subtype Vector_5 is Vector(1 .. 5);
...
function Sum_5(A: Vector_5) return Float
```

An actual parameter corresponding to such a constrained formal array must have the same number of components; sliding is allowed as for assignment. So we could have

```
W: Vector(0 .. 4);
...
S := Sum_5(W);
```

The actual parameter of a function can also be an aggregate (including a string). In fact the behaviour is exactly as for an initial value described in Section 8.3. If the formal parameter is unconstrained then the aggregate must supply its bounds and so cannot contain **others**. If the formal parameter is constrained then it provides the bounds; an aggregate without **others** could slide. But remember that an aggregate with **others** never slides.

As another example consider

```
function Inner(A, B: Vector) return Float is
   Result: Float := 0.0;
begin
   for I in A'Range loop
      Result := Result + A(I)*B(I);
   end loop;
   return Result;
end Inner;
```

This computes the inner product of the two vectors A and B by adding together the sum of the products of corresponding components. This is an example of a function with more than one parameter. Such a function is called by following the function name by a list of expressions giving the values of the actual parameters separated by commas and in parentheses. The order of evaluation of the actual parameters is not defined.

So

```
V: Vector(1 .. 3) := (1.0, 2.0, 3.0);
W: Vector(1 .. 3) := (2.0, 3.0, 4.0);
X: Float;
...
X := Inner(V, W);
```

results in X being assigned the value

```
1.0 * 2.0 + 2.0 * 3.0 + 3.0 * 4.0 = 20.0
```

Note that the function Inner is not written well since it does not check that the bounds of A and B are the same. It is not symmetric with respect to A and B since I takes (or tries to take) the values of the range A'Range irrespective of B'Range. So if the array W had bounds of 0 and 2, Constraint_Error would be raised on the third time round the loop. If the array W had bounds of 1 and 4 then an exception would not be raised but the result might not be as expected.

It would be nice to ensure the equality of the bounds by placing a constraint on B at the time of call but this cannot be done. The best we can do is simply check the bounds for equality inside the function body and perhaps explicitly raise Constraint_Error if they are not equal

```
if A'First /= B'First or A'Last /= B'Last then
   raise Constraint_Error;
end if;
```

(The use of the raise statement is described in detail in Chapter 15.)

We saw above that a formal parameter can be of an unconstrained array type. Similarly, a function result can be an array whose bounds are not known until the function is called. The result type can be an unconstrained array and the bounds are then obtained from the expression in the return statement.

As an example the following function returns a vector which has the same bounds as the parameter but whose component values are in the reverse order

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

The variable R is declared to be of type Vector with the same bounds as A. Note how the loop reverses the value. The result takes the bounds of the expression R. Sadly, the function is called Rev rather than Reverse because **reverse** is a reserved word.

The matching rules for results of both constrained and unconstrained arrays are the same as for parameters. Sliding is allowed and so on.

If a function returns a record or array value then a component can be immediately selected, indexed or sliced as appropriate without assigning the value to a variable. Indeed the result is treated as a (constant) object in its own right. So

```
Rev(Y)(I)
```

denotes the component indexed by I of the array returned by the call of Rev.

It should be noted that a parameterless function call, like a parameterless procedure call, has no parentheses. There is thus a possible ambiguity between calling a function with one parameter and indexing the result of a parameterless call; such an ambiguity could be resolved by, for example, renaming the functions as will be described in Section 13.7.

We conclude this section by discussing the other form of return statement which we mention here for completeness although its importance will not be obvious until

we deal with matters such as limited types in Section 11.5 and tasks in Section 22.2. This is the extended return statement. We could rewrite the last example as follows

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

An extended return statement is bracketed between **return** and **end return**. After **return** there is the declaration of a variable which is to be the result; it can but need not have an initial value. This declaration is then followed by the reserved word **do** and then a sequence of statements and finally the closing **end return**. Running into the **end return** causes control to pass back to the function call with the value of the variable (in this case R) as the result.

An extended return statement can include any statements except another extended return statement. It can include blocks and thus internal declarations and so on. It can also include a simple return statement which then causes control to be returned. But such a simple return statement must not have an expression since the result to be returned is given by the variable of the extended return statement itself. So the structure might be

```
return R: T := E do
   if ... then
      ...
      return;                    -- returns R
   end if;
   ...
end return;
```

The return object R can be marked as **aliased** (see Section 11.4) and as **constant** for uniformity with other declarations.

The example of the function Rev illustrates that although the return object must be constrained, the type given in the specification need not be constrained. Thus the specification has the unconstrained array type Vector, but the extended return statement declares R to be of the constrained subtype Vector(A'Range). This necessary constraint could also be given by the initial value of R. A similar situation occurs with class wide types as discussed in Section 14.2.

A function could have several extended return statements perhaps in the branches of if or case statements. The specification might give the result as being unconstrained whereas individual branches might return results with different constraints. An example with discriminants will be found in Section 18.3 where the function Frankenstein has the return type Person but individual branches return objects of type Man or Woman. Similarly, the result type might be a class wide type such as Object'Class introduced in Section 3.3 but individual branches might return a Circle, a Triangle, a Square, and so on.

If we leave an extended return statement directly by a goto or exit statement then this does not cause a return from the function.

A variation is that the **do** ... **end return** part can be omitted so we might have

```
function Zero return Complex is
begin
   return C: Complex;
end Zero;
```

in which case the initial value of the variable C is the result. If the type Complex is as in Section 8.7 then the returned value will be (0.0, 0.0) because that is the default value of the type Complex.

As mentioned earlier, the extended return statement is vital in certain situations but for the moment we can treat it as simply an alternative syntax. From the point of view of style it makes it clear from the beginning what is to be returned and ensures that we cannot forget to return the object.

In Ada 2012, if a function is short then it can often be written as an expression function in which there is no return statement but whose result is given simply by an expression in parentheses. Expression functions are important in contracts (see Chapter 16). Their practicality is increased by the introduction of conditional and quantified expressions described in Chapter 9.

For example, the function Sign described above can be rewritten in Ada 2012 as simply

```
function Sign(X: Integer) return Integer is
   (if X > 0 then +1 elsif X < 0 then –1 else 0);
```

As mentioned earlier, in versions of Ada upto Ada 2005, functions could only have parameters of mode **in**. However, in Ada 2012 they can have parameters of any modes. The various modes are described in Section 10.3. One consequence of this is that Ada 2012 introduces various rules concerning aliasing as described in Section 10.4.

Exercise 10.1

1 Write a function Even which returns True or False according to whether its Integer parameter is even or odd.

2 Rewrite the factorial function so that the parameter may be positive or zero but not negative. Remember that the value of Factorial(0) is to be 1. Use the subtype Natural introduced in Section 6.5.

3 Write a function Outer that forms the outer product of two vectors. The outer product $\boldsymbol{C}$ of two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ is a matrix such that $C_{ij} = A_i B_j$.

4 Write a function Make_Colour which takes an array of values of type Primary and returns the corresponding value of type Colour. See Section 8.6. Check that Make_Colour((R, Y)) = Orange.

5 Rewrite the function Inner to use sliding semantics so that it works providing the arrays have the same length. Raise Constraint_Error (as outlined above) if the arrays do not match. Use an extended return statement.

6 Write a function Make_Unit that takes a single parameter N and returns a unit $N \times N$ matrix with components of type Float. Use the function to declare a constant unit $N \times N$ matrix. See Exercise 8.3(**3**).

7 Write a function GCD to return the greatest common divisor of two nonnegative integers. Use Euclid's algorithm that

$$\gcd(x, y) = \gcd(y, x \bmod y) \quad y \neq 0$$
$$\gcd(x, 0) = x$$

Write the function using recursion and then rewrite it using a loop statement.

10.2 Operators

In the previous section we stated that a function body commenced with the reserved word **function** followed by the designator of the function. In all the examples of that section the designator was in fact an identifier. However, it can also be a character string provided that the string is one of the language operators in double quotes. These are

abs	**and**	**mod**	**not**	**or**	**rem**	**xor**
=	/=	<	<=	>	>=	
+	–	*	/	**	&	

In such a case the function defines a new meaning of the operator concerned. As an example we can rewrite the function Inner of the previous section as an operator thus

```
function "*" (A, B: Vector) return Float is
   Result: Float := 0.0;
begin
   for I in A'Range loop
      Result := Result + A(I)*B(I);
   end loop;
   return Result;
end "*";
```

We call this new function by the normal syntax of uses of the operator "*". Thus instead of

```
X := Inner(V, W);
```

we now write

```
X := V * W;
```

This meaning of "*" is distinguished from the existing meanings of integer and floating point multiplication by the context provided by the types of the actual parameters V and W and the type of the destination X.

The giving of several meanings to an operator is another instance of overloading which we have already met with enumeration literals. The rules for the

overloading of subprograms in general are discussed later in this chapter. It suffices to say at this point that any ambiguity can usually be resolved by qualification. Overloading of predefined operators is not new. It has existed in most programming languages for the past sixty years.

We can now see that the predefined meanings of all operators are as if there were a series of functions with declarations such as

```
function "+" (Left, Right: Float) return Float;
function "<" (Left, Right: Float) return Boolean;
function "<" (Left, Right: Boolean) return Boolean;
```

Moreover, every time we declare a new type, new overloadings of predefined operators such as "=" and "<" may be created.

Observe that the predefined operators always have Left and Right as formal parameter names (real mathematicians would prefer X and Y which have served the community well since the days of Newton; but Ada had to be awkward!).

Although we can add new meanings to operators we cannot change the syntax of the call. Thus the number of parameters of "*" must always be two and the precedence cannot be changed and so on. The operators "+" and "–" are unusual in that a new definition can have either one parameter or two parameters according to whether it is to be called as a unary or binary operator. Thus the function Sum could be rewritten as

```
function "+" (A: Vector) return Float is
   Result: Float := 0.0;
begin
   for I in A'Range loop
      Result := Result + A(I);
   end loop;
   return Result;
end "+";
```

and we would then write

```
S := +V;
```

rather than

```
S := Sum(V);
```

Function bodies whose designators are operators often contain interesting examples of uses of the operator being overloaded. Thus the body of "*" contains a use of "*" in A(I)*B(I). There is, of course, no ambiguity since the expressions A(I) and B(I) are of type Float whereas our new overloading is for type Vector. Sometimes there is the risk of accidental recursion. This particularly applies if we try to replace an existing meaning rather than add a new one.

Apart from the operator "/=" there are no special rules regarding the types of the operands and results of new overloadings. Thus a new overloading of "=" need not return a Boolean result. On the other hand, if it is Boolean then a corresponding new

overloading of "/=" is implicitly created. Moreover, explicit new overloadings of "/=" are also allowed provided only that the result type is not Boolean.

The membership tests **in** and **not in** and the short circuit forms **and then** and **or else** cannot be given new meanings. That is why we said in Section 6.9 that they were not technically classed as operators.

In the case of operators represented by reserved words, the characters in the string can be in either case. Thus a new overloading of **or** can be declared as "or" or "OR" or even "Or".

In the previous section, it was mentioned that Ada 2012 introduced expression functions and permitted functions to have parameters of any mode. These extensions do not apply to operators.

Exercise 10.2

1 Write a function "<" that operates on two Roman numbers and compares them according to their corresponding numeric values. That is, so that "CCL" < "CCXC". See Exercise 8.4(**2**).

2 Write functions "+" and "*" to add and multiply two values of type Complex. See Exercise 8.7(**1**).

3 Write a function "<" to test whether a value of type Primary is in a set represented by a value of type Colour. See Section 8.6.

4 Write a function "<=" to test whether one value of type Colour is a subset of another.

5 Write a function "<" to compare two values of the type Date of Section 8.7.

10.3 Procedures

The other form of subprogram is a procedure; a procedure is called as a statement standing alone in contrast to a function which is always called as part of an expression. We have already encountered a number of examples of procedure calls where there are no parameters such as Work; Party; Action; and so on. We now look at procedures in general.

The body of a procedure is very similar to that of a function. The differences are

- a procedure starts with **procedure**,
- its name must be an identifier,
- it does not return a result.

In Ada 2012, functions as well as procedures can have parameters of any mode. There are three different modes **in**, **out** or **in out**. In earlier versions of Ada, functions could only have parameters of mode **in**.

The mode of a parameter is indicated by following the colon in the parameter specification by **in** or by **out** or by **in out**. If the mode is omitted then it is taken to be **in**. We usually give the mode **in** for procedures but omit it for functions. The

form of parameter known as an access parameter is actually an **in** parameter. Access parameters are discussed with access types in Chapter 11.

In the case of functions; the examples earlier in this chapter omitted **in** but could have been written, for instance, as

```
function Sqrt(X: in Float) return Float;
function "*" (A, B: in Vector) return Float;
```

The general effect of the three modes can be summarized as follows.

in The formal parameter is a constant initialized by the value of the associated actual parameter.

in out The formal parameter is a variable initialized by the actual parameter; it permits both reading and updating of the value of the associated actual parameter.

out The formal parameter is an uninitialized variable; it permits updating of the value of the associated actual parameter.

Note that both **in out** and **out** parameters behave as normal variables within the subprogram but the key difference is that an **in out** parameter is always initialized by the actual parameter whereas an **out** parameter is not.

The fine detail of the behaviour depends upon whether a parameter is passed by copy or by reference. Parameters of scalar types (and access types, see Chapter 10) are always passed by copy and we will consider them first.

As a simple example of the modes **in** and **out** consider

```
procedure Add(A, B: in Integer; C: out Integer) is
begin
   C := A + B;
end Add;
```

with

```
P, Q: Integer;
...
Add(2+P, 37, Q);
```

On calling Add, the expressions 2+P and 37 are evaluated (in any order) and are the initial values of the formals A and B which behave as constants. The value of A+B is then assigned to the formal variable C. On return the value of C is assigned to the variable Q. Thus it is (more or less) as if we had written

```
declare
   A: constant Integer := 2+P;      -- in
   B: constant Integer := 37;       -- in
   C: Integer;                      -- out
begin
   C := A + B;                      -- body
   Q := C;                          -- out
end;
```

As an example of the mode **in out** consider

```
procedure Increment(X: in out Integer) is
begin
   X := X + 1;
end;
...
I: Integer;
...
Increment(I);
```

On calling Increment, the value of I is the initial value of the formal variable X. The value of X is then incremented. On return, the final value of X is assigned to the actual parameter I. So it is rather as if we had written

```
declare
   X: Integer := I;
begin
   X := X + 1;
   I := X;
end;
```

For any scalar type (such as Integer) the modes thus correspond simply to copying the value **in** at the call or **out** upon return or both in the case of **in out**.

If the mode is **in** then the actual parameter may be any expression of the appropriate type or subtype. If the mode is **out** or **in out** then the actual parameter must be a variable (it could not be a constant or an aggregate for example). The identity of such a variable is determined when the subprogram is called and cannot change during the call.

Suppose we had

```
I: Integer;
A: array (1 .. 10) of Integer;

procedure Silly(X: in out Integer) is
begin
   I := I + 1;
   X := X + 1;
end;
```

then the statements

```
A(5) := 1;
I := 5;
Silly(A(I));
```

result in A(5) becoming 2, I becoming 6, but A(6) is not affected.

If a parameter is an array or record then the mechanism of copying, described above, may generally be used but alternatively an implementation may use a reference mechanism in which the formal parameter provides direct access to the actual parameter. A program which depends on the particular mechanism because

of aliasing is said to have a bounded error. See the exercises at the end of this section.

Parameters of certain types (and arrays and records with any components of those types) are always passed by reference. It so happens that we have not yet dealt with any of these types which are: task and protected types (Chapter 20), tagged record types (Chapter 14) and explicitly limited types (Chapter 12). Private types behave as the corresponding full type (Chapter 12).

Note that because a formal array parameter takes its bounds from the actual parameter, the bounds are always copied in at the start even in the case of an **out** parameter. Of course, for simplicity, an implementation could always copy in the whole array anyway. Indeed, when **out** parameters are passed by copy certain types are always copied in so that dangerous undefined values do not arise. This applies to access types (Chapter 11), discriminated record types (Chapter 18) and record types which have components with default initial values such as the type Complex in Section 8.7.

We now consider the question of constraints on parameters; these are similar to the rules for function results which were mentioned in Section 10.1.

In the case of scalar parameters the situation is as expected from the copying model. For an **in** or **in out** parameter any constraint on the formal must be satisfied by the value of the actual at the beginning of the call. Conversely for an **in out** or **out** parameter any constraint on the variable which is the actual parameter must be satisfied by the value of the formal parameter upon return from the subprogram. Any constraint imposed by the result of a function must also be satisfied.

In the case of arrays the situation is somewhat different. If the formal parameter is a constrained array type, the association is just as for assignment, the number of components in each dimension must be the same but sliding is permitted. If, on the other hand, the formal parameter is an unconstrained array type, then, as we have seen, it takes its bounds from those of the actual. The foregoing applies irrespective of the mode of the array parameter. Similar rules apply to function results; if the result is a constrained array type then the expression in the result can slide. Moreover, beware that passing results of unconstrained array types may be inefficient.

In the case of the simple records we have discussed so far there are no constraints and so there is nothing to say. The parameter and result mechanism for other types will be discussed in detail when they are introduced.

We noted above that an actual parameter corresponding to a formal **out** or **in out** parameter must be a variable. This allows the actual parameter in turn to be an **out** or **in out** formal parameter of some outer subprogram.

A further possibility is that an actual parameter can also be a type conversion of a variable provided, of course, that the conversion is allowed. As an example, since conversion is allowed between numeric types, we can write

```
F: Float;
...
Increment(Integer(F));
```

If F initially had the value 2.3, it would be converted to the integer value 2, incremented to give 3 and then on return converted to 3.0 and finally assigned back to F.

This conversion (technically known as a view conversion) of **in out** or **out** parameters is particularly useful with arrays. Suppose we write a library of subprograms applying to our type Vector and then acquire from someone else some subprograms written to apply to the type Row of Section 8.2. The types Row and Vector are essentially the same; it just so happened that the authors used different names. Array type conversion allows us to use both sets of subprograms without having to change the type names systematically.

As a final example consider the following

```
procedure Quadratic(A, B, C: in Float;
                    Root_1, Root_2: out Float; OK: out Boolean) is
   D: constant Float := B**2 – 4.0*A*C;
begin
   if D < 0.0 or A = 0.0 then
      OK := False;
      return;
   end if;
   Root_1 := (–B+Sqrt(D)) / (2.0*A);
   Root_2 := (–B–Sqrt(D)) / (2.0*A);
   OK := True;
end Quadratic;
```

The procedure Quadratic attempts to solve the equation

$$ax^2 + bx + c = 0$$

If the roots are real they are returned via the parameters Root_1 and Root_2 and OK is set to True. If the roots are complex (D < 0.0) or the equation degenerates (A = 0.0) then OK is set to False. Note the use of the return statement. Since this is a procedure there is no result to be returned and so the word **return** is not followed by an expression. It just updates the **out** or **in out** parameters as necessary and returns control back to where the procedure was called. Note also that unlike a function we can 'run' into the **end**; this is equivalent to obeying **return**. Naturally, procedures cannot have extended return statements.

The reader will note that if OK is set to False then no value is assigned to the **out** parameters Root_1 and Root_2. The copy rule for scalars then implies that the corresponding actual parameters become undefined. In practice, junk values are presumably assigned to the actual parameters and this could raise Constraint_Error if an actual parameter were constrained. This is probably bad practice and so it might be better to assign safe values such as 0.0 to the roots. An alternative is to make these parameters of mode **in out** so that the initial values of the actual parameters are left unchanged in the case of no real roots.

The procedure could be used in a sequence such as

```
declare
   L, M, N: Float;
   P, Q: Float;
   Status: Boolean;
begin
   ...   -- sets values into L, M and N
```

```
      Quadratic(L, M, N, P, Q, Status);
      if Status then
         -- roots are in P and Q
      else
         -- fails
      end if;
   end;
```

This is a good moment to emphasize the point made in Section 6.7 that it is often better to introduce our own two-valued enumeration type rather than use the predefined type Boolean. The above example would be clearer if we had declared

```
type Roots is (Real_Roots, Complex_Roots);
```

with other appropriate alterations.

We conclude this section by emphasizing that an **out** parameter passed by copy behaves as an ordinary variable that happens not to be initialized. Thus the statements of the above example could be recast in the form

```
begin
   OK := D >= 0.0 and A /= 0.0;
   if not OK then
      return;
   end if;
   Root_1 := ... ;
   Root_2 := ... ;
end Quadratic;
```

where the Boolean expression **not** OK reads the **out** parameter OK.

Exercise 10.3

1 Write a procedure Swap to interchange the values of the two parameters of type Float.

2 Rewrite the function Rev of Section 10.1 as a procedure with a single parameter. Use it to reverse an array R of type Row. Consider rewriting it with two parameters.

3 Why is the following unwise?

```
A: Vector(1 .. 1);

procedure P(V: Vector) is
begin
   A(1) := V(1) + V(1);
   A(1) := V(1) + V(1);
end;
...
A(1) := 1.0;
P(A);
```

10.4 Aliasing

Aliasing means having two names for the same object and can be a source of problems. Thus in the example in Exercise 10.3(**3**) above, within the procedure P, the arrays A and V are the same array in the case when P is called with A as a parameter. The difficulty is that the array A is accessible by two routes, directly and indirectly via the parameter P.

The big problem in this example is that the behaviour depends upon whether the parameter is passed by reference or by value. If it is passed by reference, then both assignments are doubling A(1) so that its final value is 4.0. However, if the array is passed by value, then A and V are different and both assignments assign 2.0 to A(1) and its final value is 2.0. The program has a bounded error and a good compiler will provide a warning or might raise Program_Error.

In Ada 2012, functions can have parameters of any modes. This can be useful and avoids introducing unnecessary subterfuges. For example, we might write a very simple pseudo-random number generator thus

```
function My_Random(Seed: in out Integer) return Integer is
   N: constant Integer := ... ;
   M: constant Integer := ... ;
begin
   Seed := Seed * N mod M;
   return Seed;
end My_Random;
```

where the values of N and M are suitably chosen so that the sequence iterates over a large number of the possible values in a seemingly haphazard manner. We typically initialize the sequence by giving an odd value to the variable we choose to define the sequence and then call My_Random as required

```
XXX: Integer := 12345;
...
loop
   R := My_Random(XXX);
   ...
end loop;
```

(This works surprisingly well with even quite small relatively prime values for N and M such as 5^5 and 2^{13}.)

Without parameters of mode **in out**, we have to use a procedure rather than a function (annoying), make Seed non-local (very naughty) or use a private type for Seed with perhaps a component of an access type (sly). The predefined library uses the sly approach, see Section 23.4.

However, permitting functions to have parameters of all modes gives rise to further opportunities for unexpected behaviour resulting from aliasing.

The problems arise largely because (to aid optimization, it is said), Ada does not define the order of evaluation of a number of things such as the operands of a binary operator and the parameters in a subprogram call (there is a list in Section 27.4).

It is far too late to do anything about specifying these orders of evaluation so the approach taken is to prevent as much aliasing as possible. Accordingly, Ada 2012 introduces a number of rules which keep the problems to a minimum.

First, there are rules for determining when two names are *known to denote the same object*. Thus they denote the same object if

- both names statically denote the same stand-alone object or parameter; or
- both names are selected components, their prefixes are known to denote the same object, and their selector names denote the same component.

and so on with similar rules for dereferences, indexed components and slices. There is also a rule about renaming so that if we have

```
C: Character renames S(5);
```

then C and S(5) are known to denote the same object. The index naturally has to be static. Renaming is discussed in Section 13.7.

A further step is to define when two names *are known to refer to the same object*. This covers some cases of overlapping. Thus given a record R of type T with a component C, we say that R and R.C are known to refer to the same object. Similarly with an array A we say that A and A(K) are known to refer to the same object (K does not need to be static in this example).

Given these definitions we can now state the two basic restrictions. The first concerns parameters of elementary types:

- For each name N that is passed as a parameter of mode **in out** or **out** to a call of a subprogram S, there is no other name among the other parameters of mode **in out** or **out** to that call of S that is known to denote the same object.

Roughly speaking this comes down to saying two or more parameters of mode **out** or **in out** of an elementary type cannot denote the same object. This applies to both functions and procedures.

As an example consider

```
procedure Do_It(Double, Triple: in out Integer) is
begin
   Double := Double * 2;
   Triple := Triple * 3;
end Do_It;
```

with

```
Var: Integer := 2;
...
Do_It(Var, Var);                          -- illegal in Ada 2012
```

The key problem is that parameters of elementary types are always passed by copy and the order in which the parameters are copied back is not specified. Thus Var might end up with either the value of Double or the value of Triple.

The other restriction concerns constructions which have several constituents that can be evaluated in any order and can contain function calls. Basically it says:

- If a name N is passed as a parameter with mode **out** or **in out** to a function call that occurs in one of the constituents, then no other constituent can involve a name that is known to refer to the same object.

Constructions cover many situations such as aggregates, assignments, ranges and so on as mentioned earlier.

This rule excludes the following aggregate

```
(Var, F(Var))                          -- illegal in Ada 2012
```

where F has an **in out** parameter.

The rule also excludes the assignment

```
Var := F(Var);                         -- illegal
```

if the parameter of F has mode **in out**. Remember that the destination of an assignment can be evaluated before or after the expression. So if Var were an array element such as A(I) then the behaviour could vary according to the order. To encourage good practice, it is also forbidden even when Var is a stand-alone object.

Similarly, the procedure call

```
Proc(Var, F(Var));                     -- illegal
```

is illegal if the parameter of F has mode **in out**. Examples of overlapping are also forbidden such as

```
ProcA(A, F(A(K)));                     -- illegal
ProcR(R, F(R.C));                      -- illegal
```

assuming still that F has an **in out** parameter and that ProcA and ProcR have appropriate profiles because, as explained above, A and A(K) are known to refer to the same object as are R and R.C.

On the other hand

```
Proc(A(J), F(A(K)));                   -- OK
```

is permitted provided that J and K are different objects because this is only a problem if J and K happen to have the same value.

The intent is to detect situations that are clearly troublesome. Other situations that might be troublesome (such as if J and K happen to have the same value) are allowed, since to prevent them would make many programs illegal that are not actually dubious. This would cause incompatibilities and upset many users whose programs are perfectly correct.

10.5 Named and default parameters

The forms of subprogram call we have been using so far have given the actual parameters in positional order. As with aggregates we can also use the named notation in which the formal parameter name is also supplied; the parameters do not then have to be in order.

So we could write

```
Quadratic(A => L, B => M, C => N,
                                    Root_1 => P, Root_2 => Q, OK => Status);
Increment(X => I);
Add(C => Q, A => 2+P, B => 37);
```

We could even write

```
Increment(X => X);
```

as we will see in Section 10.6.

This notation can also be used with functions

```
F := Factorial(N => 4);
S := Sqrt(X => T+0.5);
X := Inner(B => W, A => V);
```

The named notation cannot, however, be used with operators called with the usual infixed syntax (such as V*W) because there is clearly no convenient place to put the names of the formal parameters.

As with record aggregates, the named and positional notations can be mixed and any positional parameters must come first and in their correct order. However, unlike record aggregates, each parameter must be given individually and **others** may not be used. So we could write

```
Quadratic(L, M, N, Root_1 => P, Root_2 => Q, OK => Status);
```

The named notation leads into the topic of default parameters. It sometimes happens that one or more **in** parameters usually take the same value on each call; we can give a default expression in the subprogram specification and then omit it from the call.

Consider the problem of ordering a dry martini in the United States. One is faced with choices described by the following enumeration types

```
type Spirit is (Gin, Vodka);
type Style is (On_The_Rocks, Straight_Up);
type Trimming is (Olive, Twist);
```

The default expressions can then be given in a procedure specification thus

```
procedure Dry_Martini(Base: Spirit := Gin;
                      How: Style := On_The_Rocks;
                      Plus: Trimming := Olive);
```

Typical calls might be

```
Dry_Martini(How => Straight_Up);
Dry_Martini(Vodka, Plus => Twist);
Dry_Martini;
Dry_Martini(Gin, Straight_Up);
```

The first call uses the named notation; we get gin, straight up plus olive. The second call mixes the positional and named notations; as soon as a parameter is omitted the named notation must be used. The third call illustrates that all parameters can be omitted. The final call shows that a parameter can, of course, be

supplied even if it happens to take the same value as the default expression; in this case it avoids using the named form for the second parameter.

Note that default expressions can only be given for **in** parameters. They cannot be given for operators but they can be given for functions designated by identifiers. Such a default expression (like a default expression for an initial value in a record type declaration) is only evaluated when required; that is, it is evaluated each time the subprogram is called and no corresponding actual parameter is supplied. Hence the default value need not be the same on each call although it usually will be. Default expressions are widely used in the standard input–output package to provide default formats.

Default expressions illustrate the subtle rule that a parameter specification of the form

```
P, Q: in Integer := E
```

is strictly equivalent to

```
P: in Integer := E;  Q: in Integer := E
```

(The reader will recall a similar rule for object declarations; it also applies to record components.) As a consequence, the default expression is evaluated for each omitted parameter in a call. This does not usually matter but would be significant if the expression E included a function call with side effects.

Exercise 10.5

1 Write a function Add which returns the sum of the two integer parameters and takes a default value of 1 for the second parameter. How many different ways can it be called to return N+1 where N is the first actual parameter?

2 Rewrite the specification of Dry_Martini to reflect that you prefer Vodka at weekends. Hint: declare a function to return your favourite spirit according to the global variable Today.

10.6 Overloading

We saw in Section 10.2 how new meanings could be given to existing language operators. This overloading applies to subprograms in general.

A subprogram will overload an existing meaning rather than hide it provided that its specification is sufficiently different. Hiding will occur if the number, order and base types of the parameters and any result are the same; this level of matching is known as type conformance. A procedure cannot hide a function and vice versa. Note that the names of the parameters, their mode and the presence or absence of constraints or default expressions do not matter. One or more overloaded subprograms may be declared in the same declarative part.

Subprograms and enumeration literals can overload each other. In fact an enumeration literal is formally thought of as a parameterless function with a result of the enumeration type. There are two kinds of uses of identifiers – the overloadable ones and the non-overloadable ones. At any point an identifier either refers to a single entity of the non-overloadable kind or to one or many of the

overloadable kind. A declaration of one kind hides the other kind and cannot occur in the same declaration list.

Each use of an overloaded identifier has to have a unique meaning as determined by the so-called overload resolution rules which use information such as the types of parameters. If the rules do not lead to a single meaning then the program is ambiguous and thus illegal.

Such ambiguities can often be resolved by qualification as we saw when the operator "<" was used with the character literals in Section 9.4. As a further example consider the British Channel Islands; the largest three are Guernsey, Jersey and Alderney. There are styles of woollen garments named after each

```
type Garment is (Guernsey, Jersey, Alderney);
```

and breeds of cattle named after two of them (the Alderney breed became extinct as a consequence of World War II)

```
type Cow is (Guernsey, Jersey);
```

and we can perhaps imagine shops that sell both garments and cows according to

```
procedure Sell(Style: Garment);
procedure Sell(Breed: Cow);
```

The statement

```
Sell(Alderney);
```

is not ambiguous since Alderney has only one interpretation. However

```
Sell(Jersey);
```

is ambiguous since we cannot tell whether a cow or garment is being sold. One way of resolving the ambiguity is to use type qualification thus

```
Sell(Cow'(Jersey));
```

In this example the ambiguity could also be resolved by using a named parameter call thus

```
Sell(Breed => Jersey);
```

We conclude by noting that ambiguities typically arise only when there are several overloadings. In the case here both Sell and Jersey are overloaded; in the example in Section 8.4 both the operator "<" and the literals 'X' and 'L' were overloaded.

10.7 Declarations, scopes and visibility

We said earlier that it is sometimes necessary or just convenient to give a subprogram specification on its own without the body. The specification is then followed by a semicolon and is known as a subprogram declaration. A complete subprogram, which always includes the full specification, is known as a subprogram body.

Subprogram declarations and bodies must, like other declarations, occur in a declarative part and a subprogram declaration must be followed by the corresponding body in the same declarative part. (Strictly speaking it must be in the same declarative region as we shall see when we discuss the impact of packages on visibility in Section 13.6 – but in the case of declarations in blocks and subprograms this comes to the same thing.)

An example of where it is necessary to use a subprogram declaration occurs with mutually recursive procedures. Suppose we wish to declare two procedures F and G which call each other. Because of the rule regarding linear elaboration of declarations we cannot write the call of F in the body of G until after F has been declared and vice versa. Clearly this is impossible if we just write the bodies because one must come second. However, we can write

```
procedure F( ... );          -- declaration of F

procedure G( ... ) is        -- body of G
begin
   ...
   F( ... );
   ...
end G;

procedure F( ... ) is        -- body of F repeats
begin                        -- its specification
   ...
   G( ... );
   ...
end F;
```

and then all is well.

If the specification is repeated then it must be given in full and the two must be the same. Technically we say that the two profiles in the specification must have full conformance. The profile is the formal parameter list plus result type if any. Some slight variation is allowed provided the static meaning is the same. For example: a numeric literal can be replaced by another numeric literal with the same value; an identifier can be replaced by a dotted name as described later in this section; an explicit mode **in** can be omitted; a list of parameters of the same subtype can be given distinctly; and of course the lexical spacing can be different. Thus the following two profiles are fully conformant

```
(X: in Integer := 1000; Y, Z: out Integer)
(X: Integer := 1e3; Y: out Integer; Z: out Integer)
```

It is worth noting that one reason for not allowing explicit constraints in parameter specifications is to remove any problem regarding conformance since there is then no question of evaluating constraint expressions twice and possibly having different results because of side effects. No corresponding question arises with default expressions (which are of course written out twice) since they are only evaluated when the subprogram is called.

There are other, less rigorous, levels of conformance which we will meet when we discuss access to subprogram types in Section 11.8 and renaming in Section

13.7. We have already met the weakest level of conformance which is called type conformance and controls the hiding of one subprogram declaration by another. As we saw in the previous section, hiding occurs provided just the types in the profiles are the same.

Another important situation where we have to write a subprogram declaration as well as a body, occurs in Chapter 12 when we discuss packages. Even if not always necessary, it is sometimes clearer to write subprogram declarations as well as bodies. An example might be in the case where many subprogram bodies occur together. The subprogram declarations could then be placed together at the head of the declarative part in order to act as a summary of the bodies to come.

Since subprograms occur in declarative parts and themselves contain declarative parts, they may be textually nested without limit. The normal hiding rules applicable to blocks described in Section 6.2 also apply to declarations in subprograms. (The only complication concerns overloading as discussed in the previous section.) We are also now in a position to describe the difference between visibility and direct visibility as illustrated by the nested procedures in Figure 10.1.

Just as for the example in Section 6.2, the inner J hides the outer one and so the outer J is not directly visible inside the procedure Q after the start of the declaration of the inner J.

However, we say that it is still visible even though not directly visible because we can refer to the outer J by the so-called dotted notation in which the identifier J is prefixed by the name of the unit immediately containing its declaration followed by a dot. So within Q we can refer to the outer J as P.J as illustrated by the initialization of L.

If the prefix is itself hidden then it can always be written the same way. Thus the inner J could be referred to as P.Q.J.

An object declared in a block cannot usually be referred to in this way since a block does not normally have a name. However, a block can be named in a similar way to a loop as shown in the following

```
Outer:
declare
   J: Integer := 0;
begin
   ...
   declare
      K: Integer := J;
      J: Integer;
      L: Integer := Outer.J;
   begin
      ...
   end;
end Outer;
```

Here the outer block has the identifier Outer. Unlike subprograms, but like loops, the identifier has to be repeated after the matching **end**. Naming the block enables us to initialize the inner declaration of L with the value of the outer J.

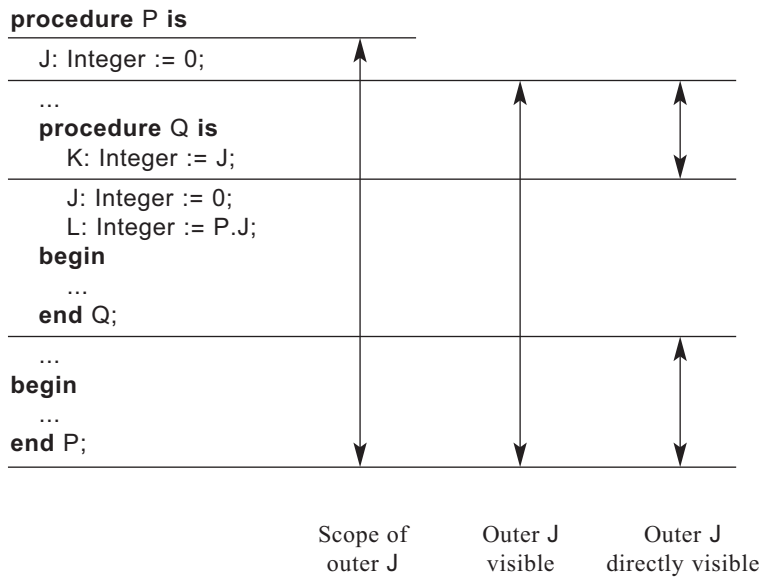


Figure 10.1 Scope and visibility.

Within a loop it is possible to refer to a hidden loop parameter in the same way. We could even rewrite the example in Section 8.1 of assigning zero to the elements of AA as

```
LL:
for I in AA'Range(1) loop
   for I in AA'Range(2) loop
      AA(LL.I, I) := 0.0;
   end loop;
end loop LL;
```

although one would be a little crazy to do so!

It should be noted that the dotted form can always be used even if it is not necessary.

This notation can also be applied to operators. Thus the variable Result declared inside "*" can be referred to as "*".Result. And equally if "*" were declared inside a block B then it could be referred to as B."*". If it is called with this form of name then the normal function call must be used

```
X := B."*"(V, W);
```

Indeed, the functional form can always be used as in

```
X := "*"(V, W);
```

and we could then also use the named notation

```
X := "*"(A => V, B => W);
```

The named notation also permits the formal parameter name to be used even if it is not directly visible. Thus we can write

```
X: Integer;
...
Increment(X => X);
```

even though the formal parameter X is not generally visible because it has been hidden by the newly declared X used as the actual parameter.

As we have seen, subprograms can alter global variables and therefore have side effects. (A side effect is one brought about other than via the parameter and result mechanism.) It is generally considered rather undesirable to write subprograms, especially functions, that have side effects. However, some side effects are beneficial. Any subprogram which performs input–output has a side effect on the file; a function delivering successive members of a sequence of random numbers only works because of its side effects; if we need to count how many times a function is called then we use a side effect; and so on. However, care must be taken when using functions with side effects that the program is correct since there are various circumstances in which the order of evaluation is not defined. There are various rules regarding aliasing which we met in Section 10.4 which help us to avoid many problems in this area.

We conclude this section with a brief discussion of the hierarchy of **exit**, **return** and **goto** and the scopes of block and loop identifiers and labels.

A **return** statement terminates the execution of the immediately embracing subprogram. It can occur inside an inner block or inside a loop in the subprogram and therefore also terminate the loop. An extended return statement may include inner simple return statements, blocks and loops but not other extended return statements.

An **exit** statement terminates the named or immediately embracing loop. It can also occur inside an inner block or extended return statement but cannot occur inside a subprogram declared in the loop and thereby also terminate the subprogram. A **goto** statement can transfer control out of a loop or block or an extended return statement but not out of a subprogram.

As far as scope is concerned, identifiers of labels, blocks and loops behave as if they are declared at the end of the declarative part of the immediately embracing subprogram or block (or package or task body). Moreover, distinct identifiers must be used for all blocks, loops and labels inside the same subprogram (or package or task body) even if some are in inner blocks. Thus two labels in the same subprogram cannot have the same identifier even if they are inside different inner blocks. This rule reduces the risk of goto statements going to the wrong label particularly when a program is amended.

Checklist 10

Parameter and result subtypes must not be given by a subtype indication with an explicit constraint.

Formal parameter specifications are separated by semicolons not commas.

A function must return a result and should not run into its final end although a procedure can.

"/=" can only be explicitly defined if the result is not Boolean.

The order of evaluation of parameters is not defined.

The parameter and result mechanism allows an array to slide.

Scalar parameters are passed by copy. Arrays and simple records may be passed by copy or by reference.

A default parameter expression is only evaluated when the subprogram is called and the corresponding parameter is omitted.

The extended return statement was added in Ada 2005.

New in Ada 2012

The abbreviated from of function known as an expression function was introduced in Ada 2012.

Functions can have parameters of any mode in Ada 2012.

There are new rules on aliasing in Ada 2012.

11 Access Types

This last chapter concerning the small-scale aspects of Ada is about access types. These are often known as pointer types, reference types or simply addresses in other languages and provide indirect access to other entities.

Playing with pointers is like playing with fire. Fire is perhaps the most important tool known to man. Carefully used, fire brings enormous benefits; but when fire gets out of control, disaster strikes. Pointers have similar characteristics but are well tamed in the form of access types in Ada.

This taming is done through the notion of accessibility which is discussed in some detail in Sections 11.5 and 11.6. Parts of these sections might be found hard to digest and could well be skipped at a first reading.

There are two forms of access types, those which access objects and those which access subprograms. Their type may be named or anonymous. Of particular importance are parameters of an anonymous access type. The related access discriminants are discussed in Chapter 18.

11.1 Flexibility versus integrity

The manipulation of objects by referring to them indirectly through values of other objects is a common feature of most programming languages. It is also a contentious topic since, although the technique provides considerable flexibility, it is also the cause of many programming errors and moreover, used incautiously, can make programs very hard to understand and maintain. It is worth a historical digression in order to place Ada in perspective.

Algol 68 was an early language to use indirection and used the term references. Indeed the definition of Algol 68 revolved around references to such an extent that

it created the impression that the language was academically elaborate. There were also technical difficulties of dangling references, that is variables pointing to objects that no longer exist.

BCPL, from which C was derived, used references or pointers as the foundation of its storage model. Arrays were just seen as objects that could be referred to dynamically by adding an index to a base address. Thus the natural implementation model was made visible in the language itself. This almost negative abstraction was perhaps a great mistake and a step backwards in the evolution of abstraction as the driving force in language design.

C has inherited the BCPL model and it is considered quite normal practice in C to add integers to pointers (addresses) thereby creating implementation dependencies and complete freedom to do silly things.

Pascal was an example of austerity in the use of pointers. Pascal only allowed pointers to objects created in a distinct storage area commonly called the heap. The rules in Pascal were such that it was not possible to create a pointer to an object declared in the normal way on the stack. There was thus no risk of leaving the scope of the referenced object while still within the scope of the referring object and thereby leaving the latter pointing nowhere.

Ada 83 followed the Pascal model and so although secure was inflexible. This inflexibility was particularly noticeable when interfacing to programs written in other languages such as C and especially for interaction with graphical user interfaces. It only had what we now call pool specific types.

Ada 95 added more flexibility while keeping the security inherent in the strongly typed model. It introduced general access types and thus enabled pointers to items not in a storage pool (the Ada term for a heap). Security was maintained through the introduction of accessibility rules whose general objective was to prevent dangling references. Most accessibility rules were applied statically and thus had no run-time overhead. However, this was sometimes inflexible and so more dynamic techniques were also provided by access parameters which were of an anonymous type. Finally, simple access to subprogram types were introduced.

However, the Ada 95 model was somewhat inconsistent. Although it introduced general access types and access to subprogram types, it only permitted anonymous types when used as parameters, did not solve the downward closure problem and was inconsistent in its treatment of null values and constants.

Ada 2005 swept away all these inconsistencies. Access types can be named or anonymous in (almost) all contexts. Null values and constant values are handled more uniformly and the introduction of anonymous access to subprogram types as parameters solved the downward closure problem.

Ada 2012 makes a few changes in this area. There are some corrections to the rules regarding checking accessibility and some simplification to conversions between access types. Perhaps the biggest improvement is the ability to create subpools of storage pools thereby giving the user much more control over the allocation and deallocation of storage for groups of items.

Java is currently popular. It has pointers which are called references. In fact almost everything is declared using references although this is hidden from the user. This means that Java is inappropriate for high integrity applications.

Finally, note that Ada uses pointers less than many languages because it allows dynamically sized objects without using a heap.

11.2 Access types and allocators

In the case of the types we have met so far, the name of an object is bound permanently to the object itself, and the lifetime of an object extends from its declaration until control leaves the unit containing the declaration. This is far too restrictive for many applications where a more fluid control of the allocation of objects is desired. In Ada this can be done by using an access type. Objects of an access type, as the name implies, provide access to other objects and these other objects can be allocated in a storage pool independent of the block structure. One of the simplest uses of an access type is for list processing. Consider

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

type Cell is
   record
      Next: Cell_Ptr;
      Value: Integer;
   end record;

L: Cell_Ptr;
```

These declarations introduce type Cell_Ptr which accesses Cell. The variable L can be thought of as a reference variable which can only point at objects of type Cell; these are records with two components, Next which is also a Cell_Ptr and can therefore access (point to or reference) other objects of type Cell, and Value of type Integer (the 'real' contents of the record). The records can therefore be formed into a linked list. Initially there are no record objects, only the single pointer L which by default takes the initial value **null** which points nowhere. We could have explicitly given L this default value thus

```
L: Cell_Ptr := null;
```

Note the circularity in the definitions of Cell_Ptr and Cell. Because of this circularity and the need to avoid forward references, it is necessary first to give an incomplete declaration of Cell. Having done this we can declare Cell_Ptr and then complete the declaration of Cell. Between the incomplete and complete declarations, we say that we have an incomplete view of the type Cell. Such an incomplete view can only be used for a few purposes the most important of which is the definition of an access type. Moreover, the incomplete and complete declarations must be in the same list of declarations except for one case concerning private types; see Section 12.5.

When we deal with anonymous access types in Section 11.7, we shall see that it is not always necessary to use incomplete types in these circumstances. However, incomplete types are also important with mutually related types as we shall see in Section 13.5 and so it is helpful to illustrate their use.

The accessed objects are created by the execution of an allocator which can (but need not) provide an initial value. An allocator consists of the reserved word **new** followed by either just the type of the new object or a qualified expression providing also the initial value of the object. The result of an allocator is an access

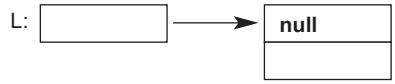


Figure 11.1 An access object.

value which can then be assigned to a variable of the access type. The new object itself will be in a storage pool as described in Section 11.9.

So the statement

```
L := new Cell;
```

creates a record of type Cell and then assigns to L a designation of (reference to or pointer to) the object. We can picture the result as in Figure 11.1.

Note that the Next component of the record takes the default value **null** whereas the Value component is undefined.

The components of the object referred to by L can be accessed using the normal dotted notation. So we could assign 37 to the Value component by

```
L.Value := 37;
```

If we attempt to do this when L has the value null, the exception Constraint_Error is raised. This check prevents nasty errors which can arise with other languages.

Alternatively we could have provided an initial value with the allocator

```
L := new Cell'(null, 37);     -- or new Cell'(Next => null, Value => 37);
```

The initial value here takes the form of a qualified aggregate, and as usual has to provide values for all the components irrespective of whether some have default initial expressions. The allocator could also have been used to initialize L when it was declared

```
L: Cell_Ptr := new Cell'(null, 37);
```

Distinguish carefully the types Cell_Ptr and Cell. The variable L is of type Cell_Ptr which accesses Cell whereas Cell is the accessed type which follows **new**.

Suppose we now want to create a further record and link it to the existing record. We could do this in three steps by declaring a further variable

```
N: Cell_Ptr;
```

and then executing

```
N := new Cell'(L, 10);
L := N;
```

The effect of these three steps is illustrated in Figure 11.2. Note how the assignment statement

```
L := N;
```

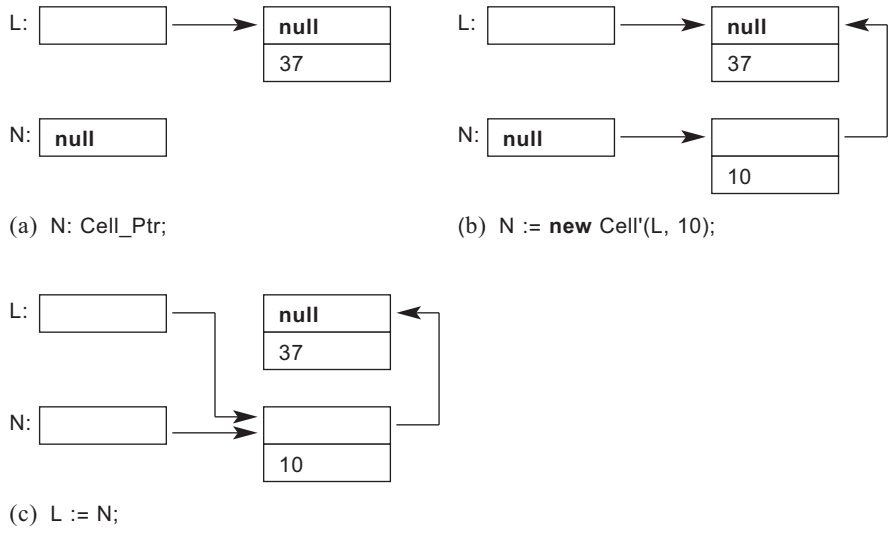


Figure 11.2 Extending a list.

copies the access values (that is, the pointers) and not the objects. If we wanted to copy the objects we could do it component by component

```
L.Next := N.Next;  L.Value := N.Value;
```

or by using **all**

```
L.all := N.all;
```

L.**all** refers to the whole object accessed by L. Using **all** is called dereferencing and is often automatic – we can think of L.Value as short for L.**all**.Value.

Similarly

```
L = N
```

will be true if L and N refer to the same object, whereas

```
L.all = N.all
```

will be true if the objects referred to happen to have the same value.

We could declare a constant of an access type but since it is a constant we must supply an initial value

```
C: constant Cell_Ptr := new Cell'(null, 0);
```

The fact that C is constant means that it must always refer to the same object. However, the value of the object could itself be changed. So

```
C.all := L.all;
```

is allowed but a direct assignment to C is not

```
C := L;                    -- illegal
```

We did not really need the variable N in order to extend the list since we could simply have written

```
L := new Cell'(L, 10);
```

This statement can be made into a general procedure for creating a new record and adding it to the beginning of a list

```
procedure Add_To_List(List: in out Cell_Ptr; V: in Integer) is
begin
   List := new Cell'(List, V);
end Add_To_List;
```

The new record containing the value 10 can now be added to the list accessed by L by writing

```
Add_To_List(L, 10);
```

The parameter passing mechanism for access types is defined to be by copy like that for scalar types. However, in order to prevent an access value from becoming undefined an **out** parameter is always copied in at the start. Remember also that an uninitialized access object takes the specific default value **null**. These two facts prevent undefined access values which could cause a program to go berserk.

The value **null** is useful for determining when a list is empty. The following function returns the sum of the Value components of the records in a list

```
function Sum(List: Cell_Ptr) return Integer is
   Local: Cell_Ptr := List;
   S: Integer := 0;
begin
   while Local /= null loop
      S := S + Local.Value;
      Local := Local.Next;
   end loop;
   return S;
end Sum;
```

Observe that we have to make a copy of List because formal parameters of mode **in** are constants. The variable Local is then used to work down the list until we reach the end. The function works even if the list is empty.

A more elaborate data structure is the binary tree. This can be represented by nodes each of which has a value plus pointers to two subnodes (subtrees) one or both of which could be null. Appropriate declarations are

```
type Node;
type Node_Ptr is access Node;
```

```
type Node is
   record
      Left, Right: Node_Ptr;
      Value: Float;
   end record;
```

As an interesting example of the use of trees consider the following procedure Sort which sorts the values in an array into ascending order

```
procedure Sort(A: in out Vector) is
   Index: Integer;
   Tree: Node_Ptr := null;

   procedure Insert(T: in out Node_Ptr; V: in Float) is
   begin
      if T = null then
         T := new Node'(null, null, V);
      elsif V < T.Value then
         Insert(T.Left, V);
      else
         Insert(T.Right, V);
      end if;
   end Insert;

   procedure Output(T: in Node_Ptr) is
   begin
      if T /= null then
         Output(T.Left);
         A(Index) := T.Value;
         Index := Index + 1;
         Output(T.Right);
      end if;
   end Output;

begin                          -- body of Sort
   for I in A'Range loop
      Insert(Tree, A(I));
   end loop;
   Index := A'First;
   Output(Tree);
end Sort;
```

The recursive procedure Insert adds a new node containing the value V to the tree in such a way that the values in the left subtree of a node are always less than the value at the node and the values in the right subtree are always greater than (or equal to) the value at the node. The recursive procedure Output copies the values in the tree into the array A by first outputting the left subtree, then copying the value at the node and finally outputting the right subtree.

The procedure Sort simply builds up the tree by calling Insert with each of the components of the array in turn and then calls Output to copy the ordered values back into the array.

The access types we have met so far have referred to records. This will often be the case but an access type can refer to any type, even another access type. So we could have an access to an integer thus

```
type Ref_Int is access Integer;
R: Ref_Int := new Integer'(46);
```

Note that the value of the integer referred to by R is denoted by R.**all**. So we can write

```
R.all := 13;
```

to change the value from 46 to 13.

An access type can also refer to an indefinite type such as the type String. In Section 8.5 we noted, when declaring the Zoo, that the animals (or rather their names) all had to have the same length. However, with a bit of ingenuity, we can overcome this by using access types. Consider

```
type A_String is access String;
```

which enables us to declare variables which can access strings of any size. So we can write

```
A: A_String := new String'("Hello");
```

We see that although we no longer have to pad the strings to a fixed length, we now have the burden of the allocation. However, we can craftily write

```
function "+" (S: String) return A_String is
begin
   return new String'(S);
end "+";
```

and then

```
type A_String_Array is array (Positive range <>) of A_String;

Zoo: constant A_String_Array :=
          (+"aardvark", +"baboon", ..., +"very long animal... ", ..., +"zebra");
```

Remember from Section 8.5 that we can declare an array of any definite subtype; all access types are definite even if they designate objects of an indefinite type and so we can declare arrays of the type A_String. (This flexibility of access types is especially crucial for object oriented programming as we saw in the brief overview in Section 3.3.)

With this formulation there is no limit on the length of the strings and we have more or less created a ragged array. However, the length of a particular string cannot be changed. Moreover, there is the overhead of the access value which is significant if the strings are short. We will return to the topic of ragged arrays in Sections 11.4 and 18.2.

A few final points of detail. Allocators illustrate the importance of the rules regarding when and how often an expression is evaluated in certain contexts. For

example, an expression in an aggregate is evaluated for each index value concerned and so

```
A: array (1 .. 10) of Cell_Ptr := (others => new Cell);
```

creates an array of ten components and initializes each of them to access a different new cell.

As a further example

```
A, B: Cell_Ptr := new Cell;
```

creates two new cells (see Section 6.1), whereas

```
A: Cell_Ptr := new Cell;
B: Cell_Ptr := A;
```

naturally creates only one. Remember also that default expressions for record components and subprogram parameters are re-evaluated each time they are required; if such an expression contains an allocator then a new object will be created each time.

If an allocator provides an initial value then this can take the form of any qualified expression. So we could have

```
L: Cell_Ptr := new Cell'(N.all);
```

in which case the object is given the same value as the object referred to by N. We could have

```
K: Integer := 46;
R: Ref_Int := new Integer'(K);
```

in which case the new object takes the value of K.

In this section we have only discussed the use of access types with allocated objects. In Section 11.4 we shall see how to use them with declared objects.

Exercise 11.2

1 Write a

```
procedure Append(First: in out Cell_Ptr; Second: in Cell_Ptr);
```

which appends the list Second (without copying) to the end of the list First. Take care of the special case where First is **null**.

2 Write a function Size which returns the number of nodes in a tree.

3 Write a function Copy which makes a complete copy of a tree.

4 Write the converse unary function "+" which takes an A_String as parameter and returns the corresponding String. Use this function to output the camel in the Zoo.

5 Write a function "&" to concatenate two parameters of type A_String.

11.3 Null exclusion and constraints

In the previous section we noted that if we dereference a value of an access type such as Cell_Ptr as in L.Value, and L happens to have the value **null** then Constraint_Error is raised. This means that there has to be a check on all such dereferences and this is clearly somewhat inefficient.

Accordingly, we can specify that an access type does not have the value **null** by adding a null exclusion when it is declared thus

```
type Cell_Ptr is not null access Cell;
```

All variables of the type now have to be declared with an initial value. This would not be very helpful with this list example because we would have to introduce some other technique for identifying the end of a list (the last cell could point to itself perhaps or the list could be cyclic).

Alternatively we could leave Cell_Ptr as before and declare a subtype thus

```
type Cell_Ptr is access Cell;
subtype NN_Cell_Ptr is not null Cell_Ptr;
...
M: NN_Cell_Ptr := L;                    -- checks not null
```

There will then be a check to ensure that L does not have the value **null** before it is assigned to M. But when we write M.Value there is no need to check M. Another approach is to give the null exclusion when we declare M thus

```
M: not null Cell_Ptr := L;
```

A null exclusion has much in common with a constraint. But there are differences. An important one is that a null exclusion can appear in a parameter profile but a constraint cannot. Thus

```
procedure P(X: in out not null Ref_Int);        -- legal
procedure Q(X: in out Integer range 1 .. N);    -- illegal
function F(X: Integer) return not null Ref_Int; -- legal
```

As explained in Section 10.7, we cannot use a constraint in a parameter specification because of possible problems of conformance and the constraint being evaluated twice. No such problem arises with a null exclusion. A null exclusion can also appear in a function result as shown above.

From time to time we say that two subtypes have to statically match. An example occurred with array conversions in Section 8.2. Where relevant, null exclusions also have to match.

We now turn to the question of constraints and access types. These can be applied to both the type being accessed and to the access type itself. This discussion may be found a bit confusing but careful attention to the detail will show that it is quite logical.

Considering first the type being accessed, we could have

```
type Ref_Pos is access Positive;
```

or equivalently

```
type Ref_Pos is access Integer range 1 .. Integer'Last;
```

The values in the objects referred to are all constrained to be positive. We could write either of

```
RP: Ref_Pos := new Positive'(10);
RP: Ref_Pos := new Integer'(10);
```

Note that if we wrote **new** Positive'(0) then Constraint_Error would be raised because 0 is not of subtype Positive. However, if we wrote **new** Integer'(0) then Constraint_Error is only raised because of the context of the allocator.

Access types can also refer to constrained and unconstrained arrays. We met the type A_String in the previous section. We could also have

```
type Ref_Matrix is access Matrix;
R: Ref_Matrix;
```

where the type Matrix is as in Section 8.2, and then obtain new matrices with an allocator where the bounds must be provided either through an explicit initial value thus

```
R := new Matrix'(1 .. 5 => (1 .. 10 => 0.0));
```

or by applying constraints

```
R := new Matrix(1 .. 5, 1 .. 10);
```

Note the subtle distinction whereby a quote is needed in the case of the full initial value but not when we just give the constraints. This is because the first takes the form of a qualified expression whereas the second is just a subtype indication.

We could not write just

```
R := new Matrix;
```

because all array objects must have bounds (must be definite). Moreover, once allocated the bounds of a particular matrix cannot be changed. However, since R itself is unconstrained it can refer to matrices of different bounds from time to time. So we could then write

```
R := new Matrix(1 .. 10, 1 .. 5);
```

We now turn to considering the case where the access type itself is constrained. We can do this by introducing a subtype

```
subtype Ref_Matrix_3x4 is Ref_Matrix(1 .. 3, 1 .. 4);
R_3x4: Ref_Matrix_3x4;
```

and R_3x4 can then only reference matrices with corresponding bounds. Alternatively we could have directly written

```
R_3x4: Ref_Matrix(1 .. 3, 1 .. 4);
```

The objects to be accessed can be allocated using an explicit constraint

```
R_3x4 := new Matrix(1 ..3, 1 .. 4);
```

or through declaring a subtype

```
subtype Matrix_3x4 is Matrix(1 .. 3, 1 .. 4);
...
R_3x4 := new Matrix_3x4;
```

This is allowed because the subtype Matrix_3x4 supplies the array bounds.

Moreover, because the subtype supplies the bounds we could use it to qualify an aggregate and so we could initialize the new object by

```
R_3x4 := new Matrix_3x4'(others => (others => 0.0));
```

which is an interesting example of an array aggregate with **others**.

It is important to appreciate that, unlike the unconstrained variable R declared above, R_3x4 cannot refer to matrices with different bounds. As a consequence the following would raise Constraint_Error

```
R_3x4 := new Matrix(1 .. 4, 1 ..3);
```

Components of an accessed array can be referred to in the usual way, so

```
R(1, 1) := 0.0;
```

sets component (1, 1) of the matrix accessed by R to zero. The whole matrix can be referred to by **all**. So

```
R_3x4.all := (1 .. 3 => (1 .. 4 => 0.0));
```

sets all components of the matrix accessed by R_3x4 to zero.

We can think of R(1, 1) as an abbreviation for R.**all**(1, 1). As with records, dereferencing is automatic. We can also write attributes such as R'First(1) or R.**all**'First(1). Similarly, slicing is permitted for one-dimensional arrays.

11.4 Aliased objects

In Section 11.2 we saw how access types provide a means of manipulating objects created by allocators. Access types can also be used to provide indirect access to declared objects. The declaration of the type has to include the word **all** and is then known as a general access type. Thus we write

```
type Int_Ptr is access all Integer;
```

and then we can assign the 'address' of any variable of type Integer to a variable of type Int_Ptr provided that the designated variable is marked as **aliased** thus

```
IP: Int_Ptr;
I: aliased Integer;
...
IP := I'Access;
```

We can then read and update the variable I indirectly through the access variable IP by, for example

```
IP.all := 25;
```

Observe how the access value (the pointer) assigned to IP is created by the use of the Access attribute. We can only apply the Access attribute to objects declared as aliased (or those considered by default to be aliased such as tagged type parameters, see Section 14.2).

There are two reasons for specifically requiring that an object be marked as **aliased**; one is as a warning to the programmer that the object might be manipulated indirectly; the other is for the compiler so that it does not allocate space for the object in a nonstandard way (such as in a register) and thereby prevent access in the normal manner (it conversely enables the compiler to perform optimizations easily if it is not marked).

As mentioned earlier, there are accessibility rules which ensure that dangling references cannot arise. They are dealt with in detail in Section 11.6 but the general principle is that we can only apply the Access attribute to an object whose lifetime is at least that of the access type.

A variation is that we can restrict the access to be read-only by replacing **all** in the type definition by **constant**. This allows read-only access to any variable and also to a constant. So we can write

```
type Const_Int_Ptr is access constant Integer;
CIP: Const_Int_Ptr;
I: aliased Integer;
C: aliased constant Integer := 1815;
...
CIP := I'Access;          -- access to a variable, or
CIP := C'Access;          -- access to a constant
```

But we could not assign C'Access to the variable IP of the general access type Int_Ptr since that would enable us to write indirectly to the constant C.

The type accessed by a general access type can of course be any type such as an array or record. We can thus build chains from records statically declared. Note that we can also use an allocator to generate general access values so that a chain could include a mixture of records from both storage mechanisms.

The components of an array can also be aliased as in

```
AI: array (1 .. 100) of aliased Integer;
...
IP := AI(I)'Access;
```

Finally, note that the accessed object could also be a component of a record type. Thus we could point directly to a component of a record (provided that the component is marked as aliased). In a fast implementation of Conway's Game of Life a cell might contain access values directly referencing the component of its eight neighbours containing the counter indicating whether the cell is alive or dead. So we can write

```
type Ref_Count is access constant Integer range 0 .. 1;
type Ref_Count_Array is array (Integer range <>) of Ref_Count;

type Cell is
   record
      Life_Count: aliased Integer range 0 .. 1;
      Total_Neighbour_Count: Integer range 0 .. 8;
      Neighbour_Count: Ref_Count_Array(1 .. 8);
   end record;
```

We can now link the cells together by statements such as

```
This_Cell.Neighbour_Count(1) :=
      Cell_To_The_North.Life_Count'Access;
```

Any constraints on the accessed object (Life_Count) must statically match those on the access type (Ref_Count); in this case they both have static bounds of 0 and 1. We met static matching in Section 8.2 when discussing the rules for conversion between array types – the same rules apply here.

The heart of the computation which computes the sum of the life counts in the neighbours might be

```
C.Total_Neighbour_Count := 0;
for I in C.Neighbour_Count'Range loop
   C.Total_Neighbour_Count :=
         C.Total_Neighbour_Count + C.Neighbour_Count(I).all;
end loop;
```

We also made the type Ref_Count an **access constant** type since we only need read access to the counter; this prevents us accidentally updating it indirectly.

General access types can also refer to composite types so we might have

```
type String_Ptr is not null access all String;
A_String: aliased String := "Hello";
SP: String_Ptr := A_String'Access;
```

The string could be changed indirectly by for example

```
SP.all := "Howdy";
```

But of course SP.**all** := "Hi"; would raise Constraint_Error.

Static matching between constraints has to be strictly observed when applying the Access attribute and in fact an array with bounds obtained from an initial value is considered different from one with an explicit constraint. So

```
Another_String: aliased String(1 .. 7) := "Welcome";
...
SP := Another_String'Access;                    -- illegal
```

is illegal because the explicit constraint on Another_String does not statically match the accessed type of String_Ptr (which is unconstrained). But A_String does match String_Ptr since there is no explicit constraint on A_String.

General access types can also be used to declare ragged arrays as for example a table of strings of different lengths such as the Zoo discussed in Section 11.2. We might declare an access type thus

```
type G_String is access constant String;
```

and the strings can then be declared in the normal way (with bounds obtained from their initial values). Without the use of aliasing we would have to allocate the strings dynamically using an allocator. Nevertheless this technique is cumbersome because all the individual strings have to be named objects.

Conversion between different general access types is permitted. The accessed types must be the same and any constraints must statically match (but see Section 14.2 if the accessed type is tagged and Section 18.5 if it has discriminants). But an access to constant type cannot be converted to an access to variable type since otherwise we might obtain write access to a constant. So we can write

```
CIP := Const_Int_Ptr(IP);
```

but not

```
IP := Int_Ptr(CIP);          -- illegal
```

Access to constant types are useful for providing read-only access to allocated objects. Thus if we have

```
type Const_T_Ptr is access constant T;
type T_Ptr is access all T;
My_Object: T_Ptr := new T'( ... );
Your_View: Const_T_Ptr := Const_T_Ptr(My_Object);
```

then although the variable My_Object enables the value of the allocated object to be changed, this cannot be done using Your_View. It is important to allocate the object using the type T_Ptr. If we did it with the type Const_T_Ptr then we would be stuck with an object that could never be changed and of course the type conversion in the opposite direction would not be possible.

Finally, note that **aliased** can be used generally for parameters and for the return object in an extended return statement. An example with parameters occurs in the functions such as Reference in containers (see Section 24.2).

Exercise 11.4

1 Using the type G_String declare the Zoo of Section 11.2 with declared strings (without allocators).

2 Declare a world of N by M cells for the Game of Life. Link them together in an appropriate manner. Use a dummy dead cell for the boundary so that the heart of the computation remains unchanged.

3 Reformulate the cells by making the access type refer to the cell as a whole rather than the internal component; how would this change the heart of the computation and the answer to the previous exercise?

11.5 Accessibility

The accessibility rules are designed to provide reasonable flexibility with complete security from dangling references. A simple static strategy applies in most circumstances and ensures that no run-time checks are required.

The basic rule is that the lifetime of an accessed object must be at least as long as that of the access type. Lifetime is a bit like scope except that it refers to the dynamic existence of the entity whereas scope refers to its (potential) static visibility.

As an illustration of the problem consider

```
procedure Main is
   type AI is access all Integer;
   Ref1: AI;
begin
   declare
      Ref2: AI;
      I: aliased Integer;
   begin
      Ref2 := I'Access;       -- illegal
      ...
      Ref1 := Ref2;
   end;
   ...
   declare
      ...   -- some other variables
   begin
      Ref1.all := 0;
   end;
end Main;
```

This is illegal because we have applied Access to a variable I with a lesser lifetime than the access type AI. The problem is not so much with the assignment to Ref2 which has the same lifetime as I but the fact that we can later assign Ref2 to Ref1 and Ref1 has a longer lifetime than I. The eventual assignment of 0 to Ref1.**all** could in principle overwrite almost anything because the value in Ref1 now refers to where I was and this space could now be used by some other variable of a different type (maybe another access type).

Other rules would have been possible. It might have been decreed that the assignment to Ref1 be forbidden or maybe that the final assignment to Ref1.**all** was the real culprit. For a number of reasons these alternatives are not sensible because they require extensive run-time checks or violate the idea that values can be assigned freely between variables of the same subtype.

So the actual rule is simply that the access value cannot be created in the first place if there is a risk that it might outlive the object that it refers to. In principle this is a dynamic rule but in most situations the check can be performed statically. This is illustrated by the following

```
procedure Main is
   type T is ...
```

```
   X: aliased T;
begin
   declare
      type A is access all T;
      Ptr: A;
   begin
      Ptr := X'Access;
      ...
   end;
end Main;
```

The object X outlives the access type A and so the assignment of X'Access to Ptr is permitted. In this example the dynamic lifetime follows from the simple static structure. However, because of the existence of procedure calls, we know that the dynamic structure at any time can include more active levels than just those in the static structure visible from the point concerned. It might thus be thought that static checks would not be satisfactory at all. However, because we can only write X'Access at points where both X and the access type A are in scope, it can in fact be shown that static checks based on the scope are exactly equivalent to dynamic checks based on the lifetime in examples such as this.

Of course a particular program might be safe even though it violates the static rules. If we are absolutely convinced that a program is safe then the alternative attribute Unchecked_Access can be applied by writing

```
Ref2 := I'Unchecked_Access;
```

The checks are then omitted and we are at risk. The first program above would then execute but be erroneous; anything could happen. See also Section 25.4 concerning how to minimize the risks of Unchecked_Access.

As we have seen, it is possible to convert one access type to another. In order to ensure that the lifetime of the accessed object is at least that of the target access type, conversion is only allowed if the target access type has a lifetime not greater then the source access type. Consider

```
declare
   type AI is access all Integer;
   I: aliased Integer;
   RefI: AI := I'Access;
begin
   declare
      type AJ is access all Integer;
      J: aliased Integer;
      RefJ: AJ := J'Access;
   begin
      RefI := AI(RefJ);        -- illegal
      ...
   end;
   ...
end;
```

The conversion to the outer type is not permitted since RefI would then have a dangling reference on exit from the inner block. On the other hand, a conversion from RefI to RefJ is perfectly safe and so is permitted. Note that the check can be performed statically.

11.6 Access parameters

In the simple cases considered in the previous section, static checks always suffice. Many programs use access types at the outermost level only and then everything has the same lifetime. But sometimes the static rules are too severe and a program is illegal even though nothing could go wrong. Consider

```
procedure Main is
   type T is ...
   procedure P is
      type A is access all T;
      Ptr: A := X'Access;                    -- illegal, X not in scope
   begin
      ...
   end P;
begin
   declare
      X: aliased T;
   begin
      P;
   end;
end Main;
```

This is illegal because X is out of scope at the point where we have written X'Access. However, if we could have assigned the 'address' of X to Ptr then nothing could have gone wrong because the lifetime of the type A is less than that of the variable X. This is such a common situation that the rules for parameters of an anonymous type are dynamic so that access values can be passed more flexibly. Such parameters are known as access parameters. Consider

```
procedure Main is
   type T is ...
   procedure P(Ptr: access T) is
   begin
      ...
   end P;
begin
   declare
      X: aliased T;
   begin
      P(X'Access);
   end;
end Main;
```

The parameter Ptr is an access parameter (as mentioned earlier, these are classed as **in** parameters although the word **in** never appears). It is initialized with the value of X'Access and all is well. Within the body of P we can dereference Ptr and manipulate the components of X as expected. The parameter Ptr is of course a constant and cannot be changed although the object it refers to can naturally be changed by assignment.

Type conversion between an access parameter and another access type is possible. Just allowing conversion to an inner type (this can be checked statically) would be too restrictive and so conversion to an outer type is also permitted and in this case an accessibility check is carried out at run time to ensure that the lifetime of the object referred to is not less than the target access type. The following is therefore allowed

```
procedure Main is
   type T is ...
   type A is access all T;
   Ref: A;
   procedure P(Ptr: access T) is
   begin
      ...
      Ref := A(Ptr);          -- dynamic check on conversion
   end P;
   X: aliased T;
begin
   P(X'Access);
   ...                        -- can now manipulate X via Ref
end Main;
```

Here we have converted the access parameter to the outer type A and assigned it to the variable Ref. For the call of P with X'Access this is safe because the lifetime of X is not less than the type A. But it might not be safe for all calls of P (such as in the previous example where X was declared in a local block). So the check has to be dynamic. If it fails then Program_Error is raised.

In order for this check to be possible all access parameters carry with them an indication of the accessibility of the actual parameter. Typically all that is necessary is to pass the static depth of the original object and this is then checked against the depth of the target type on the conversion.

Perhaps surprisingly this simple model with its combination of static checks in most circumstances and dynamic checks on some conversions of access parameters gives exactly the correct degree of flexibility combined with complete security.

The actual parameter corresponding to an access parameter can be (i) an access to an aliased object such as X'Access, (ii) another access parameter with the same accessed type, (iii) a value of a named or anonymous access type again with the same accessed type, or (iv) an allocator. In each case an appropriate indication of accessibility is passed. An access parameter, like other **in** parameters, can have a default expression and naturally this must be of one of these four forms.

An access parameter can be (i) used to provide access to the accessed object by dereferencing, (ii) passed as an actual parameter to another access parameter, or (iii) converted to another access type.

Dynamic accessibility checks occur on conversion to a named access type such as

```
Ref := A(Ptr);
```

or (which is really equivalent) if we write

```
Ref := Ptr.all'Access;
```

which dereferences to give the accessed object and then re-creates a reference to it.

An access parameter can be passed on to another access parameter; typically the accessibility indication is passed on unchanged but in the unusual circumstance where the called subprogram is internal to the calling subprogram, the accessibility level is replaced by that of the (statically known) formal calling parameter if less than the original actual parameter. Note that access parameters can have a null exclusion or be constant or both.

Access parameters are often an alternative to **in out** parameters. They both provide read and write access to the data concerned. And indeed, in the case of a record the components are referred to in the same way because of automatic dereferencing. However, the form of the actual parameter is different. Given

```
type T is ...
type A is access all T;
X: T;
Ref: A := new T'( ... );

procedure PIO(P: in out T);
procedure PA(Ptr: access T);
```

then calls of PIO take the form

```
PIO(X);  PIO(Ref.all);
```

whereas calls of PA require that variables be aliased so we have

```
X: aliased T;
...
PA(X'Access);  PA(Ref);
```

When we come to Chapter 14 we shall see that if the type T is a tagged type then objects such as X and Ref can call procedures using the prefixed notation – in which case the forms of the calls are exactly the same thus

```
X.PIO;  Ref.PIO;
X.PA;  Ref.PA;
```

An historical important difference between access and in out parameters was that in Ada 2005 a function can have an access parameter but not an in out parameter. This difference is eliminated in Ada 2012 because as we saw in the previous chapter, functions can now have parameters of any mode.

The full benefit of access parameters cannot be illustrated at the moment but will become clear when we discuss object oriented programming in Chapter 14 and access discriminants in Chapter 18.

The reader will probably have found the whole topic of accessibility and access parameters rather tedious. Accessibility is a bit like visibility; if we do something silly the compiler will tell us and we need not bother with the details of the rules most of the time. In any case, practical programs usually have a quite flat structure and problems will rarely arise. It is, however, very comforting to know that provided we avoid Unchecked_Access then dangling references will not arise and our program will not crash in a heap.

Exercise 11.6

1 Analyse the checks on the type conversions in the following indicating whether they are dynamic or static and whether they pass or fail.

```
procedure Main is
   type T is ...
   type A1 is access all T;
   Ref1: A1;
   procedure P(Ptr: access T) is
      type A2 is access all T;
      Ref2: A2;
   begin
      Ref1 := A1(Ptr);                    -- OK?
      Ref2 := A2(Ptr);                    -- OK?
      ...
   end P;
   X1: aliased T;
begin
   declare
      X2: aliased T;
   begin
      P(X1'Access);
      P(X2'Access);
   end;
end Main;
```

2 Similarly analyse the following. Note that the chained call of P2 from P1 is the interesting situation where the accessibility level has to be adjusted if the level of the original object is deeper than that of the formal parameter Ptr2. For convenience the level of the identifiers is indicated by their name. Note that the level of a formal parameter is one deeper than the subprogram itself.

```
procedure Main is
   type T is ...
   type A1 is access all T;
   Ref1: A1;
   procedure P1(Ptr2: access T) is
      type A2 is access all T;
```

```
      Ref2: A2;
      procedure P2(Ptr3: access T) is
         type A3 is access all T;
         Ref3: A3;
      begin
         Ref1 := A1(Ptr3);                    -- OK?
         Ref2 := A2(Ptr3);                    -- OK?
         Ref3 := A3(Ptr3);                    -- OK?
      end P2;
   begin
      P2(Ptr2);       -- chained call
   end P1;
   X1: aliased T;
begin
   declare
      X2: aliased T;
   begin
      declare
         X3: aliased T;
      begin
         P1(X1'Access);
         P1(X2'Access);
         P1(X3'Access);
      end;
   end;
end Main;
```

11.7 Anonymous access types

Giving a name to every access type can be a nuisance in some circumstances and so anonymous access types were introduced in Ada 2005. For example we can write

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

Not only does this avoid introducing the named type Cell_Ptr but it also avoids the incomplete type declaration of Cell. An interesting point is that the name of the type Cell is being used in its own declaration. This also occurs with other types which we shall meet in due course.

In the case of components of an anonymous access type, the type is deemed to be declared at the same level as the enclosing declaration and the usual accessibility rules then apply. Thus the declaration

```
type R is
   record
```

```
      C: access Integer;
   end record;
```

is essentially equivalent to

```
type anon is access all Integer;
type R is
   record
      C: anon;
   end record;
```

However, different rules apply to stand-alone objects such as

```
V: access Integer;
```

In Ada 2005 this was treated in the same way as the component C. However, this can give rise to problems and so, in Ada 2012, the object V carries with it the accessibility of its current value as happens with access parameters described in the previous section.

Conversions from an access parameter to a local object of an anonymous type are permitted

```
procedure P(Ptr: access T) is
   Local_Ptr: access T;
begin
   Local_Ptr := Ptr;                    -- implicit conversion
   ...
end P;
```

In Ada 2012, the accessibility information contained in the access parameter is copied to the local variable (whereas in Ada 2005 it was lost).

Conversions between anonymous access types are permitted; consider

```
A_List: access Cell := ...
...
A_List := A_List.Next;                  -- implicit conversion
```

Note that there is strictly a conversion here between the anonymous access type of A_List and that of the component Next. But they are really the same type and so the conversion is permitted. Sometimes it is convenient to name a type and conversions between named and anonymous types are also allowed in many circumstances.

```
type List is access all Cell;
X: List := ...                          -- of named type
Y: access Cell := ...                   -- of anonymous type
X := List(Y);              -- conversion to named type
Y := X;                    -- conversion to anonymous type
```

A conversion to a named access type can use the type name whereas a conversion to an anonymous type does not –- it cannot because it does not have a name! In Ada 2005, conversions to a named type always had to use the type name in the

conversion. However, Ada 2012 is more lenient and the name is only required if the conversion could fail a check (typically an accessibility check). In the example above the conversion cannot fail so we could have simply written

```
X := Y;                    -- OK in Ada 2012
```

The other possible check that might fail is when tags are checked when converting between tagged types, see Section 14.4.

Of course, if we attempted to convert an access to Integer to an access to Float then this is just illegal and the program will not compile.

Conversions concerning null exclusions do not need to be named. For example, suppose we declare Y to be of an anonymous type with a null exclusion and attempt to assign X to Y.

```
X: List := ...                      -- of named type
Y: not null access Cell := ...      -- of anonymous type
Y := X;                             -- legal, but check needed
```

A run-time check is required to ensure that null is not being assigned to Y; Constraint_Error is raised if the check fails. Named conversions are never required when the conversion concerns a subtype property such as a null exclusion. This is much the same as converting between a value in an object of type Integer and assigning it to a variable of subtype Day_Number in Section 6.4. We do not use a named conversion but Constraint_Error will be raised if the check fails.

We can also use **constant** with anonymous access types. We need to distinguish

```
ACT: access constant T := X1'Access;
CAT: constant access T := X1'Access;
```

In the first case ACT is a variable and can be used to access different objects X1 and X2 of type T. But it cannot be used to change the value of those objects. In the second case CAT is a constant and can only refer to the object given in its initialization. But we can change the value of the object that CAT refers to.

We can also write

```
CACT: constant access constant T := X1'Access;
```

The object CACT is then a constant and provides read-only access to the object X1 to which it refers. It cannot be changed to refer to another object such as X2, nor can the value of X1 be changed via CACT.

Note that we never write **all** when declaring an anonymous access type in contrast to named general access types where we had

```
type A is access all T;
```

Anonymous access types are always general and so the use of **all** is not necessary (or allowed).

Anonymous access types can also be used as subprogram parameters and results. Thus we might have

```
function F(X, Y: access Cell) return not null access Cell;
procedure P(Z: access constant Integer; W: access Integer);
```

A null exclusion and **constant** can be used with both parameters and results. Parameters of anonymous access types are known as access parameters and were discussed in the previous section. Access parameters have mode **in** although this is not explicitly stated. Thus in the case of the procedure P, the parameter Z provides read-only access to the object being referenced whereas the parameter W provides both read and write access to the object being referenced. But the parameters Z and W themselves cannot be changed.

We conclude with a few words regarding a somewhat fictitious type known as *universal_access*. We cannot directly refer to this type but it is used to explain the behaviour of **null** and equality with anonymous access types. (We shall encounter other universal types when we discuss numeric types in more detail in Chapter 17.) The first point is that the literal **null** is considered to be of this type *universal_access* and appropriate implicit conversion rules allow **null** to be assigned to access objects. And the other is that there is a predefined function "=" in the package Standard with specification

```
function "=" (Left, Right: universal_access) return Boolean;
```

and it is this predefined function which is usually called when we compare anonymous access values. There is a corresponding "/=" as usual.

Now suppose we have

```
type A is access Integer;
R, S: access Integer;
...
if R = S then
```

Since we can do an implicit conversion from the anonymous access type of R and S to the type A, there is confusion as to whether the comparison uses the equality operator of the type *universal_access* or that of the type A. Accordingly, there is a preference rule that states that in the case of ambiguity there is a preference for equality of the type *universal_access*. Similar preference rules apply to *root_integer* and *root_real* (see Chapter 17).

Another example is instructive. Suppose we wish to do a deep comparison of two linked lists defined by the type Cell. We might wish to replace the predefined function by our own recursive function "=" thus

```
function "=" (L, R: access Cell) return Boolean is
begin
   if L = null or R = null then                -- universal =
      return L = R;                            -- universal  =
   elsif L.Value = R.Value then
      return L.Next = R.Next;                  -- recurses OK
   else
      return False;
   end if;
end "=";
```

This does work because the calls of "=" in the first two lines call the predefined "=" rather than the new version being declared because of the preference rule. This did not work in Ada 2005 which did not have this preference rule and so we had to replace the lines by

```
if Standard."=" (L, null) or Standard."=" (R, null) then
   return Standard."=" (L, R);
```

which was a bit ugly.

An important point regarding anonymous access types is that they cannot be used if we wish to deallocate storage using Unchecked_Deallocation as described in Section 25.2.

Exercise 11.7

1 Which of the following are legal?

```
ACT := X2'Access;
ACT.all := X2;
CAT := X2'Access;
CAT.all := X2;
```

11.8 Access to subprograms

The ability to pass subprograms as parameters of other subprograms has been a feature of most languages since Fortran and Algol 60. A notable exception was Ada 83 in which all binding of subprogram calls to the actual subprogram was determined statically. There were a number of reasons for taking such a static approach in Ada 83. There was concern for the implementation cost of dynamic binding, it was also clear that the presence of dynamic binding would reduce the provability of programs, and moreover it was felt that the introduction of generics where subprograms could be passed as parameters would cater for practical situations where formal procedure parameters were used in other languages.

Ada 2005 and Ada 2012 allow access to subprogram types in a very similar way to access to object types. They can be named or anonymous, null exclusions are permitted, and types and objects can be marked as constant. There are also similar accessibility rules that prevent a program from doing terrible things. There is one key difference and this concerns anonymous access types as parameters. In the case of objects we have seen that access parameters carry a dynamic indication of the accessibility level of the actual parameter. But anonymous access to subprogram parameters do not carry such an indication and thus behave somewhat differently. We shall start by considering simple named access to subprogram types and then look at parameters of anonymous access to subprogram types.

An access to subprogram value can be created by the Access attribute and a subprogram can be called indirectly by dereferencing such an access value. Thus we can write

```
type Math_Function is access function (F: Float) return Float;
Do_It: Math_Function;
X, Theta: Float;
```

and Do_It can then 'point to' functions such as Sin, Cos, Tan and Sqrt which we will assume have specifications such as

```
function Sin(X: Float) return Float;
```

We can then assign an appropriate access to subprogram value to Do_It thus

```
Do_It := Sin'Access;
```

and later indirectly call the subprogram currently referred to by Do_It by writing

```
X := Do_It(Theta);
```

This is really an abbreviation for X := Do_It.**all**(Theta); but just as with many other uses of access types the .**all** is not usually required although it would be necessary if there were no parameters.

The access to subprogram mechanism can be used to program general dynamic selection and to pass subprograms as parameters. It also allows program call-back to be implemented in a natural and efficient manner.

The following procedure applies the function passed as parameter to all the elements of an array

```
procedure Iterate(Func: in not null Math_Function; V: in out Vector) is
begin
   for I in V'Range loop
      V(I) := Func(V(I));
   end loop;
end Iterate;

A_Vector: Vector := (100.0, 4.0, 0.0, 25.0);
...
Iterate(Sqrt'Access, A_Vector);      -- A_Vector is now (10.0, 2.0, 0.0, 5.0)
```

We can similarly have access to procedure types. Thus we might have algorithms for encryption of messages and apply them indirectly as follows

```
type Converter is not null access procedure (S: in out String);

procedure Encrypt(Text: in out String);
procedure Decrypt(Text: in out String);

procedure Apply(Proc: in Converter; To: in out String) is
begin
   Proc(To);                    -- indirect call
end Apply;

Sample: String := "Send reinforcements. We're going to advance.";
...
Apply(Proc => Encrypt'Access, To => Sample);
```

Tradition has it that the message becomes "Send 3/4d, we're going to a dance."! (In old British currency, 3/4d, that is three shillings and four pence, is pronounced 'three-and-fourpence'.)

Note that the last two examples used **not null** with the access to subprogram type. It will usually be the case that a null exclusion is appropriate since a null value would be unusual and the null exclusion avoids run-time checks.

Simple classic numerical codes can also be implemented in the traditional way. Thus an integration routine might have the following specification

```
type Integrand is not null access function (X: Float) return Float;
function Integrate(Fn: Integrand; Lo, Hi: Float) return Float;
```

for the evaluation of

$$\int_{lo}^{hi} f(x)dx$$

and we might then write

```
Area := Integrate(Log'Access, 1.0, 2.0);
```

which will compute the area under the curve for log(x) from 1.0 to 2.0. Within the body of the function Integrate there will be calls of the actual subprogram passed as parameter; this is a simple form of call-back.

A common paradigm within the process control industry is to implement sequencing control through successive calls of a number of interpreter actions. A sequence compiler might interactively build an array of such actions which are then obeyed. Thus we might have

```
type Action is access procedure;
Action_Sequence: array (1 .. N) of Action;
...  -- build the array
...  -- and then obey it
for I in Action_Sequence'Range loop
   Action_Sequence(I).all;
end loop;
```

where we note the need for .**all** because there are no parameters.

There are a number of rules which ensure that access to subprogram values cannot be misused. Subtype conformance matching between the profiles ensures that the subprogram always has the correct number and type of parameters and that any constraints statically match. Subtype conformance is weaker than the full conformance required between the body and specification of the same subprogram as described in Section 10.6. Subtype conformance ignores the formal parameter names and also the presence, absence or value of default initial expressions.

Accessibility rules also apply to access to subprogram types and ensure that a subprogram is not called out of context. Thus we can only apply the Access attribute if the subprogram has a lifetime at least that of the access type. This means that the simple integration routine using the named type Integrand does not work in many cases. Thus suppose we wish to integrate a function such as Exp(X**2) where Exp is some library function. We might try

```
procedure Main is
   function F(X: Float) return Float is
   begin
      return Exp(X**2);
   end F;

   Result, L, H: Float;
begin
   ...                              -- set bounds in L and H say
   Result := Integrate(F'Access, L, H);        -- illegal
   ...
end Main;
```

but this is illegal because the subprogram F has a lifetime less than that of the access type Integrand. In this particular case the problem can be overcome by declaring F itself at the same level as Integrand. But this is not always possible if functions are nested in some way as for example if we had a double integral.

The proper solution is to change the function Integrate to use an anonymous access to subprogram type as parameter and thereby dispense with the troublesome type Integrand. The specification becomes

```
function Integrate(Fn: not null access function(X: Float) return Float;
                   Lo, Hi: Float) return Float;
```

and now we can write Integrate(F'Access, ...) as required. Note that the profile for Integrate includes the profile for the anonymous type of the parameter Fn. This nesting of profiles causes no problems. The use of an access to a local procedure in this way is often called a downward closure.

We will now look at the rules for type conversions and see how they prevent unsafe assignments to global variables. The following illustrates both access to object and access to subprogram parameters.

```
type AOT is access all Integer;
type APT is access procedure (X: in out Float);

Evil_Obj: AOT;
Evil_Proc: APT;

procedure P(Objptr: access Integer;
            Procptr: access procedure (X: in out Float)) is
begin
   Evil_Obj := AOT(Objptr);                 -- may fail at run time
   Evil_Proc := APT(Procptr);               -- always fails at compile time
end P;

declare
   An_Obj: aliased Integer;
   procedure A_Proc(X: in out Float) is
   begin ... end A_Proc;
begin
   P(An_Obj'Access, A_Proc'Access);         -- legal
end;
```

```
...
Evil_Obj.all := 0;          -- would assign to nowhere
Evil_Proc.all( ... );       -- would call nowhere
```

The procedure P has an access to object parameter Objptr and an access to subprogram parameter Procptr; they are both of anonymous type. The call of P in the local block passes the addresses of a local object An_Obj and a local procedure A_Proc to P. This is permitted. We now attempt to assign the parameter values from within P to global objects Evil_Obj and Evil_Proc with the intent of assigning indirectly via Evil_Obj and calling indirectly via Evil_Proc after the object and procedure referred to no longer exist.

Both of these wicked deeds are prevented by the accessibility rules. In the case of the object parameter Objptr, the accessibility level of the actual An_Obj is greater than that of the type AOT and the conversion is prevented at run time and Program_Error is raised. But if An_Obj had been declared at the same level as AOT and not within an inner block then the conversion would have been permitted.

However, somewhat different rules apply to anonymous access to subprogram parameters. They do not carry an indication of the accessibility level of the actual parameter but simply treat it as if it were infinite (strictly – deeper than anything else). This prevents the conversion to the type APT and all is well; this is detected at compile time. But note that if the procedure A_Proc had been declared at the same level as APT then the conversion would still have failed because the accessibility level is treated as infinite.

There are a number of reasons for the different treatment of anonymous access to subprogram types. A big problem is that named access to subprogram types are implemented in the same way as C pointers to functions in almost all compilers. Permitting the conversion from anonymous access to subprogram types to named ones would thus have caused problems because that model does not work for many implementations. Carrying the accessibility level around would not have prevented these conversions. The key goal is simply to provide a facility corresponding to that in Pascal and not to encourage too much fooling about with access to subprogram types. Recall that the attribute Unchecked_Access is permitted for access to object types but is considered far too dangerous for access to subprogram types for similar reasons.

Conversion between access to subprogram types also requires that the profiles have subtype conformance and that any null exclusion is not violated.

It is of course possible for a record to contain components whose types are access to subprogram types. We will now consider a possible fragment of the system which drives the controls in the cockpit of some mythical Ada Airlines. There are a number of physical buttons on the console and we wish to associate different actions corresponding to pushing the various buttons.

```
type Button;

type Response_Ptr is access procedure (B: in out Button);

type Button is
   record
      Response: Response_Ptr;
      ...   -- other aspects of the button
```

```
   end record;

procedure Associate(B: in out Button; ... );
procedure Push(B: in out Button);
procedure Set_Response(B: in out Button; R: in Response_Ptr);
```

A button is represented as a record containing a number of components describing properties of the button (position of message on the display for example). The component Response is an access to a procedure which is the action to be executed when the button is pushed. Note carefully that the button value is passed to this procedure as a parameter so that the procedure can obtain access to the other components of the record describing the button. Incidentally, observe that the incomplete type Button is allowed to be used for a parameter or result of the access to subprogram type Response_Ptr before its full type declaration in order to break the otherwise inevitable circularity.

The procedure Set_Response assigns an appropriate access value to the component Response and the procedure Associate makes the connection between the physical button and the software button and fills in the other components. Other functions (not shown) provide access to them. The procedure Push is called when any physical button is pushed, the parameter indicating its identity. The bodies of the subprograms might be as follows

```
procedure Push(B: in out Button) is
begin
   B.Response(B);          -- indirect call
end Push;

procedure Set_Response(B: in out Button; R: in Response_Ptr) is
begin
   B.Response := R;        -- set procedure value in record
end Set_Response;
```

We can now set the specific actions we want when a button is pushed. We might want some emergency action to happen when a big red button is pushed.

```
Big_Red_Button: Button;

procedure Emergency(B: in out Button) is
begin
   Broadcast("mayday");
   ...
   Eject_Pilot;
end Emergency;
...

Associate(Big_Red_Button, ... );
Set_Response(Big_Red_Button, Emergency'Access);
...
if Disaster then
   Push(Big_Red_Button); ...
```

It is interesting to observe that we could not have used anonymous access to subprogram types instead of the named type Response_Ptr. This is because the assignment to B.Response in the procedure Set_Response would then violate the accessibility rules since the accessibility level of the anonymous parameter R would be infinite. But we could have made the component of the record of an anonymous type.

Finally note that certain subprograms are considered intrinsic which means that they are essentially built in to the compiler. We say that their calling convention is Intrinsic whereas normal subprograms have calling convention Ada. The Access attribute cannot be applied to intrinsic subprograms.

Examples of intrinsic subprograms which we have met so far are the predefined operations in Standard such as "+", all enumeration literals (which, as mentioned in Section 10.6, are thought of as parameterless functions), an implicitly declared "/=" as a companion to "=" (see Section 10.2), and attributes that are subprograms such as Pred and Succ. Other intrinsic subprograms are those implicitly declared by derivation (Section 12.3) except in the case of tagged types (Section 14.1), and subprograms immediately declared in protected bodies (Section 20.4).

An important use of access to subprogram types is for interfacing to subprograms written in other languages; this involves the use of various pragmas referring to calling conventions and is discussed in Section 25.5. See also Section 20.4 for access types referring to protected operations. We shall meet many examples of the use of anonymous access to subprogram types as downward closures when we discuss the container library in Chapter 24.

Exercise 11.8

1 Using the function Integrate show how to evaluate

$$\int_0^P e^t \sin t \, dt$$

2 Declare the specification of an appropriate function for finding a root of the equation $f(x) = 0$, and then show how you would find the root of

$$e^x + x = 7$$

3 Write a further function Integrate to evaluate

$$\int_{LX}^{HX} \int_{LY}^{HY} F(x, y) \, dy \, dx$$

in which $F(x, y)$ is an arbitrary function of the two variables x and y. This function Integrate should use the existing function Integrate for a single variable.

11.9 Storage pools

Accessed objects are allocated in a space called a storage pool associated with the access type. This pool will typically cease to exist when the scope of the access type

is finally left but by then all the access variables will also have ceased to exist, so no dangling reference problems can arise.

If an allocated object becomes inaccessible because no declared objects refer to it directly or indirectly then the storage it occupies may be reclaimed so that it can be reused for other objects. An implementation may (but need not and most do not) provide a garbage collector to do this.

Alternatively, there is a mechanism whereby a program can indicate that an object is no longer required; if, mistakenly, there are still references to such allocated objects then the use of such references is erroneous. For details see Sections 25.2 and 25.4 which also describe how a user may create and control individual storage pools and subpools.

It is important to realize that each declaration of an access type introduces a new logically distinct set of accessed objects. Such sets might reside in different storage pools. Two access types can refer to objects of the same type but the access objects must not refer to objects in the wrong set. So we could have

```
type Ref_Int_A is access all Integer;
type Ref_Int_B is access all Integer;
RA: Ref_Int_A := new Integer'(10);
RB: Ref_Int_B := new Integer'(20);
```

The objects created by the two allocators are both of the same type but the access values are of different types determined by the context of the allocator and the objects might be in different pools. But we can convert between the types by using the type name because the accessed types are the same.

So we have

```
RA.all := RB.all;          -- legal, copies the values of the objects
RA := RB;                  -- illegal, different types
RA := Ref_Int_A(RB);       -- legal, has explicit conversion
```

If a named access type omits **all** then we call it a pool specific type because it can only access objects allocated in a pool. The key differences between pool specific types and general access to object types are

- pool specific types can only refer to allocated objects and never to aliased declared objects,
- pool specific types cannot be marked as constant,
- conversion between different pool specific types is forbidden.

There is an exception to the last rule. If one type is derived from another as discussed in Section 12.3, then conversion between them is permitted because they share the same storage pool.

However, although we cannot generally convert between pool specific types, we can convert from a pool specific type to a general type, but conversion in the opposite direction is not permitted. Similarly, we can convert from a pool specific type to an anonymous type but not in the opposite direction. The reason is simply that the general or anonymous type might designate an object that is not in a pool.

Checklist 11

An incomplete declaration can only be used in an access type.

The scope of an allocated object is that of the access type.

Access objects have a default initial value of **null**.

An allocator in an aggregate is evaluated for each index value.

An allocator with a complete initial value uses a quote.

A general named access type has **all** or **constant** in its definition.

An anonymous access type never has **all** but may have **constant**. Anonymous access types are general.

The attribute Access can only be applied to non-intrinsic subprograms and to aliased objects.

Beware Unchecked_Access.

Conversion to a pool specific access type is not allowed.

Anonymous access to object parameters (access parameters) carry a dynamic indication of the accessibility of the actual parameter. But anonymous access to subprogram parameters have an infinite accessibility level.

The only anonymous access types prior to Ada 2005 were access parameters (and access discriminants).

Access parameters could not be marked **constant** in Ada 95.

Null exclusions were added in Ada 2005.

Anonymous access to subprogram types were added in Ada 2005.

New in Ada 2012

Stand-alone objects of an anonymous access type carry the accessibility level of their value with them in Ada 2012.

Conversions do not need to be named in Ada 2012 unless a check could fail.

There is a preference rule for predefined equality of *universal_access* in Ada 2012.

Program 2

Sylvan Sorter

This program uses the treesort algorithm of Section 11.2 except that it sorts a list rather than an array.

The program first reads in a sequence of positive integers terminated by the value zero (or a negative value) and builds it into a list which is printed in a tabular format. This unsorted list is then converted into a binary tree which is printed in the usual representation. The binary tree is then converted back into an ordered list and finally the sorted values are printed. The program repeats until it encounters an empty sequence (just a zero or negative number).

The two library packages Lists and Trees contain the key data structures Cell and Node and associated operations. Although we have not yet discussed packages, private types and compilation units in depth this is much better than putting everything inside the main subprogram which is the style which would have to be adopted using just the algorithmic facilities described in Part 2. The types List and Tree are private types. Their full type reveals that they are access types referring to Cell and Node. Thus the inner structure of the types Cell and Node are not used outside their defining packages. The small package Page declares a constant defining the page width and a subtype defining a line of text.

There are then a number of library subprograms for performing the main activities, namely Read_List, Print_List, Print_Tree, Convert_List_To_Tree and Convert_Tree_To_List. Finally, the main subprogram calls these within an outer loop.

This program illustrates the use of both named and anonymous access types. The type Cell uses an anonymous type for its component whereas the type Tree uses a named type.

Using an anonymous type in Tree would result in many named conversions in Ada 2005 although, as mentioned in Section 11.7, the conversions would not need to be named in Ada 2102 because they could not fail.

```
package Lists is
   type List is private;

   function Is_Empty(L: List) return Boolean;
   procedure Clear(L: out List);
   function Make_List(V: Integer) return List;
   procedure Take_From_List(L: in out List;
                            V: out Integer);
   procedure Append(First: in out List;
                    Second: in List);
private
   type Cell is
      record
         Next: access Cell;
         Value: Integer;
      end record;

   type List is access all Cell;
end;

package body Lists is
   function Is_Empty(L: List) return Boolean is
   begin
      return L = null;
   end Is_Empty;

   procedure Clear(L: out List) is
   begin
      L := null;
   end Clear;

   function Make_List(V: Integer) return List is
   begin
      return new Cell'(null, V);
   end Make_List;

   procedure Take_From_List(L: in out List;
                            V: out Integer) is
   begin
      V := L.Value;
      L := List(L.Next);
   end Take_From_List;
```

```
   procedure Append(First: in out List;
                    Second: in List) is
      Local: access Cell := First;
   begin
      if First = null then
         First := Second;
      else
         while Local.Next /= null loop
            Local := Local.Next;
         end loop;
         Local.Next := Second;
      end if;
   end Append;

end Lists;
```

```
package Trees is
   type Tree is private;

   function Is_Empty(T: Tree) return Boolean;
   procedure Clear(T: out Tree);
   procedure Insert(T: in out Tree; V: in Integer);
   function Depth(T: Tree) return Integer;

   function Left_Subtree(T: Tree) return Tree;
   function Right_Subtree(T: Tree) return Tree;
   function Node_Value(T: Tree) return Integer;

private
   type Node;
   type Tree is access Node;
   type Node is
      record
         Left, Right: Tree;
         Value: Integer;
      end record;
end;

package body Trees is
   function Is_Empty(T: Tree) return Boolean is
   begin
      return T = null;
   end Is_Empty;

   procedure Clear(T: out Tree) is
   begin
      T := null;
   end Clear;

   procedure Insert(T: in out Tree; V: in Integer) is
   begin
      if T = null then
         T := new Node'(null, null, V);
      elsif V < T.Value then
         Insert(T.Left, V);
      else
         Insert(T.Right, V);
      end if;
   end Insert;

   function Depth(T: Tree) return Integer is
   begin
      if T = null then return 0; end if;
      return 1 + Integer'Max(Depth(T.Left),
                             Depth(T.Right));
   end Depth;

   function Left_Subtree(T: Tree) return Tree is
   begin
      return T.Left;
   end Left_Subtree;

   function Right_Subtree(T: Tree) return Tree is
   begin
      return T.Right;
   end Right_Subtree;

   function Node_Value(T: Tree) return Integer is
   begin
      return T.Value;
   end Node_Value;

end Trees;
```

```
package Page is
   Width: constant Integer := 40;
   subtype Line is String(1 .. Width);
end Page;
```

```
with Lists;  use Lists;
with Ada.Integer_Text_IO;  use Ada;
procedure Read_List(L: out List;
                    Max_Value: out Integer) is
   Value: Integer;
begin
   Max_Value := 0;
   Clear(L);
   loop
      Integer_Text_IO.Get(Value);
      exit when Value <= 0;
      if Value > Max_Value then
         Max_Value := Value;
      end if;
      Append(L, Make_List(Value));
   end loop;
end Read_List;
```

```
function Num_Size(N: Integer) return Integer is
begin   -- allows for leading space
   return Integer'Image(N)'Length;
end Num_Size;
```

```
with Page;  with Num_Size;
with Lists;  use Lists;
with Ada.Text_IO, Ada.Integer_Text_IO; use Ada;
```

```
procedure Print_List(L: in List;
                     Max_Value: in Integer) is
   Temp: List := L;
   Value: Integer;
   Count: Integer := 0;
   Field: constant Integer :=
                              Num_Size(Max_Value);
begin
   Text_IO.New_Line;
   loop
      exit when Is_Empty(Temp);
      Take_From_List(Temp, Value);
      Count := Count + Field;
      if Count > Page.Width then
         Text_IO.New_Line;
         Count := Field;
      end if;
      Integer_Text_IO.Put(Value, Field);
   end loop;
   Text_IO.New_Line(2);
end Print_List;
```

```
with Page;  with Num_Size;
with Trees;  use Trees;
with Ada.Text_IO;  use Ada;
procedure Print_Tree(T: in Tree;
                     Max_Value: in Integer) is

   Max_Width: Integer :=
         Num_Size(Max_Value) * 2**(Depth(T)-1);

   A: array (1 .. 4*Depth(T)-3) of Page.Line :=
                  (others => (others => ' '));

   procedure Put(N, Row, Col, Width: Integer) is
      Size: Integer := Num_Size(N);
      Offset: Integer := (Width - Size + 1)/2;
      Digit: Integer;
      Number: Integer := N;
   begin
      if Size > Width then
         for I in 1 .. Integer'Max(Width, 1) loop
            A(Row)(Col+I-1) := '*';
         end loop;
      else
         A(Row)(Col+Offset..Col+Offset+Size-1) :=
                              Integer'Image(Number);
      end if;
   end Put;

   procedure Do_It(T: Tree;
                   Row, Col, W: Integer) is
      Left: Tree := Left_Subtree(T);
      Right: Tree := Right_Subtree(T);
   begin
      Put(Node_Value(T), Row, Col, W);
      if not (Is_Empty(Left) and
                              Is_Empty(Right)) then
         A(Row+1)(Col+W/2) := '|';
      end if;

      if not Is_Empty(Left) then
         A(Row+2)(Col+W/4 .. Col+W/2) :=
                        (others => '-');
         A(Row+3)(Col+W/4) := '|';
         Do_It(Left, Row+4, Col, W/2);
      end if;

      if not Is_Empty(Right) then
         A(Row+2)(Col+W/2 .. Col+3*W/4) :=
                        (others => '-');
         A(Row+3)(Col+3*W/4) := '|';
         Do_It(Right, Row+4, Col+W/2, (W+1)/2);
      end if;
   end Do_It;

begin
   if Max_Width > Page.Width then
      Max_Width := Page.Width;
   end if;
   Do_It(T, 1, 1, Max_Width);
   Text_IO.New_Line;
   for I in A'Range loop
      Text_IO.New_Line
      Text_IO.Put(A(I));
   end loop;
   Text_IO.New_Line(2);
end Print_Tree;
```

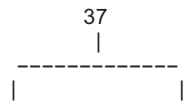
```
with Lists, Trees;  use Lists, Trees;
procedure Convert_List_To_Tree(L: in List;
                               T: out Tree) is

   Temp: List := L;
   Value: Integer;
begin
   Clear(T);
   loop
      exit when Is_Empty(Temp);
      Take_From_List(Temp, Value);
      Insert(T, Value);
   end loop;
end Convert_List_To_Tree;
```

```
with Lists, Trees;  use Lists, Trees;
procedure Convert_Tree_To_List(T: in Tree;
                               L: out List) is
   Right_L: List;
begin
   if Is_Empty(T) then Clear(L); return; end if;
   Convert_Tree_To_List(Left_Subtree(T), L);
   Append(L, Make_List(Node_Value(T)));
   Convert_Tree_To_List(Right_Subtree(T), Right_L);
   Append(L, Right_L);
end Convert_Tree_To_List;
```

```
with Lists, Read_List, Print_List;
with Trees, Print_Tree;
with Convert_List_To_Tree;
with Convert_Tree_To_List;
with Ada.Text_IO;  use Ada.Text_IO;
procedure Sylvan_Sorter is
   The_List: Lists.List;
   The_Tree: Trees.Tree;
   Max_Value: Integer;
begin
   Put("Welcome to the Sylvan Sorter");
   New_Line(2);
   loop
      Put_Line("Enter list of positive integers" &
                                        " ending with 0");
      Read_List(The_List, Max_Value);
      exit when Lists.Is_Empty(The_List);
      Print_List(The_List, Max_Value);
      Convert_List_To_Tree(The_List, The_Tree);
      Print_Tree(The_Tree, Max_Value);
      Convert_Tree_To_List(The_Tree, The_List);
      Print_List(The_List, Max_Value);
   end loop;
   New_Line;
   Put_Line("Finished");  Skip_Line(2);
end Sylvan_Sorter;
```

This program has deliberately been written in a rather extravagant manner using recursion quite freely. The list processing is particularly inefficient because the list is very simply linked and so adding an item to the end requires traversal of the list each time (this is in procedure Append). It would clearly be better to use a structure with links to both ends of the list as illustrated in Exercise 12.5(**3**).

Most of the program is straightforward except for the procedure Print_Tree. This has internal subprograms Put and Do_It. The basic problem is that the obvious approach of printing the tree recursively by first printing the left subtree then the node value and then the right subtree requires going up and down the page. However, Text_IO always works down the page and so cannot be used directly. Accordingly, we declare an array of strings A sufficient to hold all of the tree; this array is initialized to all spaces. The number of rows is computed by using the function Depth which returns the depth of the tree. Each fragment of tree is printed in the form



and so takes four lines. Hence the number of lines to be output is 4*Depth(T)–3.

The maximum value in the tree is remembered in Max_Value when reading the numbers in the first place. The minimum field width to print all numbers satisfactorily is then computed as Num_Size(Max_Value) which allows at least one space between adjacent numbers. This is then used to compute the overall width of the tree. If this is less than the page width then all is well. If not then it is set to the page width and we carry on regardless. The procedure Do_It is then called.

The procedure Do_It has parameters indicating the row and column of the array A from where it is to start printing and the field width available. It calls itself recursively with suitable values of these parameters for the left and right subtrees. If either subtree is empty the corresponding lines and recursive call of itself are omitted. The node value is output by a call of the procedure Put.

The procedure Put also has format parameters; it uses the attribute Image and prints the number centrally within the field. If it is unable to print the number within the given field then it outputs asterisks in the Fortran tradition (at least one); this will only happen if the width of the overall tree had to be reduced because it exceeded the page width – even then it might not happen if the numbers at the deepest part of the tree are relatively small. If the tree is very deep then the lines will eventually overlap and the whole thing will degenerate into rows of lines and asterisks; but it will keep going on to the bitter end!

This program, although curious, does illustrate many algorithmic features of Ada, such as slices, recursion, the attributes Max and Image and so on.

A stylistic point is that in some subprograms we have used the technique of first testing for the special case and, having performed the appropriate action, then leaving the subprogram via a return statement. Some readers might consider this bad practice and that we should really use an else part for the normal action and avoid the use of return in the middle of the subprogram. In defence of the approach taken one can say that mathematical proofs often dispose of the simple case in this way and moreover it does take a lot less space and means that the main algorithm is less indented.

Another stylistic point is that many of the variables declared to be of type Integer could well be of subtype Natural or Positive. We leave to the reader the task of considering this improvement.

A final point is that the program makes much use of heap storage. If the implementation has no garbage collector then it will eventually run out of space and raise Storage_Error. We will return to this issue in Chapter 25.

Part 3

The Big Picture

This third part is the core of the book and is largely about abstraction. As mentioned in Chapter 1, the evolution of programming languages is essentially about understanding various aspects of abstraction and here we look in depth at the facilities for data abstraction, object oriented programming and programming by contract.

Chapter 12 shows how packages and private types are the keystone of Ada and can be used to control visibility by giving a client and server different views of an object. This chapter also introduces the simplest ideas of type derivation and inheritance and the very

important notion of a limited type which is a type (strictly a view of a type) for which copying is not permitted. Limited types are important for modelling those real-world objects for which copying is inappropriate. Chapter 13 then discusses the hierarchical library structure and the facilities for separate compilation. As mentioned earlier, these enable a program to be compiled in distinct units without any loss of strong typing between units.

A third complete program follows Chapters 12 and 13 and illustrates the use of the hierarchical library in the construction of an abstract data type and associated operations.

The basic facilities for object oriented programming are then introduced in Chapter 14. This covers type extension and inheritance, dynamic polymorphism, dispatching, class wide types, abstract types and interfaces. This chapter concentrates very much on the basic nuts and bolts, and a further discussion of OOP is deferred until Chapter 21 when other aspects of the language have been introduced.

Chapter 15 then covers exceptions and is followed by Chapter 16 which discusses the very important topic of contracts which are perhaps the most important new feature introduced in Ada 2012.

Chapter 17 is a detailed discussion of numeric types including modular types and the rather specialized fixed point types which were not discussed in Chapter 6.

Chapters 18 and 19 return to the theme of abstraction by considering two forms of parameterization. Chapter 18 discusses the parameterization of types by discriminants. Chapter 19 discusses genericity which enables both subprograms and packages to be parameterized in many different ways. Genericity provides static polymorphism checked at compile time in contrast to the dynamic polymorphism of type extension.

Chapter 20 completes the main discussion of core features by describing the concepts relating to tasking; these include tasks which are units which execute in parallel and protected objects which provide shared access to common data without risk of interference.

At this point all the important features of the language have been described and the main purpose of the final two chapters is to illustrate how the various features work together.

Chapter 21 discusses various aspects of OOP such as the use of interfaces and object constructors; it also outlines the subtle new iterators introduced in Ada 2012 which simplify the use of containers. Chapter 22 extends the discussion on tasking by introducing synchronized interfaces which combine the properties of OOP and concurrent programming; it also describes selected topics from the specialized annexes.

This part concludes with a program which illustrates various aspects of tasking and generics.

12 Packages and Private Types

The previous chapters described the small-scale features of Ada in some detail. These features correspond to the areas addressed by the pioneering languages prior to about 1975. This chapter and the next discuss the important concepts of abstraction and programming in the large which entered into languages generally around 1980. The more recent concepts of type extension, inheritance and polymorphism are dealt with in Chapter 14.

In this chapter we discuss packages (which is really what Ada is all about) and the important concept of a private type. The topic of library units and especially library packages is discussed in detail in the next chapter.

12.1 Packages

In this section we introduce the concept of an Abstract State Machine (ASM) and in the next section we extend this to Abstract Data Types (ADT). The general idea of abstraction is to distinguish the inner details of how something works from an external view of it. This is a common idea in everyday life. We can use a watch to look at the time without needing to know how it works. Indeed the case of the watch hides its inner workings from us.

One of the major problems with early simple languages, such as C and Pascal, is that they do not offer enough control of visibility. For example, suppose we have a stack implemented as an array plus a variable to index the current top element, thus

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

We might then declare a procedure Push to add an item and a function Pop to remove an item.

```
procedure Push(X: in Integer) is
begin
   Top := Top + 1;
   S(Top) := X;
end Push;

function Pop return Integer is
begin
   Top := Top – 1;
   return S(Top + 1);
end Pop;
```

In a simple block structured language it is not possible to have access to the subprograms Push and Pop without also having direct access to the variables S and Top. As a result we can bypass the intended protocol of only accessing the stack through calls of Push and Pop. For example we might make use of our knowledge of the implementation details and change the value of Top directly without corresponding changes to the array S. This could well result in a call of Pop returning a junk value which was not placed on the stack by an earlier call of Push.

The Ada package overcomes this by allowing us to place a wall around a group of declarations and only permit access to those which we intend to be visible. A package actually comes in two parts: the specification which gives the interface to the outside world, and the body which gives the hidden details.

The above example should be written as

```
package Stack is                          -- specification
   procedure Push(X: in Integer);
   function Pop return Integer;
end Stack;

package body Stack is                     -- body
   Max: constant := 100;
   S: array (1 .. Max) of Integer;
   Top: Integer range 0 .. Max;

   procedure Push(X: in Integer) is
   begin
      Top := Top + 1;
      S(Top) := X;
   end Push;

   function Pop return Integer is
   begin
      Top := Top – 1;
      return S(Top + 1);
   end Pop;

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

The package specification (strictly declaration) starts with the reserved word **package**, the name of the package and **is**. This is followed by declarations of the entities which are to be visible. (There might then be a private part as discussed in the next section.) A package specification finishes with **end**, its name (optionally) and the terminating semicolon. In the example we just have the declarations of the two subprograms Push and Pop.

The package body also starts with **package** but this is then followed by **body**, the name and **is**. We then have a normal declarative part, **begin**, sequence of statements, **end**, optional name and terminating semicolon.

In this example the declarative part contains the variables which represent the stack and the bodies of Push and Pop. The sequence of statements between **begin** and **end** is executed when the package is declared and can be used for initialization. If there is no need for an initialization sequence, the **begin** can be omitted. Indeed, in this example we could equally have performed the initialization by writing

```
Top: Integer range 0 .. Max := 0;
```

The package is another case where we need distinct subprogram declarations and bodies. Indeed, we cannot put a body into a package specification. Moreover, if a package specification contains the specification of a subprogram, then the package body must contain the corresponding subprogram body; we say that the body completes the subprogram. We can think of the package specification and body as being just one large declarative region with only some items visible. However, a subprogram body can be declared in a package body without its specification having to be given in the package specification. Such a subprogram would be internal to the package and could only be called from within, either from other subprograms, some of which would presumably be visible, or perhaps from the initialization sequence.

There is one variation to the rule that the whole of a subprogram cannot appear in a package specification and that concerns expression functions which were introduced in Section 10.1. A very short function can be written in one lump thus

```
function Sign(X: Integer) return Integer is
  (if X > 0 then +1 elsif X < 0 then −1 else 0);
```

and such an expression function can appear in a package specification and does not need a body. Moreover, if we do indeed just give the specification of the function in a package specification thus

```
function Sign(X: Integer) return Integer;
```

then in the corresponding package body we can give the function body in full using the long form as at the start of Section 10.2 or we can use the abbreviated form provided by an expression function as the completion of the function.

Packages are typically declared at the outermost or library level of a program and often compiled separately as described in detail in Chapter 13. But packages may be declared in any declarative part such as that in a block, subprogram or indeed another package. If a package specification is declared inside another package specification then, just as for subprograms, the body of the inner one must be declared in the body of the outer one. And again both specification and body of the inner package could be in an outer package body in which case the inner package would not be visible outside that body.

Apart from the rule that a package specification cannot contain bodies, it can contain any of the other kinds of declarations we have met.

The elaboration of a package body consists simply of the elaboration of the declarations inside it followed by the execution of the initialization sequence if there is one. The package continues to exist until the end of the scope in which it is declared (or forever in the case of a library package). Entities declared inside the package have the same lifetime as the package itself. Thus the variables S and Top retain their values between successive calls of Push and Pop.

Now to return to the use of our package. The package itself has a name and the entities in its visible part (the specification) can be thought of as components of the package in some sense. It is natural therefore that, in order to call Push, we must also mention Stack. In fact the dotted notation is used. So we could write

```
declare
   package Stack is          -- specification
      ...                    -- and
      ...                    -- body
   end Stack;
begin
   ...
   Stack.Push(M);
   ...
   N := Stack.Pop;
   ...
end;
```

Inside the package we would call Push as just Push, but we could still write Stack.Push just as in Chapter 10 we saw how we could refer to a local variable X of procedure P as P.X. Inside the package we can refer to S or Stack.S, but outside the package, Max, S and Top are not accessible in any way.

Packages such as Stack are often referred to as abstract state machines. They contain internal state (in this case the hidden variables S and Top) and are abstract because access is through some protocol independent of how the state is implemented. The term encapsulation is also used to refer to the way in which a package encapsulates the items inside it.

It would in general be painful always to have to write Stack.Push to call Push from outside. Instead we can write **use** Stack; as a sort of declaration and we may then refer to Push and Pop directly. The use clause could follow the declaration of the specification of Stack in the same declarative part or could be in another declarative part where the package is visible. So we could write

```
declare
   use Stack;
begin
   ...
   Push(M);
   ...
   N := Pop;
   ...
end;
```

The use clause is like a declaration and similarly has a scope to the end of the block. Outside we would have to revert to the dotted notation. We could have an inner use clause referring to the same package – it would do no harm.

Two or more packages could be declared in the same declarative part. Generally, we could arrange all the specifications together and then all the bodies, or alternatively the corresponding specifications and bodies could be together. Thus we could have spec A, spec B, body A, body B, or spec A, body A, spec B, body B. The rules governing the order are simply

- linear elaboration of declarations,
- specification must precede body for same package (or subprogram).

The specification of a package may contain things other than subprograms. Indeed an important case is where it does not contain subprograms at all but merely a group of related variables, constants and types. In such a case the package needs no body. It does not provide any hiding properties but merely gives commonality of naming. (Note that a body could be provided; its only purpose would be for initialization.)

However, a package declared at the library level is only allowed to have a body if it requires one for some reason such as providing the body for a subprogram declared in the specification. This avoids some awkward surprises which could otherwise occur as will be explained in Section 27.3.

As an example we could provide a package containing the type Day and some useful related constants.

```
package Diurnal is
   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   subtype Weekday is Day range Mon .. Fri;
   Tomorrow: constant array (Day) of Day :=
                                   (Tue, Wed, Thu, Fri, Sat, Sun, Mon);
   Next_Work_Day: constant array (Weekday) of Weekday :=
                                            (Tue, Wed, Thu, Fri, Mon);
end Diurnal;
```

Several examples of packages without bodies occur in the predefined library such as Ada.Numerics which we met in Chapter 4. We shall see other important examples when we deal with abstract tagged types and interfaces in Chapter 14.

A final point. A subprogram cannot be called successfully during the elaboration of a declarative part if its body appears later. This did not prevent the mutual recursion of the procedures F and G in Section 10.6 because in that case the call of F only occurred when we executed the sequence of statements of the body of G. But it can prevent the use of a function in an initial value. So

```
function A return Integer;
I: Integer := A;
```

is incorrect, and would result in Program_Error being raised.

A similar rule applies to subprograms in packages. If we call a subprogram from outside a package but before the package body has been elaborated, then Program_Error will be raised.

Exercise 12.1

1 The sequence defined by

$$X_{n+1} = (X_n \times 5^5) \bmod 2^{13}$$

provides a crude source of pseudorandom numbers. The initial value X_0 should be an odd integer in the range 0 to 2^{13}.

Write a package Random containing a procedure Init to initialize the sequence and a function Next to deliver the next value in the sequence.

2 Write a package Complex_Numbers which makes visible the type Complex, a constant I = √–1, and functions +, –, *, / acting on values of type Complex. See Exercise 9.2(**2**).

12.2 Private types

We have seen how packages enable us to hide internal objects from the user of a package. Private types enable us to create Abstract Data Types in which the details of the construction of a type are hidden from the user.

In Exercise 12.1(**2**) we wrote a package Complex_Numbers providing a type Complex, a constant I and some operations on the type. The specification of the package was

```
package Complex_Numbers is
   type Complex is
      record
         Re, Im: Float;
      end record;

   I: constant Complex := (0.0, 1.0);

   function "+" (X: Complex) return Complex;      -- unary +
   function "–" (X: Complex) return Complex;      -- unary –

   function "+" (X, Y: Complex) return Complex;
   function "–" (X, Y: Complex) return Complex;
   function "*" (X, Y: Complex) return Complex;
   function "/" (X, Y: Complex) return Complex;
end;
```

The trouble with this formulation is that the user can make use of the fact that the complex numbers are held in cartesian representation. Rather than always using the complex operator "+", the user could also write things like

```
C.Im := C.Im + 1.0;
```

rather than the more abstract

```
C := C + I;
```

In fact, with the above package, the user has to make use of the representation in order to construct values of the type.

We might wish to prevent use of knowledge of the representation so that we could change the representation to perhaps polar form at a later date and know that the user's program would still be correct. We can do this with a private type. Consider

```
package Complex_Numbers is
   type Complex is private;                    -- visible part
   I: constant Complex;
   function "+" (X: Complex) return Complex;
   function "–" (X: Complex) return Complex;
   function "+" (X, Y: Complex) return Complex;
   function "–" (X, Y: Complex) return Complex;
   function "*" (X, Y: Complex) return Complex;
   function "/" (X, Y: Complex) return Complex;
   function Cons(R, I: Float) return Complex;
   function Re_Part(X: Complex) return Float;
   function Im_Part(X: Complex) return Float;
private                                        -- private part
   type Complex is
      record
         Re, Im: Float;
      end record;
   I: constant Complex := (0.0, 1.0);
end;
```

The part of the package specification before the reserved word **private** is the visible part and gives the information available externally to the package. The type Complex is declared to be private. This means that outside the package nothing is known of the details of the type. The only operations available are assignment, = and /= plus those added by the writer of the package as subprograms specified in the visible part.

We may also declare constants of a private type such as I in the visible part. The initial value cannot be given in the visible part because the details of the type are not yet known. Hence we just state that I is a constant; we call it a deferred constant.

After **private** comes the so-called private part in which we have to give the details of types declared as private and give the initial values of any deferred constants.

A private type can be implemented in any way consistent with the operations visible to the user. It can be a record as we have shown; equally it could be an array, an enumeration type and so on; it could even be declared in terms of another private type. In our case it is fairly obvious that the type Complex is naturally implemented as a record; but we could equally have used an array of two components such as

```
type Complex is array (1 .. 2) of Float;
```

After the full type declaration of a private type we have to give full declarations of any deferred constants of the type including their initial values.

It should be noted that as well as the functions +, –, * and / we have also provided Cons to create a complex number from its real and imaginary components

and Re_Part and Im_Part to return the components. Some such functions are necessary because the user no longer has direct access to the internal structure of the type. Of course, the fact that Cons, Re_Part and Im_Part correspond to our thinking externally of the complex numbers in cartesian form does not prevent us from implementing them internally in some other form as we shall see in a moment.

The body of the package is as shown in the answer to Exercise 12.1(**2**) plus the additional functions which are trivial. It is therefore

```
package body Complex_Numbers is

   ...   -- unary + -

   function "+" (X, Y: Complex) return Complex is
   begin
      return (X.Re + Y.Re, X.Im + Y.Im);
   end "+";

   ...   -- and - * / similarly

   function Cons(R, I: Float) return Complex is
   begin
      return (R, I);
   end Cons;

   function Re_Part(X: Complex) return Float is
   begin
      return X.Re;
   end Re_Part;

   ...   -- and Im_Part similarly

end Complex_Numbers;
```

The package Complex_Numbers could be used in a fragment such as

```
declare
   use Complex_Numbers;
   C, D: Complex;
   F: Float;
begin
   C := Cons(1.5, -6.0);
   D := C + I;                  -- Complex +
   F := Re_Part(D) + 6.0;       -- Float +
   ...
end;
```

Outside the package we can declare variables and constants of type Complex in the usual way. Note the use of Cons to create the effect of a complex literal. We cannot, of course, do mixed operations between the type Complex and the type Float. Thus we cannot write

```
C := 2.0 * C;
```

but instead must write

```
C := Cons(2.0, 0.0) * C;
```

If this is felt to be tedious we could add further overloadings of the operators to allow mixed operations.

Let us suppose that for some reason we now decide to represent the complex numbers in polar form. The visible part of the package will be unchanged but the private part could now become

```
private
   π: constant := 3.14159_26536;
   type Complex is
      record
         R: Float;
         θ: Float range 0.0 .. 2.0*π;
      end record;
   I: constant Complex := (1.0, 0.5*π);
end;
```

Note how the named number π is for convenience declared in the private part; anything other than a body can be declared in a private part if it suits us – we are not restricted to just declaring the types and constants in full. Things declared in the private part are also available in the body. An alternative to declaring our own number π is to use the value in the package Ada.Numerics or at least to initialize our own number with that value; see Section 13.7.

The body of our package Complex_Numbers will now need completely rewriting. Some functions will become simpler and others will be more intricate. In particular it will be convenient to provide a function to normalize the angle θ so that it lies in the range 0 to 2π. The details are left for the reader.

However, since the visible part has not been changed the user's program will not need changing; we are assured of this since there is no way in which the user could have written anything depending on the details of the private type. Nevertheless, as we shall see in the next chapter, the user's program will need recompiling because of the general dependency rules. This may seem slightly contradictory but the compiler needs the information in the private part in order to be able to allocate storage for objects of the private type declared in the user's program. If we change the private part the size of the objects could change and then the object code of the user's program would change even though the source was the same and compiled separately.

We can therefore categorize the three distinct parts of a package as follows: the visible part which gives the logical interface to clients, the private part which gives the physical interface, and the body which gives the implementation details. When Ada was first designed some thought was given to the idea that these three parts might be written as distinct units but it was dismissed on the grounds that it would be very tedious.

It is important to appreciate that there are two quite different views of a private type such as Complex. Outside the package we know only that the type is private; inside the package and after the full type declaration we know all the properties

implied by the declaration. Thus we see that we have two different views of the type according to where we are and hence which declaration we can see; these are known as the *partial view* and the *full view* respectively.

Between a private type declaration and the later full type declaration, the type is in a curiously half-defined state (technically it is not frozen which means that we do not yet know all about it). Because of this there are severe restrictions on its use, the main ones being that it cannot be used to declare variables or allocate objects. But it can be used to declare deferred constants, other types and subtypes and subprogram specifications (also entries of tasks and protected types).

Thus we could write

```
type Complex_Array is array (Integer range <>) of Complex;
```

and then

```
C: constant Complex_Array;
```

in the visible part. But until the full declaration is given we cannot declare variables of the type Complex or Complex_Array.

However, we can declare the specifications of subprograms with parameters of the types Complex and Complex_Array and can even supply default expressions. Such default expressions can use deferred constants and functions.

Deferred constants provide another example of a situation where information has to be repeated and thus be consistent. Both deferred and full declarations must have the same type; if they both supply constraints (directly or using a subtype) then they must statically match (as defined in Section 8.2). However, it is possible for the deferred declaration just to give the type and then for the full declaration to impose a constraint. Similarly if the deferred constant is marked as aliased or has a null exclusion then the full one must match. Deferred constants can also be of an anonymous access type.

So in the case of the array C above the full declaration might be

```
C: constant Complex_Array(1 .. 10) := ... ;
```

or even

```
C: constant Complex_Array := ... ;
```

where in the latter case the bounds are taken from the initial value.

The type Complex_Array illustrates the general distinction between the partial view and the full view that you can only use what you can see. As an example consider the operator "<". (Remember that "<" only applies to arrays if the component type is discrete.) Outside the package we cannot use "<" since we do not know whether or not the type Complex is discrete. Inside the package we find that it is not discrete and so still cannot use "<". If it had been discrete we could have used "<" after the full type declaration but of course we still could not use it outside. On the other hand, slicing is applicable to all one-dimensional arrays and so can be used both inside and outside the package.

Deferred constants are not necessarily of a private type, any constant in a package specification can be deferred to the private part. Indeed the constant C is not of a private type since Complex_Array is not itself private.

Another use for a deferred constant is to provide read-only access to a variable declared in the private part. Consider

```
   type Int_Ptr is access constant Integer;
   The_Ptr: constant Int_Ptr;                    -- deferred constant
private
   The_Variable: aliased Integer;
   The_Ptr: constant Int_Ptr := The_Variable'Access;
```

The external user can read the value of The_Variable via The_Ptr but cannot change it. However, The_Variable can of course be updated by subprograms declared in the package or by any initialization sequence of the package body. This technique might be used to provide access to a 'constant' table of data which has to be computed dynamically. This is another example of having two views of something, in this case a constant view and a variable view.

Note that the above could be written using anonymous access types thus

```
   The_Ptr: constant access constant Integer;    -- deferred constant
private
   The_Variable: aliased Integer;
   The_Ptr: constant access constant Integer := The_Variable'Access;
```

which provides a neat illustration of the use of **constant access constant**.

A deferred constant can also be used to import a constant from an external system; see Section 25.5.

Note also that we could provide a parameterless function

```
function I return Complex;
```

instead of a deferred constant. One advantage is that we can change the returned value without changing the package specification and so having to recompile the user's program. Of course in the case of I we are unlikely to need to change the value anyway! Another advantage is that functions can be overloaded so that we could have several constants (of different types) with the same name such as Unit or Empty. Yet another advantage is mentioned in the next section when we consider derived types. In general, a function (with aspect Inline; see Section 25.1) is nearly always better than a deferred constant.

We conclude this section by summarizing the overall structure of a package regarding the declaration of subprograms. A package is generally in three parts: the visible part of the specification, the private part of the specification, and the body.

A procedure comes usually in two parts: its specification and its body. The specification says what it does whereas the body says how it does it. Sometimes it is not necessary to give a distinct specification since the first part of the body repeats it anyway. If there is a distinct specification then we say that the body is the completion of the procedure. Possible arrangements are

- Procedure spec in visible part of package; body in package body. Procedure can be called from outside the package and from within body.
- Procedure spec in private part of package; body in package body. Procedure can be called from places that can see the private part and from within the body.

- Procedure spec in package body; body in package body. Procedure can be called from within body.
- There is no procedure spec; body in package body. Procedure can be called from within body.

The above also applies to functions. However, the situation is more flexible because of the introduction of expression functions. An expression function can be used just as a shorthand for a body. But an expression function can also be used as a complete function without a specification or to complete a distinct specification where both specification and completion are in the package specification. An important situation is where the specification is in the visible part and is completed by an expression function in the private part and so has access to the entities in the private part; see Section 16.3 for an example. Moreover, in recursive situations we might declare both in the visible part; some examples will be found in the *Rationale*.

Exercise 12.2

1 Write further functions "*" to allow mixed multiplication of real and complex numbers. Consider functions declared both inside and outside the package.

2 Rewrite the fragment of user program for complex numbers omitting the use clause.

3 Complete the package Rational_Numbers whose visible part is

```
package Rational_Numbers is

   type Rational is private;
   function "+" (X: Rational) return Rational;          -- unary +
   function "–" (X: Rational) return Rational;          -- unary –
   function "+" (X, Y: Rational) return Rational;
   function "–" (X, Y: Rational) return Rational;
   function "*" (X, Y: Rational) return Rational;
   function "/" (X, Y: Rational) return Rational;        -- binary division

   function "/" (X: Integer; Y: Positive) return Rational;     -- constructor
   function Numerator(R: Rational) return Integer;
   function Denominator(R: Rational) return Positive;

private
   ...
end;
```

A rational number is a number of the form N/D where N is an integer and D is a positive integer. For predefined equality to work it is essential that rational numbers are always reduced by cancelling out common factors. This may be done using the function GCD of Exercise 10.1(**7**). Ensure that an object of type Rational has an appropriate default value of zero.

4 Why does

```
function "/" (X: Integer; Y: Positive) return Rational;
```

not hide the predefined integer division?

12.3 Primitive operations and derived types

This seems a good moment to discuss the question of which operations really belong to a type. In Section 6.3 we noted that 'A type is characterized by a set of values and a set of primitive operations ...'. The set of values is pretty obvious but the set of primitive operations is not intuitively obvious nor is it obvious why the definition of such a set should be important.

The key to why we need to define the set of primitive operations of a type lies in the concept of a class which we mentioned in Chapter 3. Types can be grouped into various categories with common properties. These categories play two roles in Ada, as the basis for type derivation and extension and also as the basis for formal parameter types of generics. We will introduce the ideas by first considering the simplest form of derived types; the more flexible tagged types which can be extended will be discussed in Chapter 14 and generic parameters will be discussed in Chapter 19.

Sometimes it is useful to introduce a new type which is similar in most respects to an existing type but is nevertheless a distinct type. If T is a type we can write

```
type S is new T;
```

and then S is said to be a derived type and T is the parent type of S. Remember that we are only discussing untagged types in this section.

A type plus all the types derived from it form a category of types called a derivation class or simply a class. Classes are very important and the types in a class share many properties. For example if T is a record type then S will be a record type whose components have the same names and so on.

The set of values of a derived type is a copy of the set of values of the parent. An important instance of this is that if the parent type is an access type then the derived type is also an access type and they share the same storage pool. Note that we say that the set of values is a copy; this reflects that they are truly different types and values of one type cannot be assigned to objects of the other type; however, as we shall see in a moment, conversion between the two types is possible. The notation for literals and aggregates (if any) is the same and any default initial expressions for the type or its components are the same.

The primitive operations of a type are

- predefined operations such as assignment, predefined equality, appropriate attributes and so on, and
- for a derived type, primitive operations inherited from its parent (these can be overridden), and
- for a type (immediately) declared in a package specification, subprograms with a parameter or result of the type also declared in the package specification – including any private part.

The predefined types follow the same rules, for example the primitive operations of Integer include the operators such as "+", "–" and "<" which are notionally declared in Standard along with Integer.

Enumeration literals are also primitive operations of a type since they are considered to be parameterless functions returning a value of the type. So in the case

of the type Boolean, False and True are primitive operations. It is rather as if there were functions such as

```
function True return Boolean is
begin
   return Boolean'Val(1);
end;
```

So the general idea is that a type has certain predefined primitive operations, it inherits some from its parent (if any) and more can be added.

Note especially that an operation inherited from the parent type can be overridden by explicitly declaring a new subprogram with the same identifier in the same declarative region as the derived type declaration itself (a declarative region is a list of declarations such as in a block or subprogram or a package specification and body taken together). The new subprogram must have type conformance with the overridden inherited operation.

A subtle point is that inherited primitive operations are also intrinsic so that the Access attribute cannot be applied to them whereas it can be applied to explicitly declared operations. This rule does not apply to tagged types (see Section 14.1).

There are a couple of minor rules which arise from consideration of the freezing of the type (see Section 25.1). We cannot derive from a private type until after its full type declaration. Also if we derive from a type in the same package specification as that in which it is declared then the derived type will inherit all the primitive operations of the parent declared so far. Further operations might then be declared for the parent but these will not be inherited by the derived type.

Although derived types are distinct, nevertheless because of their close relationship, a value of one type can be directly converted to another type if they have a common ancestor. Consider

```
type Light is new Colour;
type Signal is new Colour;
type Flare is new Signal;
```

These types form a hierarchy rooted at Colour. We can convert from any one type to another and do not have to give the individual steps. So we can write

```
L: Light;
F: Flare;
...
F := Flare(L);
```

and we do not need to laboriously write

```
F := Flare(Signal(Colour(L)));
```

The introduction of derived types extends the possibility of conversion between array types discussed in Section 8.2. In fact a value of one array type can be converted to another array type if the component subtypes statically match and the index types are the same or convertible to each other.

The reader might wonder at the benefit of derived types and why we might wish to have one type very like another.

One use for derived types is when we want to use an existing type, but wish to avoid the accidental mixing of objects of conceptually different types. Suppose we wish to count apples and oranges. Then we could declare

```
type Apples is new Integer;
type Oranges is new Integer;

No_Of_Apples: Apples;
No_Of_Oranges: Oranges;
```

Since Apples and Oranges are derived from the type Integer they both have the operation "+". So we can write statements such as

```
No_Of_Apples := No_Of_Apples + 1;
No_Of_Oranges := No_Of_Oranges + 1;
```

but we cannot inadvertently write

```
No_Of_Apples := No_Of_Oranges;
```

If we did want to convert the oranges to apples we would have to write

```
No_Of_Apples := Apples(No_Of_Oranges);
```

The numeric types will be considered in Chapter 17 in more detail but it is worth mentioning here that strictly speaking a type such as Integer has no literals. Literals such as 1 and integer named numbers are of a type known as *universal_integer* and implicit conversion to any integer type occurs if the context so demands. Thus we can use 1 with Apples and Oranges because of this implicit conversion and not because the literal is inherited.

Now suppose that we have overloaded procedures to sell apples and oranges

```
procedure Sell(N: Apples);
procedure Sell(N: Oranges);
```

Then we can write

```
Sell(No_Of_Apples);
```

but (much as the problem we had with cows and garments in Section 10.6), writing Sell(6); is ambiguous because we do not know which fruit we are selling. We can resolve the ambiguity by qualification thus

```
Sell(Apples'(6));
```

When a subprogram is inherited a new subprogram is not actually created. A call of the inherited subprogram is really a call of the parent subprogram; **in** and **in out** parameters are implicitly converted just before the call; **in out** and **out** parameters or a function result are implicitly converted just after the call. So

```
My_Apples + Your_Apples
```

is effectively

```
Apples(Integer(My_Apples) + Integer(Your_Apples))
```

Another example of the use of derived types to avoid errors will be found in the answer to Exercise 23.4(**2**) where they are used in order to distinguish the hands of Jack and Jill when they are playing paper, stone and scissors.

An important use of derived types is with private types when we wish to express the fact that a private type is to be implemented as an existing type such as Integer or Cell_Ptr. We shall see an example of this in Section 12.5.

Looking back at the type Complex of the previous section we can see that the primitive operations are assignment, predefined equality and inequality and the subprograms "+", ..., Im_Part declared in the package specification.

In Exercise 12.2(**2**) we noted that it was tedious to use the operators of the type Complex without a use clause since, given objects A, B and C of type Complex, we would have to write

```
C := Complex_Numbers."+"(A, B);
```

which is painful to say the least. However, there is a strong school of thought that use clauses are bad for you since they obscure the origin of entities. In order to alleviate this dilemma, Ada 95 introduced the so-called use type clause. This can be placed in a declarative part just like a use package clause and has similar scope. It makes just the primitive operators of a type directly visible. Thus we might write

```
declare
   use type Complex_Numbers.Complex;
   A, B, C: Complex_Numbers.Complex;
   ...
begin
   ...
   C := A + B;
```

Note that the full dotted notation is still required for referring to other entities in the package such as the function Cons and the type Complex itself.

A further form of use clause is also permitted in Ada 2012. This is the use all type clause written thus

```
use all type Complex_Numbers.Complex;
```

This makes directly visible all the operations of a type and not just those denoted by operators. It therefore includes Cons, Re_Part and Im_Part but not the type Complex itself since that is not a primitive operation.

Another example might be instructive. Suppose we have a package Palette containing a type Colour and a function for mixing colours thus

```
package Palette is
   type Colour is (Red, Orange, Yellow, Green, Blue, ... );
   function Mix(This, That: Colour) return Colour;
end Palette;
```

In another package without any use clauses, if we want to mix Red and Green to make a mucky colour we have to write the laborious

```
Mucky: Palette.Colour := Palette.Mix(Palette.Red, Palette.Green);
```

Adding a **use type** clause makes no difference but if we provide a **use all type** clause then we can write

```
Mucky: Palette.Colour := Mix(Red, Green);
```

Remember that enumeration literals behave as functions and so are primitive operations. But the type name itself still has to refer to the package name unless we provide a use package clause.

If we wished to introduce a distinct type derived from Complex for an electromagnetic application (much as we had distinct types to count apples and oranges) then we could perhaps write

```
type Field is new Complex;
```

and we could then ensure that values of the field would not inadvertently get mixed up with other complex numbers. However, it is important to note that although the type Field inherits all the operations of Complex, it does not inherit the constant I. To maintain the inheritance abstraction it is necessary to make I into a function.

A few words on constraints and null exclusions: we can derive from a subtype using the more general subtype indication. So we could have

```
type Chance is new Float range 0.0 .. 1.0;
```

in which case the underlying derived type is derived from the underlying base type. It is as if we had written

```
type anon is new Float;
subtype Chance is anon range 0.0 .. 1.0;
```

and so Chance denotes a constrained subtype of the (anonymous) derived type. The set of values of the new derived type is actually (a copy of) the full set of values of the type Float. The operations of the type "+", ">" and so on also work on the full unconstrained set of values. So given

```
C: Chance;
```

we can legally write

```
C > 2.0
```

even though 2.0 could never be successfully assigned to C. The Boolean expression is, of course, always false (unless C was uninitialized and by chance had a silly value).

As a further example of constraints it is instructive to consider the specification of an inherited subprogram in more detail. It is obtained from that of the parent by simply replacing all instances of the parent base type in the original specification by the new derived type. Subtypes are replaced by equivalent subtypes with corresponding constraints and default initial expressions are converted by adding a type conversion. Any parameters or result of a different type are left unchanged. As an abstract example consider

```
type T is ... ;
subtype S is T range L .. R;
```

```
function F(X: T;  Y: T := E;  Z: Q) return S;
```

where E is an expression of type T and the type Q is quite unrelated. If we write

```
type TT is new T;
```

then it is as if we had also written

```
subtype SS is TT range TT(L) .. TT(R);
```

and the specification of the inherited function F will then be

```
function F(X: TT;  Y: TT := TT(E);  Z: Q) return SS;
```

in which we have replaced T by TT, S by SS, added the conversion to the expression E but left the unrelated type Q unchanged. Note that the parameter names are naturally the same.

Derived types are in some ways an alternative to private types. Derived types have the advantage of inheriting literals but they often have the disadvantage of inheriting too much. Thus, we could derive types Length and Area from Float.

```
type Length is new Float;
type Area is new Float;
```

We would then be prevented from mixing lengths and areas but we would have inherited the ability to multiply two lengths to give a length and to multiply two areas to give an area as well as hosts of irrelevant operations such as exponentiation. Such unnecessary inherited operations can be eliminated by overriding them with abstract subprograms such as

```
function "*" (Left, Right: Length) return Length is abstract;
```

An abstract subprogram has no body and can never be called; any attempt to do so is detected during compilation. Abstract subprograms are very important with tagged types and the rules are rather different as we shall see in Chapter 14. An important difference is that the related concepts of abstract types and interfaces do not apply to untagged types.

Two more topics mainly used with tagged types but also applicable to untagged types deserve mention. One is the concept of a null procedure. We can write

```
procedure Option(X: in T) is null;
```

Such a procedure declaration cannot be given a body but behaves as if it had a body consisting of

```
procedure Option(X: in T) is
begin
   null;
end Option;
```

If declared in the same specification as X then it will be inherited by a type derived from X and it could then be overridden by a procedure with a conventional body.

There is of course no such thing as a null function because a function must return a result.

Another feature is that a subprogram can be given a so-called overriding indicator. Suppose that the type Colour has a primitive procedure Option thus

```
procedure Option(C: in Colour);
```

then when we declare the type Signal derived from Colour we can replace the inherited procedure with a new version. We can write

```
overriding
procedure Option(S: in Signal);
```

The overriding indicator then confirms that we did indeed mean to override the inherited operation. This is a safeguard against a number of errors. Thus if we inadvertently wrote Operation rather than Option so that the procedure did not override then the program would fail to compile. On the other hand, if we do introduce a new operation then we can write

```
not overriding
procedure Operation(S: in Signal);
```

Overriding indicators are always optional (see Section 14.6). They can also be used on a subprogram body and in other constructions such as renaming (Section 13.7), generic instantiations (Section 19.1) and with entries (Section 22.3).

We finish this section by mentioning a curious anomaly concerning the type Boolean which really does not matter so far as the normal user is concerned. If we derive a type from Boolean then the predefined relational operators =, < and so on continue to deliver a result of the predefined type Boolean whereas the logical operators **and**, **or**, **xor** and **not** are inherited normally and deliver a result of the derived type. We cannot go into the reason here other than to say that it relates to the fact that the relational operators for all types return a result of type Boolean (see also Section 23.1). However, it does mean that the theorem of Exercise 6.7(**3**) that **xor** and /= are equivalent only applies to the type Boolean and not to a type derived from it.

Exercise 12.3

1 Declare a package Metrics containing types Length and Area with appropriate redeclarations of the various operations "*", "/" and "**".

12.4 Equality

In the previous section we saw that predefined equality was an operation of most types; however, it is sometimes inappropriate and needs to be overridden. Remember from Section 10.2 that if we do redefine "=" and it returns the type Boolean then a corresponding function "/=" is automatically implied.

As an example, recall that in Section 12.1 we introduced the package Stack as an abstract state machine which declared a single stack. It is perhaps more useful to define a package providing an abstract data type so that we can declare several stacks. Consider

```
package Stacks is
   type Stack is private;
   procedure Push(S: in out Stack; X: in Integer);
   procedure Pop(S: in out Stack; X: out Integer);
   function "=" (S, T: Stack) return Boolean;
private
   Max: constant := 100;
   type Integer_Vector is array (Integer range <>) of Integer;

   type Stack is
      record
         S: Integer_Vector(1 .. Max);
         Top: Integer range 0 .. Max := 0;
      end record;

end;
```

Each object of type Stack is a record containing an array S and integer Top. Note that Top has a default initial value of zero. This ensures that when we declare a stack object, it is correctly initialized to be empty. Note also the introduction of the type Integer_Vector because a record component may not be of an anonymous array type.

The body of the package could be

```
package body Stacks is

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

   procedure Pop(S: in out Stack; X: out Integer) is
   begin
      X := S.S(S.Top);
      S.Top := S.Top – 1;
   end Pop;

   function "=" (S, T: Stack) return Boolean is
   begin
      if S.Top /= T.Top then
         return False;
      end if;
      for I in 1 .. S.Top loop
         if S.S(I) /= T.S(I) then
            return False;
         end if;
      end loop;
      return True;
   end "=";

end Stacks;
```

This example illustrates many points. The parameter S of Push has mode **in out** because we need both to read from and to write to the stack. We have made Pop a procedure for uniformity with Push although in Ada 2012 it could be a function declared thus

```
function Pop(S: in out Stack) return Integer is ...
```

which might be more convenient. So in Ada 2005 we have to write

```
Pop(A_Stack, Y);          -- using a procedure Pop
```

whereas in Ada 2012 we can write

```
Y := Pop(A_Stack);        -- using a function Pop
```

Remember that in Ada 2012, the ability for a function to have parameters of any mode does not extend to operators. However, "=" can be a function because we only need to read the values of the two stacks and not to update them.

The function "=" has the interpretation that two stacks are equal only if they have the same number of items and the corresponding items have the same value. It would obviously be quite wrong to compare the whole records because the unused components of the arrays would also be compared. Incidentally, the function body could be written more simply using slices – see Exercise 12.4(**3**).

The type Stack is a typical example of a data structure where the value of the whole is more than just the sum of the parts; the interpretation of the array S depends on the value of Top. Cases where there is such a relationship usually need redefinition of equality.

A minor point is that we are using the identifier S in two ways: as the name of the formal parameter denoting the stack and as the array inside the record. There is no conflict because, although the scopes overlap, the regions of visibility do not as is explained in Section 13.6. Of course, it is somewhat confusing for the reader and not good practice but it illustrates the freedom of choice of record component names.

The package could be used in a fragment such as

```
declare
   use Stacks;
   My_Stack: Stack;
   Your_Stack: Stack;
   …
begin
   Push(My_Stack, N);
   …
   Push(Your_Stack, M);
   …
   if My_Stack = Your_Stack then
      …
   end if;
   …
end;
```

It is interesting to consider how we might check that a stack is empty. We could compare it against a constant Empty declared in the visible part of the package. But a much better technique would be to provide a function Is_Empty and possibly a corresponding function Is_Full.

Having declared our own equality for the type Stack we might decide to declare a type such as

```
type Stack_Array is array (Integer range <>) of Stack;
```

This array type has predefined equals (see Section 8.2) but it is important to note that it works in terms of the original predefined equals for the type Stack and not our redefined version. We thus probably need to redefine equality for such an array type (this will also apply to slices of that type). We might even choose to define equality to return a Boolean array with each element indicating the equality of the matching components. Remember that if we do define equality to return a type other than just Boolean then a new version of "/=" will not automatically be provided and so we might need to redefine that as well.

The principle that objects of a composite type T are equal if and only if the components are equal is often called composability of equality. We have seen that equality does not compose for arrays such as Stack_Array where equality has been redefined for the component type. But equality does compose for record types. So if a component of a record type has had equality redefined then this redefinition will apply to the record type also. (In Ada 2005, composability of record types applied only if tagged.) It is important in some applications to know that types do compose and in fact all the private types declared in the predefined library are composable.

Situations where equality needs redefinition usually only arise with composite types where the meaning of the whole is more than just the sum of the parts. Nevertheless, we can redefine equality for elementary types. If we do then the meaning of certain intrinsic structures effectively involving predefined equality is not changed. Thus the case statement always uses predefined equality in choosing the sequence to be obeyed.

Another interesting example is provided by a type such as Rational of Exercise 12.2(**3**); it would be quite reasonable to allow manipulation – including assignment – of values which were not reduced provided that equality is suitably redefined. If we wish to use predefined equality then we must reduce all values to a canonical form in which component by component equality is satisfactory. In the case of the type Stack implemented as an array, a suitable form is one in which all unused elements of the array had a standard dummy value such as zero.

Exercise 12.4

1 Rewrite the specification of Stacks to include a constant Empty in the visible part.

2 Write functions Is_Empty and Is_Full for Stacks.

3 Rewrite the **function** "=" (S, T: Stack) **return** Boolean; using slices.

4 Write a suitable body for

```
function "=" (A, B: Stack_Array) return Boolean;
```

Make it conform to the normal rules for array equality which we mentioned towards the end of Section 8.2. Also write appropriate equality functions returning an array of Boolean values.

5 Rewrite Stacks so that predefined equality is satisfactory.

6 Redefine "=" for the type Rational of Exercise 11.2(**3**) assuming that the package Rational_Numbers does not reduce values to a canonical form using GCD. Would this be sensible?

12.5 Limited types

The primitive operations of a record type or private type can be completely restricted to just those declared in the package specification along with the type itself. We write one of

```
type T is limited private;
type R is limited record ... end record;
```

In such a case assignment (copying) and predefined = and /= are not defined for the type. Of course we can (re)define equality for any type anyway and so the important issue in deciding whether a type should be limited is whether we wish to prevent copying. Preventing copying is often important when a type is implemented in terms of access types or has components of access types.

A limited private type may be implemented as a nonlimited type in which case assignment (and predefined equality) will be available in the private part and body of the package. We thus see that limitedness is a property of a view of a type.

The advantage of making a private type limited is that the package writer has complete control over the objects of the type – the copying of resources can be monitored and so on. It also gives more freedom of implementation for the full type as we shall see in a moment.

We now consider an alternative formulation of the type Stack and suppose that we wish to impose no maximum stack size other than that imposed by the overall size of the computer. This can be done by representing the stack as an access type referring to a list as follows

```
package Stacks is
   type Stack is limited private;
   procedure Push(S: in out Stack; X: in Integer);
   procedure Pop(S: in out Stack; X: out Integer);
   function "=" (S, T: Stack) return Boolean;
private
   type Cell is
      record
         Next: access Cell;                    -- anonymous type
         Value: Integer;
      end record;
   type Stack is access all Cell;              -- named type
end;
```

```
package body Stacks is
   procedure Push(S: in out Stack; X: in Integer) is
   begin
      S := new Cell'(S, X);
   end;
   procedure Pop(S: in out Stack; X: out Integer) is
   begin
      X := S.Value;
      S := Stack(S.Next);                    -- conversion
   end;
   function "=" (S, T: Stack) return Boolean is ...
end Stacks;
```

We have avoided an incomplete type declaration by making the component Next of an anonymous type. This does however mean that a type conversion is required in Pop when assigning the component Next to S. In Ada 2005 this conversion must be named as shown but in Ada 2012 the name can be omitted since it cannot fail; see Section 11.7.

When the user declares a stack

```
S: Stack;
```

it automatically takes the default initial value **null** which denotes that the stack is empty. If we call Pop when the stack is empty then this will result in attempting to evaluate

```
null.Value
```

and this will raise Constraint_Error. The only way in which Push can fail is by running out of storage; an attempt to evaluate

```
new Cell'(S, X)
```

could raise Storage_Error. (See Section 25.4 for a discussion on how Pop could relinquish the storage that is no longer accessible.)

This formulation of stacks is one in which we have made the type limited private. Assignment would, of course, copy only the pointer to the stack rather than the stack itself and would have resulted in a complete mess.

Observe that assignment and predefined equality are available for the type Stack in the private part and body of the package. But we have to provide an appropriate definition of equality for use outside. This needs some care to avoid an infinite recursion as discussed in Section 11.7. But the following works

```
function "=" (S, T: Stack) return Boolean is
   SS: access Cell := S;
   TT: access Cell := T;
begin
   while SS /= null and TT /= null loop
```

```
        if SS.Value /= TT.Value then
            return False;
        end if;
        SS := SS.Next;
        TT := TT.Next;
    end loop;
    return SS = TT;          -- True if both null
end "=";
```

The key point is that we have distinguished between the type Stack itself and the type of SS so that the comparison of SS against null does not recursively invoke the function being declared.

We could of course have used an incomplete declaration in defining the type Cell. This would have avoided the type conversion in Pop. We could avoid any problems in the function "=" by introducing a derived type so that we can distinguish between the type Stack and its representation. We would then have

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

type Cell is
    record
        Next: Cell_Ptr;
        Value: Integer;
    end record;

type Stack is new Cell_Ptr;
```

The type Stack now has all the operations of the type Cell_Ptr but we can replace the inherited equality by our own definition without difficulty.

In Section 11.2 we stated that if we had to write an incomplete declaration first because of circularity (as in this type Cell) then there was an exception to the rule that the complete declaration had to occur in the same list of declarations. The exception is that it can be deferred from a private part to a body.

Thus, the complete declaration of the type Cell could be moved from the private part to the body of the package Stacks. This might be an advantage since (as we shall see in Section 13.1), it then follows from the dependency rules that a user program would not need recompiling just because the details of the type Cell are changed. In implementation terms this is possible because it is assumed that values of all access types occupy the same space – typically a single word. However, if it is deferred then the type cannot be used as an access parameter of a primitive operation or as a parameter or result of an access to subprogram type.

We can write subprograms with limited private types as parameters outside the defining package. As a simple example, the following procedure enables us to determine the top value on the stack without permanently removing it.

```
procedure Top_Of(S: in out Stack; X: out Integer) is
begin
    Pop(S, X);
    Push(S, X);
end Top_Of;
```

It is worth emphasizing that a private type such as Stack presents two views known as the partial view and the full view. Remember that within the private part (after the full type declaration) and within the body, any private type (limited or not) is treated in terms of how it is represented using the full view. On the other hand, outside the defining package and in the visible part we can only see the partial view as defined by the private declaration and the visible operations. The important principle that the properties of a type available at a given point depend entirely upon the type declaration visible from that point and not what is otherwise visible from that point is discussed further in Section 14.6 when we consider private extensions.

The parameter mechanism for a private type is that corresponding to how the type is represented. This applies both inside and outside the package. Of course, outside the package, we know nothing of how the type is represented and therefore should make no assumption about the mechanism used.

In the case of the limited type Stack, it is actually implemented as an access type and these are always passed by copy. However if the type (or in the case of a private type, the full type) is explicitly limited then parameters are always passed by reference.

Note also that the primitive operations of the type are those declared in the package specification including the private part. Thus some may be private and only known to the full view. We will return to the topic of primitive operation and views of a type when we consider tagged types in Chapter 14. Some rules for tagged types and derivation are somewhat different (such as conversion) and it is important to remember that in this chapter we have only been discussing the properties of untagged types.

The key property of limited types is that copying is not permitted. This prevents assignment statements but does not prevent the giving of an initial value providing that this does not involve copying. There are two ways in which such an initial value can be created, one is by using an aggregate and the other is by a constructor function.

Suppose for example that we wish to have a constructor function that makes up a stack with one integer item already in place. Consider

```
function Make_One(X: Integer) return Stack is
begin
   return new Cell'(null, X);
end Make_One;
```

This function (which has to be declared in the package Stacks because it uses the full view of the type) could then be used in a declaration outside the package thus

```
S: Stack := Make_One(7);                    -- initialization legal
```

with the result that the initial value of S is a stack containing the single item 7. However we could not write

```
S := Make_One(8);                           -- assignment illegal
```

outside the package because this is an assignment statement and copying is not allowed.

The mechanisms involved are easier to understand if we use a somewhat artificial type which is explicitly limited and not private. Thus consider the type

```
type T is limited
   record
      A: Integer;
      B: Boolean;
   end record;
```

The components of the record are not limited but the record as a whole is limited. We can initialize an object of the type T using an aggregate

```
V: T := (A => 10, B => True);
```

We should think of the individual components of the variable V as being initialized individually *in situ* – an actual aggregated value is not created and then assigned. Limited aggregates enable a constant of a limited type to be declared. They can also be used in a number of other contexts as well, such as

- a component of an enclosing record or array aggregate,
- the default expression for a record component,
- the expression in an initialized allocator,
- an actual parameter or default parameter expression of mode **in**,
- the result in a return statement,

as well as for generic parameters of limited types (see Chapter 19).

Limited constructor functions can be used in exactly the same places as limited aggregates. Moreover, if the type is private as well as limited then aggregates cannot be used but constructor functions can. A trivial function would be

```
function Init_T(X: Integer; Y: Boolean) return T is
begin
   return (A => X, B => Y);
end Init_T;
```

and we could then write

```
V: T := Init_T(10, True);
```

which has exactly the same effect as the declaration of V directly using an aggregate.

Moreover, the function builds the aggregate in the return statement directly in the variable V so that no copying occurs. So the address of V has to be passed as a hidden parameter to the function but this is not visible to the user. Also, the only allowed forms of expression in the return statement are aggregates and calls of other functions.

The function is more flexible than the aggregate since it can do arbitrary calculations with its parameters before constructing the return aggregate. Even more flexibility can be obtained by using an extended return statement as mentioned in Section 10.1. We can write

```
function Init_T( ... ) return T is
begin
   return X: T := ... do
      ...                          -- do arbitrary computation on X
   end return;
end Init_T;
```

This is particularly useful when the type is indefinite such as a discriminated record containing an array and we do not know the bounds of the array statically so that we cannot write an aggregate. Other applications occur with tasks and protected records; see Section 22.2.

We have seen that there are occasions when it is an advantage to make a type limited since this prevents the user from making copies. This is typically a sensible thing to do when the implementation might involve access types or the object represents a resource as illustrated in the next section. Some types are inherently limited as we shall see when we discuss task and protected types in Section 20.7.

A composite type containing a limited component is itself limited. A simple example of a limited composite type is given by an array of a limited type and we might reconsider the type Stack_Array of the previous section. In the case where the component is limited the array is also limited and so does not have predefined equality; but it could be defined and indeed the definition given as the answer to Exercise 12.4(**4**) would be satisfactory.

Limited types are very important in object oriented programming since there are many situations where an object represents a real-world entity and where making a copy would be quite inappropriate. As mentioned above explicitly limited types are always passed by reference. There are therefore good reasons for always making a type explicitly limited if assignment is not appropriate. A good example of a limited type is the type Person in Section 18.5.

Exercise 12.5

1 Write functions Is_Empty and Is_Full for the type Stack using the formulation where Stack is derived from Cell_Ptr.

2 Write the function "=" for the type Stack also using the formulation where Stack is derived from Cell_Ptr.

3 Complete the package whose visible part is

```
package Queues is
   Empty: exception;
   type Queue is limited private;
   procedure Join(Q: in out Queue; X: in Item);
   procedure Remove(Q: in out Queue; X: out Item);
   function Length(Q: Queue) return Integer;
private
```

Items join a queue at one end and are removed from the other so that a normal first-come–first-served protocol is enforced. Trying to remove an item from an empty queue raises the exception Empty. Implement the queue as a singly linked list; maintain pointers to both ends of the list so that scanning of the list is avoided. The function Length returns the number of items in the queue.

12.6 Resource management

An important example of the use of a limited private type is in providing controlled resource management. Consider the simple human model where each resource has a corresponding unique key. This key is then issued to the user when the resource is allocated and then has to be shown whenever the resource is accessed. So long as there is only one key and copying and stealing are prevented we know that the system is foolproof. A mechanism for handing in keys and reissuing them is usually necessary if resources are not to be permanently locked up. Typical human examples are the use of metal keys with safe deposit boxes, credit cards and so on.

Now consider the following

```
package Key_Manager is
   type Key is limited private;
   procedure Get_Key(K: in out Key);
   procedure Return_Key(K: in out Key);
   function Valid(K: Key) return Boolean;
   procedure Action(K: in Key; ... );
   ...
private
   Max: constant := 100;                    -- number of keys
   type Key_Code is new Integer range 0 .. Max;
   subtype Key_Range is Key_Code range 1 .. Key_Code'Last;
   type Key is limited                      -- explicitly limited
      record
         Code: Key_Code := 0;
      end record;
end;

package body Key_Manager is
   Free: array (Key_Range) of Boolean := (others => True);

   function Valid(K: Key) return Boolean is
   begin
      return K.Code /= 0;
   end Valid;

   procedure Get_Key(K: in out Key) is
   begin
      if K.Code = 0 then
         for I in Free'Range loop
            if Free(I) then
               Free(I) := False;
               K.Code := I;
               return;
            end if;
         end loop;          -- all keys in use if end of loop reached
      end if;
   end Get_Key;
```

```
    procedure Return_Key(K: in out Key) is
    begin
       if K.Code /= 0 then
          Free(K.Code) := True;
          K.Code := 0;
       end if;
    end Return_Key;

    procedure Action(K: in Key; ... ) is
    begin
       if Valid(K) then
       ...
    end Action;

end Key_Manager;
```

The type Key is represented by an explicitly limited record with a single component Code of the type Key_Code. This type is derived from Integer in order to prevent confusion between codes and any other integers in the program. The component Code has a default value of 0 which represents an unused key. Values in the range 1 .. Max represent the allocation of the corresponding resource.

When we declare a variable of type Key it automatically takes an internal code value of zero. In order to use the key we must first call the procedure Get_Key; this allocates the first free key number to the variable. The key may then be used with various procedures such as Action which represents a typical request for some access to the resource guarded by the key.

Finally, the key may be relinquished by calling Return_Key. So a typical fragment of user program might be

```
declare
   use Key_Manager;
   My_Key: Key;
begin
   ...
   Get_Key(My_Key);
   ...
   Action(My_Key, ... );
   ...
   Return_Key(My_Key);
   ...
end;
```

A variable of type Key can be thought of as a container for a key. When initially declared the default value can be thought of as representing that the container is empty; the type Key is a record partly because only record components could take default initial values in early versions of Ada when this example was first written. Note how the various possible misuses of keys are overcome.

- The user is unable to make a copy of a key because the type Key is limited and assignment is thus not available. Moreover, it is explicitly limited and so always passed by reference and never by copy.

- If the user calls Get_Key with a variable already containing a valid key then no new key is allocated. It is important not to overwrite an old valid key otherwise that key would be lost.
- A call of Return_Key resets the variable to the default state so that the variable cannot be used as a key until a new one is issued by a call of Get_Key. Moreover, the user cannot make a copy and retain it because the type is limited.

The function Valid is provided so that the user can see whether a key variable contains the default value or an allocated value. It is obviously useful to call Valid after Get_Key to ensure that the key manager was able to provide a new key value; note that once all keys are issued, a call of Get_Key does nothing.

One apparent flaw is that there is no compulsion to call Return_Key before the scope containing the declaration of My_Key is left. The key would then be lost for ever and ever. This corresponds to the real life situation of losing a key (although in our model it can never be found again – it is as if it had been thrown into a black hole). We can overcome this using a controlled type as described in Section 14.7.

Exercise 12.6

1 Complete the package whose visible part is

```
package Bank is
   type Money is new Natural;
   type Key is limited private;
   procedure Open_Account(K: in out Key; M: in Money);
      -- open account with initial deposit M
   procedure Close_Account(K: in out Key; M: out Money);
      -- close account and return balance
   procedure Deposit(K: in Key; M: in Money);
      -- deposit amount M
   procedure Withdraw(K: in out Key; M in out Money);
      -- withdraw amount M; if account does not contain M
      -- then return what is there and close account
   function Statement(K: Key) return Money;
      -- returns a statement of current balance
   function Valid(K: Key) return Boolean;
      -- checks the key is valid
private
...
```

2 Assuming that your solution to the previous question allowed the bank the use of the deposited money, reformulate the private type to represent a home savings box or safe deposit box where the money is in a box kept by the user.

3 A thief writes the following

```
declare
   use Key_Manager;
   My_Key: Key;
```

```
   procedure Cheat(Copy: in out Key) is
   begin
      Return_Key(My_Key);
      Action(Copy, ... );
      ...
   end;
begin
   Get_Key(My_Key);
   Cheat(My_Key);
   ...
end;
```

He attempts to return his key and then use the copy. Why is he thwarted?

4 A vandal writes the following

```
declare
   use Key_Manager;
   My_Key: Key;
   procedure Destroy(K: out Key) is null;   -- a null procedure
begin
   Get_Key(My_Key);
   Destroy(My_Key);
   ...
end;
```

He attempts to destroy the value in his key by calling a procedure which does not update the **out** parameter; he anticipates that this will result in a junk value being assigned to the key. Why is he thwarted?

Checklist 12

Variables inside a package exist between calls of subprograms of the package.

For predefined equality to be sensible, the values should be in a canonical form.

Predefined equality for an array type uses predefined equality of its components.

A nonlimited private type can be implemented in terms of another private type provided it is also nonlimited.

A limited private type can be implemented in terms of any private type limited or not.

Explicitly limited types are always passed by reference.

Objects of limited types can be initialized *in situ* by aggregates and function calls in Ada 2005.

The concepts of limited aggregates and limited constructor functions were new in Ada 2005.

Overriding indicators and null procedures were new in Ada 2005.

Completion of an access type cannot be deferred to a body in some circumstances in Ada 2005.

New in Ada 2012

Expression functions are new and can be used as a complete function in a package specification or body.

Expression functions can also be used as a completion in all parts of a package. An important case is where the specification is given in the visible part and the completion occurs in the private part.

The use all type clause is new in Ada 2012.

All record types compose for equality in Ada 2012.

13 Overall Structure

In this chapter we discuss the hierarchical library structure and other mechanisms for separate compilation which were outlined in Chapter 4.

Many languages ignore the simple fact that programs are written in pieces, compiled separately and then joined together. Indeed large programs should be thought of as being composed out of a number of subsystems which themselves have internal structure. There are in fact two distinct requirements.

There is a requirement for decomposing a large coherent program into a number of internal subcomponents. Such a decomposition has particular advantages when a large program is being developed by a team.

There is also a requirement for the creation of a program library where subsystems are written for general use and consequently are written before the programs that use them. Within this structure it is convenient to decompose the interface presented to future clients so that they may select only those parts of a system that are required.

This chapter also contains a further discussion on scope and visibility and summarizes the curious topic of renaming.

13.1 Library units

We will start by considering which units of Ada can be compiled separately and the general idea of dependency.

The most common units of compilation are package specifications and bodies and subprograms. Such units may be compiled individually or for convenience several could be submitted to the compiler together. Thus we could compile the specification and body of a package together but, as we shall see, it may be more

convenient to compile them individually. As usual a subprogram body alone is sufficient to define the subprogram fully.

A library unit can also be a generic package or subprogram or an instantiation of one as discussed further in Section 27.3.

Compilation units are kept in a program library in some form. We will describe the behaviour in general terms but an implementation may use any model that satisfies two major requirements: a program cannot be built out of units that are not consistent, and a unit cannot be compiled until all units upon which it depends are present in the library.

There are two obvious compilation models according to the form in which the compiler needs the information regarding dependency. In one, the source model, the compiler only needs access to the source text of the other units; in the other, the object model, the compiler requires the other units to be compiled as well. (The object model was the one presumed in Ada 83.) In order to cover such variation we use phrases such as 'entered into the library'.

So, once entered into the library, a unit can be used by any subsequently compiled unit but the using unit must indicate the dependency by a with clause.

As a simple example suppose we compile the package Stack of Section 12.1. This package depends on no other unit and so it needs no with clause. We will compile both specification and body together so the text submitted will be

```
package Stack is
   ...
end Stack;

package body Stack is
   ...
end Stack;
```

As well as producing the object code corresponding to the package, the compiler has to ensure that the program library has all the information required in order to compile subsequent dependent units. In the case of the source model nothing extra will be needed, whereas for the object model some encoded form of the information in the specification will need to be inserted.

We now suppose that we write a procedure Main which will use the package Stack. The procedure Main is going to be the main subprogram in the usual sense. It will have no parameters and we can imagine that it is called by some magic outside the language itself. The Ada language does not prescribe that the main subprogram should have the identifier Main; it is merely a convention which we are adopting here because it has to be called something.

The text we submit to the compiler could be

```
with Stack;
procedure Main is
   use Stack;
   M, N: Integer;
begin
   ...
   Push(M);
   ...
```

```
    N := Pop;
    ...
end Main;
```

The with clause goes before the unit so that the dependency of the unit on other units is clear at a glance. A with clause may not be embedded in an inner scope.

On encountering a with clause the compiler retrieves from the program library the information describing the interface presented by the unit mentioned (in source or encoded form) so that it can check that the unit being compiled uses the interface correctly. Thus if the procedure Main tries to call Push with the wrong number or type of parameters then this will be detected during compilation. This thorough checking between separately compiled units is a key feature of Ada.

If a unit is dependent on several other units then they can be mentioned in the one with clause, or it might be more convenient to use distinct with clauses. Thus we could write

```
with Stack, Diurnal;
procedure Main is ...
```

or equally

```
with Stack; with Diurnal;
procedure Main is ...
```

For convenience we can place a use clause after a with clause. Thus

```
with Stack;  use Stack;
procedure Main is ...
```

and then Push and Pop are directly visible without more ado. The use and with clauses preceding a compilation unit are collectively known as the context clause of the unit. A use clause in a context clause can only refer to packages mentioned earlier in with clauses in the context clause.

A very minor point is that we can even gratuitously repeat the same with clause and use clause in a context clause.

Only direct dependencies need be given in a with clause. Thus if package P uses the facilities of package Q which in turn uses the facilities of package R, then, unless P also directly uses R, the with clause for P should mention only Q. The user of Q does not care about R and should not need to know since otherwise the hierarchy of development would be made more complicated.

Note carefully that a context clause (both with clauses and use clauses) in front of a package or subprogram declaration will also apply to the corresponding body. It can but need not be repeated. Of course, the body may have additional dependencies which will need indicating with its own context clause anyway. Dependencies which apply only to the body should not be given with the specification since otherwise the independence of the body and the specification would be reduced.

If a package specification and body are compiled separately then the body must be compiled after the specification has been entered into the library. We say that the body is dependent on the specification. However, any unit using the package is dependent only on the specification and not the body. If the body is changed in a

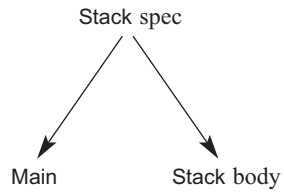


Figure 13.1 Dependencies between units.

manner consistent with not changing the specification, any unit using the package will not need recompiling. The ability to compile specification and body separately simplifies program maintenance and clearly distinguishes the logical interface from the physical implementation.

The dependencies between the specification and body of the package Stack and the procedure Main are illustrated by the graph in Figure 13.1.

The general rule regarding the order of compilation is simply that a unit must be compiled after all units on which it depends are entered into the library. Consequently, if a unit is changed and so has to be recompiled then all dependent units must also be recompiled before they can be linked into a total program.

There is one package that need not (and indeed cannot) be mentioned in a with clause. This is the package Standard which effectively contains the declarations of all the predefined types such as Integer and Boolean and their predefined operations. The package Standard is described in more detail in Section 23.1.

Finally, there are two important rules regarding library units. They must have distinct names; they cannot be overloaded. Moreover they cannot be operators. These rules simplify the implementation of an Ada program library on top of a conventional file system or database.

Exercise 13.1

1 The package D and subprograms P and Q and Main have with clauses as follows

specification of D	no with clause
body of D	**with** P, Q;
subprogram P	no with clause
subprogram Q	no with clause
subprogram Main	**with** D;

Draw a graph showing the dependencies between the units. How many different orders of compilation are possible (a) if the library uses the source model, (b) if it uses the object model?

13.2 Subunits

In this section we introduce a further form of compilation unit known as a subunit. The body of a package, subprogram (or task or protected object, see Chapter 20) can be 'taken out' of an immediately embracing library unit and itself compiled

separately. The body in the embracing unit is then replaced by a body stub. As an example suppose we remove the bodies of the subprograms Push and Pop from the package Stack. The body of Stack would then become

```
package body Stack is
   Max: constant := 100;
   S: array (1 .. Max) of Integer;
   Top: Integer range 0 .. Max;
   procedure Push(X: Integer) is separate;          -- stub
   function Pop return Integer is separate;         -- stub
begin
   Top := 0;
end Stack;
```

The removed units are termed subunits; they may then be compiled separately. They have to be preceded by **separate** followed by the name of the parent unit in parentheses. Thus the subunit Push becomes

```
separate (Stack)
procedure Push(X: Integer) is
begin
   Top := Top + 1;
   S(Top) := X;
end Push;
```

and similarly for Pop. Like distinct specifications and bodies, the specification in the subunit must have full conformance with that in the stub.

In the above example the parent unit is (the body of) a library unit. The parent body could itself be a subunit; in such a case its name must be given in full using the dotted notation starting with the ancestor library unit. Thus if R is a subunit of Q which is a subunit of P which is a library unit, then the text of R must start

```
separate (P.Q)
```

As with library units and for similar reasons, the subunits of a library unit must have distinct identifiers and not be overloaded. But we could have a subunit P.S.R as well as P.Q.R and indeed P.R. Also, subunits cannot be operators.

A subunit is dependent on its parent body (and any library units explicitly mentioned) and so must be compiled after they are entered into the library.

Visibility within a subunit is as at the corresponding body stub – it is exactly as if the subunit were plucked out with its environment intact and full type checking is maintained. As a consequence any context clause applying to the parent body need not be repeated just because the subunit is compiled separately. Moreover, it is possible to give the subunit access to additional library units by preceding it with its own context clause (possibly including use clauses). Any such context clause precedes **separate**. So the text of R might commence

```
with X;  use X;
separate (P.Q)
...
```

A possible reason for doing this might be if we can then remove any reference to library unit X from the parent P.Q and so reduce the dependencies. This would give us greater freedom with recompilation; if X were recompiled for some reason then only R would need recompiling as a consequence and not also Q.

Note that a with clause only refers to library units and never to subunits. Finally observe that several subunits or a mixture of library units, library unit bodies and subunits can be compiled together.

Exercise 13.2

1 Suppose that the package Stack is written with separate subunits Push and Pop. Draw a graph showing the dependencies between the five units: procedure Main, procedure Push, function Pop, package specification Stack, package body Stack. How many different orders of compilation are possible (a) if the library uses the source model, (b) if it uses the object model?

13.3 Child library units

One of the great strengths of Ada is the library package where the distinct specification and body decouple the user interface to a package (the specification) from its implementation (the body). This enables the details of the implementation and the clients to be recompiled separately without interference provided the specification remains stable.

However, although the simple structure we have seen so far works well for smallish programs it is not satisfactory when programs become large or complex. There are two aspects of the problem: the coarse control of visibility of private types and the inability to extend without recompilation.

There are occasions when we wish to write two distinct packages which nevertheless share a private type. We cannot do this with unrelated packages. We either have to make the type not private so that both packages can see it with the unfortunate consequence that all the client packages can also see the type; this breaks the abstraction. Or, on the other hand, if we wish to keep the abstraction, then we have to merge the two packages together and this results in a large monolithic package with increased recompilation costs.

The other aspect of the difficulty arises when we wish to extend an existing system by adding more facilities to it. If we add to a package specification then naturally we have to recompile it but moreover we also have to recompile all existing clients even if the additions have no impact upon them.

Another similar problem with a simple flat structure is the potential clash of names between library units in different parts of the system.

These problems are solved by the introduction of a hierarchical library structure containing child packages and child subprograms. There are two kinds of children: public children and private children. We will consider public children in this section and then private children in the next section.

Consider the familiar example of a package for the manipulation of complex numbers as described in Section 12.2. It contains the private type itself plus the arithmetic operations and also subprograms to construct and decompose a complex number taking a cartesian view.

```
package Complex_Numbers is
   type Complex is private;
   ...
   function "+" (X, Y: Complex) return Complex;
   ...
   function Cons(R, I: Float) return Complex;
   function Re_Part(X: Complex) return Float;
   function Im_Part(X: Complex) return Float;
private
   ...
end Complex_Numbers;
```

We have deliberately not shown the completion of the private type since it is largely immaterial how it is implemented. Although this package gives the user a cartesian view of the type, nevertheless it certainly does not have to be implemented that way as we saw in Chapter 12.

Some time later we might need to additionally provide a polar view by the provision of subprograms which construct and decompose a complex number from and to its polar coordinates. We can do this without disturbing the existing package and its clients by adding a child package as follows

```
package Complex_Numbers.Polar is

   function Cons_Polar(R, θ: Float) return Complex;
   function "abs" (X: Complex) return Float;
   function Arg(X: Complex) return Float;

end Complex_Numbers.Polar;
```

and within the body of this package we can access the full details of the private type Complex. Note the use of the Greek letter θ for the angle.

(If this example seems all too mathematical then consider the equivalent problem of representing the position of a ship in terms of either latitude and longitude or range and bearing with respect to some fixed location.)

Note the notation, a package having the name P.Q is a child package of its parent package P. We can think of the child package as being declared inside the declarative region of its parent but after the end of the specification of its parent; most of the visibility rules stem from this model. In other words the declarative region defined by the parent (which is primarily the specification and body of the parent, see Section 13.6) also includes the space occupied by the text of the children; but it is important to realize that the children are inside that region and do not just extend it. The rules are worded this way to make it clear that a child subprogram is not a primitive operation of a type declared in its parent's specification because the child is not declared in the specification but after it. (Primitive operations were introduced in Section 12.3.)

In just the same way, root library packages can be thought of as being declared in the declarative region of the package Standard and after the end of its specification. So library units are children of Standard.

An important special visibility rule is that the private part (if any) and the body of a child have visibility of the private part of the parent. In other words they have visibility of the whole of the specification of the parent and not just the visible part.

However, the visible part of a (public) child package does not have visibility of the private part of its parent; if it did it would allow renaming (discussed later in this chapter) and hence the export of the hidden private details to any client; this would break the abstraction of the private type (this rule does not apply to private children as we shall see in the next section).

The body of the child package for the complex number example can now be written using the full view of the type Complex. Assuming that it is indeed implemented as a record with components Re and Im of type Float, then access to some elementary functions will be required. So the body might be

```
with Ada.Elementary_Functions;
use Ada.Elementary_Functions;
package body Complex_Numbers.Polar is
   function Cons_Polar(R, θ: Float) return Complex is
   begin
      return (R*Cos(θ), R*Sin(θ));
   end Cons_Polar;
   ...
end Complex_Numbers.Polar;
```

In order to access the subprograms of the child package the client must, of course, have a with clause for the child. This also implicitly provides a with clause for the parent as well so we need not write both. Thus we might have

```
with Complex_Numbers.Polar;
package Client is ...
```

and then within Client we can access the various subprograms in the usual way by writing Complex_Numbers.Re_Part or Complex_Numbers.Polar.Arg and so on.

Direct visibility can be obtained by use clauses as expected. However, a use clause for the child does not imply one for the parent; but, because of the model that the child is in the declarative region of the parent, a use clause for the parent makes the child name itself directly visible. So writing

```
with Complex_Numbers.Polar;  use Complex_Numbers;
```

now allows us to refer to the subprograms as Re_Part and Polar.Arg respectively.

We could of course have added

```
use Complex_Numbers.Polar;
```

and we would then be able to refer to the subprogram in Polar just as Arg.

There is a rule that a use clause does not take effect until the end of the context clause. So we could not abbreviate this last use clause to just **use** Polar; on the grounds that we already have direct visibility of the parent.

Finally, note that any context clause on the parent also applies to its children (in the same way that it applies to its body and any subunits) and transitively to their

bodies and children and so on. Moreover, a child does not need a with clause or use clause for its parent.

Child packages neatly solve both the problem of sharing a private type over several compilation units and the problem of extending a package without having to recompile the clients. They thus provide a form of programming by extension.

A package may of course have several children. In fact with hindsight it might have been more logical to have developed our complex number package as three packages: a parent containing the private type and the four arithmetic operations and then two child packages, one giving the cartesian view and the other giving the polar view of the type. At a later date we could add yet another package providing perhaps the trigonometric functions on complex numbers and again this can be done without modifying what has already been written and thus without the risk of introducing errors.

There are a number of other examples which we have already met where child packages might be useful. We could add mixed operations between types Complex and Float in a child package (see Exercise 12.2(**1**)). We could belatedly add the functions Is_Empty and Is_Full or declare a deferred constant Empty for the package Stacks of Section 12.4 in a child package.

Declaring such a deferred constant in a child package is only possible because the private part of a child has access to the private part of its parent and so can see the full view of the type. So we might have

```
package Stacks.More is
   Empty: constant Stack:
private
   Empty: constant Stack := ((others => 0), 0);
end Stacks.More;
```

We could use this technique to declare other constants of type Complex and give them appropriate initial values.

In general the private part and body of a child package can use types declared in the private part of its parent. However, a quirk can arise in the case of access types. Recall from Section 12.5 that an accessed type in a private part can be completed in the body. In such a case the incomplete type cannot be used by a child package. This is not really surprising but is simply a consequence of the fact that a child cannot see the full type declaration.

Finally, it is very important to realize that the child mechanism is hierarchical. Children may have children to any level so we can build a complete tree providing decomposition of facilities in a natural manner. A child may have a private part and this is then visible from its children but not its parent.

Siblings have much the same relationship to each other as two unrelated packages at the top level. Thus a child can obviously only have visibility of a sibling previously entered into the library; a child can only see the visible part of its siblings; a child needs a with clause in order to access a sibling; and so on. Moreover, since siblings both have direct visibility of their parent, they can refer to each other without using the parent name as a prefix.

As already mentioned, a child (specification and body) automatically depends upon its parent (and grandparent) and needs no with clause for them. But other dependencies must be explicitly mentioned. Thus a parent body may depend upon

its children and grandchildren but their specification would have to be entered into the library first and the parent body would need with clauses for them.

One very important use of the hierarchical structure was discussed in Chapter 4 where we saw that the predefined library is structured as packages System, Interfaces and Ada each of which have numerous child packages.

We have already superficially met the package Ada.Characters.Latin_1 which is a grandchild of Ada when discussing the type Character in Section 8.4. This package contains constants for the various control characters and so we now see why we can refer to them as Ada.Characters.Latin_1.Nul and so on. Writing

```
use Ada.Characters.Latin_1;
```

allows this to be abbreviated to simply Nul.

Note the distinction between child packages and lexically nested packages such as the (obsolescent) package ASCII in Standard. Lexically nested packages cannot be separately compiled (although their bodies could be subunits) and just have single identifiers as their name in their declaration. But externally they are both referred to using the same notation and as a consequence it is illegal to have a nested package and a child package with the same identifier because it would be ambiguous. Another difference is that the private part of a nested package is never visible to a child. We shall see further similarities and differences between nested and child packages in the next section.

Although we have concentrated on child packages in this section, child subprograms are also very useful; there are some examples in the next section.

Exercise 13.3

1 Rewrite the package Complex_Numbers as a parent and two children.

2 Draw the dependency graph for the hierarchy of the previous exercise.

3 If we derive the type Field (see Section 12.3) from Complex as now structured, then which operations are inherited?

13.4 Private child units

In the previous section we introduced the concept of hierarchical child units and showed how these allowed extension and continued privacy of private types without recompilation. However, the whole idea was based around the provision of additional facilities for the client. The specifications of the additional units were all visible to the client.

In the development of large subsystems it often happens that we would like to decompose the system for implementation reasons but without giving any additional visibility to clients.

In Section 13.2 we saw how a body could be separately compiled as a subunit. However, although a subunit can be recompiled without affecting other subunits at the same level, any change to its specification requires its parent body and hence all sibling subunits to be recompiled.

Greater flexibility is provided by a form of child unit that is totally private to its parent. In order to illustrate this idea consider the following outline of an operating system.

```
package OS is
   -- parent package defines types used throughout the system
   type File_Descriptor is private;
   ...
private
   type File_Descriptor is new Integer;
end OS;

package OS.Exceptions is
   -- exceptions used throughout the system
   File_Descriptor_Error,
   File_Name_Error,
   Permission_Error: exception;
end OS.Exceptions;

private package OS.Internals is
   ...
end OS.Internals;

procedure OS.Interpret(Command: in String);

with OS.Exceptions;
package OS.File_Manager is
   type File_Mode is (Read_Only, Write_Only, Read_Write);
   function Open(File_Name: String; Mode: File_Mode)
                                                  return File_Descriptor;
   procedure Close(File: in File_Descriptor);
   ...
end OS.File_Manager;

private package OS.Internals.Debug is
   ...
end OS.Internals.Debug;
```

In this example the parent package contains the types used throughout the system. There are three public child units, the package OS.Exceptions containing various exceptions, the package OS.File_Manager which provides file open/close routines (note the explicit with clause for its sibling OS.Exceptions) and a procedure OS.Interpret which interprets a command line passed as a parameter. (Incidentally this illustrates that a child unit can be a subprogram as well as a package. It can actually be any library unit and that includes a generic declaration and a generic instantiation.) There is also a private child package called OS.Internals and a private grandchild called OS.Internals.Debug.

A private child (distinguished by starting with the word **private**) can be declared at any point in the child hierarchy. The general idea is that the private children can be used by the parent and the public children as part of their implementation but cannot be used by the clients.

Before describing the rules for private children in detail we have to introduce a new form of with clause, the so-called private with clause. As mentioned in Section 12.2, when Ada was being designed some thought was given to the idea that visible part, private part and body might be written as three distinct entities, perhaps even as

```
with A;
package P is
   ...                                  -- visible part
end;

with B;
package private P is                    -- just dreaming
   ...                                  -- private part
end;

with C;
package body P is
   ...                                  -- body
end;
```

The idea would have been that a with clause on the private part would have applied just to the private part and the body. However, it was clear that this would have been an administrative nightmare in many situations and so the two-part specification and body emerged with the private part lurking at the end of the visible part of the specification (and sharing its context clause). This was undoubtedly the right decision in general. The division into just two parts supports separate compilation well and although the private part is not part of the logical interface to the user it does provide information about the physical interface and that is needed by the compiler.

However, for some purposes this is a nuisance and a variation of the with clause can be used. If we write

```
private with Q;
package P is ...
```

then Q is visible to the private part (and body) of P but not to the visible part. In other words it behaves as a with clause on the private part which then transitively applies to the body as well. Such a private with clause can also be placed on a body in which case it is simply treated as a normal with clause.

We are not allowed to have a use clause in the same context clause as the private with clause but we can always put a use clause in the private part thus

```
private with Q;
package P is
   ...                          -- Q not visible here
private
   use Q;
   ...                          -- entities in Q directly visible here
end P;
```

We can now return to the visibility rules for private children. These are similar to those for public children but there are two extra rules.

The first extra rule is that a private child is only ever visible within the subtree of the hierarchy whose root is its parent. And moreover, within that tree, it is never visible to the visible parts of any public siblings although it can be visible to their private parts. In order for it to be visible to the private part of a public child, the specification of the public child must have a private with clause for the private sibling. A normal with clause is not permitted.

In the example as written above, the private package OS.Internals is visible to the bodies of OS, of OS.File_Manager and of OS.Interpret (OS.Exceptions has no body anyway) and it is also visible to both the specification and the body of OS.Internals_Debug. But it is not visible outside OS and a client package certainly cannot access OS.Internals at all.

However, if the private part of OS.File_Manager also needed to access the package OS.Internals then it must have a private with clause as well, thus

```
with OS.Exceptions; private with OS.Internals;
package OS.File_Manager is ...
```

The other extra rule is that the visible part of a private child can access the private part of its parent. This is quite safe because it cannot export information about a private type to a client because it is not itself visible. Nor can it export information indirectly via its public siblings because, as we have seen, it is never visible to their visible parts but only to their private parts and bodies.

We can now safely implement our system in the package OS.Internals and we can create a subtree for the convenience of development and extensibility. We might then have a third level in the hierarchy containing packages such as OS.Internals.Devices, OS.Internals.Access_Rights and so on.

The introduction of child units extends the categorization into logical, physical and implementation parts mentioned in Section 12.2. The visible part of a package plus the visible parts of its public children form the logical interface to external clients; the private parts of the parent and public children and visible and private parts of private children form the physical interface; and then the various bodies form the implementation part.

The various relationships are illustrated in Figure 13.2 which shows a package plus two public children and two private children. The solid lines with arrows indicate direct visibility which does not require with clauses; these lines can be followed transitively. Thus a private part can see the corresponding visible part and a body can see the corresponding private part and thus the visible part. Similarly, the private part of a public child can see the private part and thus the visible part of its parent but the visible part of a public child can only see the visible part of its parent.

The effect of typical with clauses is shown by broken lines. Of course a with clause only ever gives visibility of a visible part. Moreover, a line representing a with clause (normal or limited) can never go to the right; thus the specification of a public child cannot have a normal with clause for a private child, but its body can. Having followed a broken line we can then transitively follow any solid line. Thus if a client has a with clause for a child then it automatically gains access to the parent as well. One example of a private with clause is shown by a dotted line.

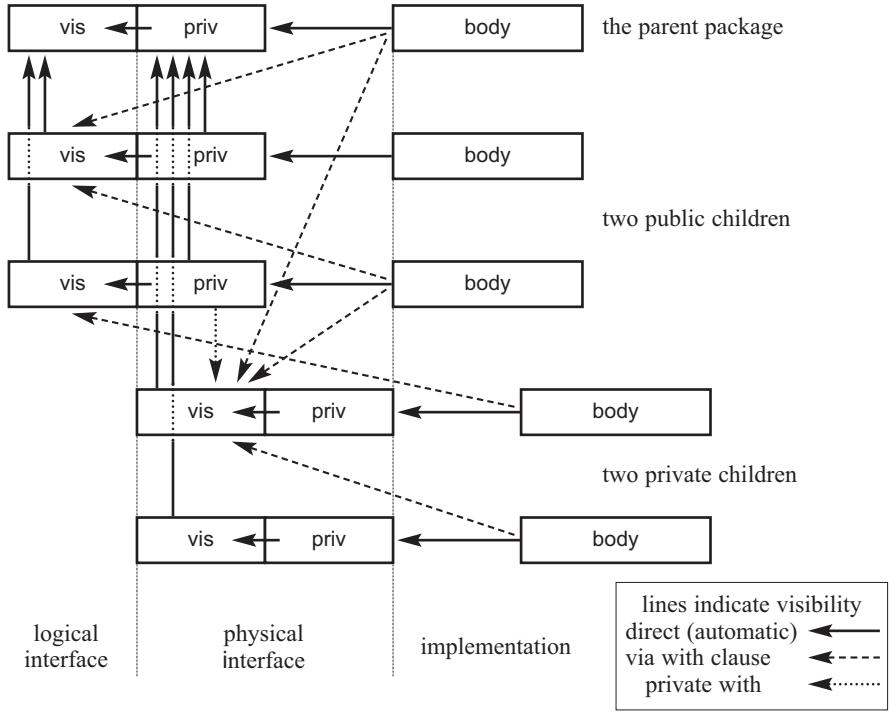


Figure 13.2 Visibility throughout a hierarchy.

Further decomposition to grandchildren is as expected. Thus a public child of a private child is effectively an extension of the specification of the private child and is visible to the bodies of its public aunts and to all parts of its private aunts. But a private child of a private child is not visible to its aunts at all.

There are a number of similarities between child and nested packages. Nested packages in a package specification behave rather like public children, whereas nested packages in a package body behave like private children. But there are differences: nested packages in a body can access earlier declarations in the body whereas a child (public or private) cannot access declarations in its parent body at all. Similar remarks apply to nested subprograms.

The rules also apply to private and public child subprograms. Thus the specification of a public child subprogram cannot have a with clause for a private sibling, but it can have a private with clause for a private sibling. A distinct body can have a with clause for a private sibling.

As a curious example, if we decided to replace the subunits of Section 13.2 by children, then the visible subprograms would become public children and a private child could be used for the declarations of the data. The structure might be as follows

```
package Stack is
end;

procedure Stack.Push(X: Integer);

function Stack.Pop return Integer;

private package Stack.Data is
   Max: constant := 100;
   S: array (1 .. Max) of Integer;
   Top: Integer range 0 .. Max := 0;
end Stack.Data;

with Stack.Data;  use Stack.Data;
procedure Stack.Push(X: Integer) is
begin
   Top := Top + 1;
   S(Top) := X;
end Stack.Push;

...   -- Pop similarly
```

The packages Stack and Stack.Data have no bodies; note in particular that the initialization of Top is done in its declaration; it could be done in a body but this requires a pragma as discussed in Section 13.8. The package Stack has an empty specification as well and so is a null package. It only exists in order to provide commonality of naming so that the private package can be declared.

An interesting point is that the use clause for Stack.Data in the context clause cannot be abbreviated to just **use** Data; but it could be so written in the declarative part of Stack.Push. The reason is that within the context clause the principle of linear elaboration means we do not know that we are in fact about to encounter a child unit of Stack. However, once inside Stack.Push we have direct visibility of Stack. Another point is that if Stack.Push did not have a distinct specification then the with clause for Stack.Data would have to be a private with clause. This structure occurs in Program 3.

Other arrangements are possible. For example the data could be in the private part of the package Stack and the subprograms would no longer need with clauses.

Although we have introduced private with clauses largely in the context of private children, nevertheless they have other uses. If we write

```
private with P;
package Q is ...
```

then we are assured that the package Q cannot inadvertently access P in the visible part. Thus writing **private with** provides additional documentation information which can be useful to both human reviewers and program analysis tools. So if we have a situation where a private with clause is all that is needed then we should use it rather than a normal with clause.

We conclude this section by summarizing the various visibility rules.

- A specification never needs a with clause for its parent; it may have one for a sibling except that a public child specification may only have a private with clause for a private sibling; it may not have a with clause for its own child. A

public child subprogram without a separate specification cannot have a with clause but can have a private with clause for a private sibling.

- A body never needs a with clause for its parent; it may have one for a sibling (public or private); it may have one for its own child.
- A context clause on a specification applies to the body and any children (and transitively to subunits and grandchildren ...).
- The entities of the parent are directly visible within a child; a use clause is not required.
- A private child is never visible outside the tree rooted at its parent. And within that tree it is not visible to the visible parts of public siblings.
- The private part and body of any child can access the private part of its parent (and grandparent ...).
- In addition the visible part of a private child can also access the private part of its parent (and grandparent ...).
- A with clause for a child automatically implies with clauses for all its ancestors.
- A use clause for a unit makes the child units accessible by simple name (this only applies to child units for which there is also a with clause).

These rules may seem a bit complex but actually stem from just a few considerations of consistency. Questions regarding access to children of sibling units and other remote relatives follow by analogy with an external client viewing the appropriate subtree.

Some consequences of the rules for child units are worth noting. We can use the same identifier for an entity in both parent and child. So given an entity X in a package P we could declare another entity X in a child P.Q thus

```
package P is
   X: Integer;
end P;

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

On the other hand we cannot use the same name for an entity in P and for the child itself. Thus the child could not be called P.X.

Exercise 13.4

1 Rewrite the body of the package Rational_Numbers of Exercise 12.2(**3**) so that the functions Normal and GCD are in a private child package Rational_Numbers.Slave.

2 Declare (in outline) an additional child package Complex_Numbers.Trig for computing the trigonometric functions Sin, Cos, etc. of complex numbers. Use a private child function Sin_Cos for the common part of the calculation. Ensure that with and use clauses are correct.

13.5 Mutually dependent units

The library structures described so far are not entirely adequate when two types depend upon each other in some way. As a simple illustration suppose we have two types Line and Point describing some geometrical configuration and that each is defined in terms of the other. For simplicity we assume that each line goes through three points and that each point is on three lines. We can do this with incomplete types thus

```
type Point;
type Line;                                   -- incomplete types

type Point is
   record
      L, M, N: access Line;
   end record;

type Line is
   record
      P, Q, R: access Point;
   end record;
```

For simple examples this is satisfactory but for more elaborate situations we might wish to declare the types and their associated operations in distinct packages. Suppose the types are declared in two child packages Geometry.Points and Geometry.Lines. We would like to write

```
with Geometry.Points;
package Geometry.Lines is
   type Line is
      record
         P, Q, R: access Points.Point;
      end record;
   ...
end Geometry.Lines;
```

with a similar declaration for the package Geometry.Points. But this is not possible because each package would have a with clause for the other and we can only have a with clause for a package that is already in the library. Clearly both packages cannot come first.

The solution is to use a limited with clause for one of them thus

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

and this package can be placed in the library first.

By writing **limited with** Geometry.Lines; we get access to all the types visible in the specification of Geometry.Lines but as if they were declared as incomplete. In other words we get an incomplete view of the types. We can then do all the things we can normally do with incomplete types such as use them to declare access types. (Of course the implementation checks later that Geometry.Lines does actually have a type Line.) We also get visibility of the names of internal packages.

Although it is only necessary to use a limited with clause on one package in order to break the circularity, it is likely that for symmetry we would place limited with clauses on the specifications of both packages. Assuming that both bodies follow both specifications then they can have normal with clauses as usual.

Incomplete types can be completed by any type other than another incomplete type in Ada 2012. They can be completed by a private type so we can write

```
type T1;                                    -- incomplete type
type T2(X: access T1) is private;
type T1(X: access T2) is private;           -- completion
```

This example uses access discriminants which are explained in Section 18.7.

Here is another example which is perhaps more realistic. This concerns employees and the departments of the organization in which they work. The information about employees needs to refer to the departments and the departments need to refer to the employees. We assume that the material regarding employees and departments is quite large so that we naturally wish to declare the two types in distinct packages Employees and Departments. So we would like to say

```
with Departments;  use Departments;
package Employees is
   type Employee is private;
   procedure Assign_Employee(E: in out Employee;
                             D: in out Department);
   type Dept_Ptr is access all Department;
   function Current_Department(E: Employee) return Dept_Ptr;
   ...
end Employees;
```

```
with Employees;  use Employees;
package Departments is
  type Department is private;
  procedure Choose_Manager(D: in out Department;
                           M: in out Employee);

  ...
end Departments;
```

We cannot write this because each package has a with clause for the other and they cannot both be declared (or entered into the library) first.

We assume of course that the type Employee includes information about the Department for whom the Employee works and the type Department contains information regarding the manager of the department and presumably a list of the other employees as well – note that the manager is naturally also an Employee.

Without limited with clauses we have to put everything into one package thus

```
package Workplace is
   type Employee is private;
   type Department is private;
   procedure Assign_Employee(E: in out Employee;
                             D: in out Department);
   type Dept_Ptr is access all Department;
   function Current_Department(E: Employee) return Dept_Ptr;
   procedure Choose_Manager(D: in out Department;
                            M: in out Employee);
private
   ...
end Workplace;
```

Not only does this give rise to huge cumbersome packages but it also prevents us from using the proper abstractions. Thus the types Employee and Department have to be declared in the same private part and so are not protected from each other's operations.

But using limited with clauses we can write

```
limited with Departments;
package Employees is
   type Employee is private;
   procedure Assign_Employee(E: in out Employee;
                             D: in out Departments.Department);
   type Dept_Ptr is access all Departments.Department;
   function Current_Department(E: Employee) return Dept_Ptr;
   ...
end Employees;
```

```
limited with Employees;
package Departments is
   type Department is private;
   procedure Choose_Manager(D: in out Department;
                            M: in out Employees.Employee);
   ...
end Departments;
```

It is important to understand that a limited with clause does not impose a dependency. Thus if a package A has a limited with clause for B, then A does not depend on B as it would with a normal with clause, and so B does not have to be compiled before A or placed into the library before A.

Note the terminology: we say that we have a limited view of a package if the view is provided through a limited with clause. So a limited view of a package provides an incomplete view of its visible types. And by an incomplete view we mean as if they were incomplete types.

Although we do not generally know whether an incomplete type is implemented by copy or by reference (unless it is tagged, see Section 14.2), nevertheless we can use an incomplete type as parameter or result provided that we do know by a point

where the subprogram is actually called or the body itself is encountered since it is only at those points that the compiler has to grind out the code for the call or the body. Earlier versions of Ada required the mechanism to be known when the specification was reached and so in the initial design of Ada 2005, the parameter D of Assign_Employee had to be changed to an access parameter.

Interesting examples of mutual dependency can occur within a hierarchy. Suppose we have a parent package App, a public child App.User_View and a private child App.Secret_Details.

In the previous section we saw that a public child could have a private with clause for a private sibling. A public and private child might also have a mutual dependency and if the public child is entered into the library first then it has to have a form of with clause for the private child that combines the restrictions of a limited with clause and a private with clause. This is called a limited private with clause and is written as follows.

```
limited private with App.Secret_Details;
package App.User_View is ...
```

A parent package can never have a normal with clause for a child since the parent must be placed in the library first. But the parent can have a limited with clause for a public child and a limited private with clause for a private child. The overall structure might be

```
limited with App.User_View;
limited private with App.Secret_Details;
package App is
   ...                          -- limited view of User_View
private
   ...                          -- limited view of Secret_Details
end App;

limited private with App.Secret_Details;
package App.User_View is
   ...                          -- full view of App, no view of Secret_Details
private
   ...                          -- limited view of Secret_Details
end App.User_View;

with App.User_View;
private package App.Secret_Details is
    ...                         -- full view of App and User_View
end App.Secret_Details;
```

There are various restrictions on the use of limited with and use clauses. The most important are that a limited with clause can only appear on a package specification (and not on a body for example) and that a use clause cannot apply to a package for which we only have a limited view.

The second restriction is perhaps surprising and means that we cannot write

```
limited with P;  use P;      -- illegal
```

The reason is to avoid various difficulties that can arise in obscure circumstances – these not concern the general user.

The fact that a limited with clause cannot appear on a body is no hardship because, as mentioned earlier, we would expect that all the specifications would come first and so we would be able to place a normal with clause on a body.

Remember from the previous section that a with clause on a parent also applies to all its children. The same applies to a limited with clause. Moreover, we can have a limited with clause on a parent unit and a corresponding normal with clause on a child. Thus the package App has a limited with clause for User_View and the child App.Secret_Details has a normal with clause for User_View. But the reverse is not permitted. Note that a child package cannot have a limited with clause for its parent (but it can have a normal with clause for its parent although it never needs one).

We cannot have both a limited with and normal with clause for the same unit in the same context clause because they imply very different views of the unit.

```
with P; limited with P;     -- illegal
```

But we can have both a private with and normal with clause in the same context clause thus

```
with P; private with P;     -- legal
```

the private with is just ignored.

13.6 Scope, visibility and accessibility

Having just introduced the concepts of library and child units it seems appropriate to summarize the major points regarding scope and visibility which will be relevant to the everyday use of Ada. For some of the fine detail, the reader is referred to the *ARM*. We have already mentioned the term declarative region. Blocks and subprograms are examples of declarative regions and the scope rules associated with them were described in Sections 6.2 and 10.6. We now have to consider the broad effect of the introduction of packages and library units.

A package specification and body together constitute a single declarative region. Thus if we declare a variable X in the specification then we cannot redeclare X in the body (except of course in an inner region such as a local subprogram).

In the case of a declaration in the visible part of a package, its scope extends from the declaration to the end of the scope of the package itself. If the package is inside the visible part of another package then this means that, applying the rule again, the scope extends to the end of that of the outer package and so on.

In the case of a declaration in a package body or in the private part of a package, its scope extends to the end of the package body.

If a unit is a library unit then its scope includes just those units which depend upon it; these are its children, body and subunits as appropriate plus those mentioning the unit in a with clause.

The rules for child units follow largely from the model of the child being in the declarative region of the parent and just after the specification of the parent. So the scope of a declaration in the specification of the parent extends through all child units. There is one important variation in the case of a declaration in the private part of a library unit: its scope does not include the visible part of any public children.

We recall from Section 10.6 that a declaration is directly visible if we can refer to it by just its direct name such as X. If not directly visible then it may still be visible and can be referred to by the dotted notation such as P.X.

Moreover, there can be places within its scope where a declaration is not visible at all. For example, a scalar, array or record object is not visible in its own declaration anywhere and a package is not visible until the reserved word **is** of its declaration.

In the case of the simple nesting of blocks and subprograms a declaration is otherwise directly visible throughout its scope and so can be referred to by its direct name except where hidden by another declaration. Where not directly visible it can be referred to by using the dotted notation where the prefixed name is that of the unit embracing the declaration.

In the case of a declaration in a package the same rules apply inside the package. Outside the package, a declaration in the visible part is visible but not directly visible unless we write a use clause.

The declarations directly visible at a given point are those directly visible before considering any use clauses plus those made directly visible by use clauses.

The basic rule is that an identifier declared in a package is made directly visible by a use clause provided the same identifier is not also in another package with a use clause and also provided that the identifier is not already directly visible anyway. If these conditions are not met then the identifier is not made directly visible and we have to continue to use the dotted notation.

A slightly different rule applies if all the identifiers are subprograms or enumeration literals. In this case they all overload each other and all become directly visible. If the specifications have type conformance (and so would normally hide each other) then the identifiers are still directly visible although things like formal parameter names may be needed to identify a particular call.

The general purpose of these rules is to ensure that adding a use clause cannot silently change the meaning of an existing piece of text. (This is the so-called Beaujolais effect. A bottle (maybe it was a case) of Beaujolais was once offered to anyone finding an example illustrating this phenomenon. Ada 83 did have some Beaujolais effects but Ada is now thought to be free from them.)

We have only given a brief sketch of the main visibility rules here and the reader is probably confused. In practice there should be no problems since the Ada compiler will, we hope, indicate any ambiguities or other difficulties and things can always be put right by adding a qualifier or using a dotted name.

There are other rules regarding record component names, subprogram parameters and so on which are as expected. For example, there is no conflict between an identifier of a record component and another use of the identifier outside the type definition itself. Consider

```
declare
   type R is
      record
         I: Integer;
      end record;

   type S is
      record
```

```
        I: Integer;
      end record;
    AR: R;
    AS: S;
    I: Integer;
begin
    ...
    I := AR.I + AS.I;          -- legal
    ...
end;
```

The visibility of the I in the type R extends from its declaration until the end of the block and so it can be referred to using the dotted notation. However, its direct visibility is confined to within the declaration of R. Another example is the use of S both as the stack and as the internal array in Section 12.4.

Similar considerations prevent conflict in named aggregates and in named parameters in subprogram calls where the name before => is not directly visible.

Note that a use clause can mention several packages. However, a use clause does not take effect until the semicolon. Suppose we have nested packages

```
package P1 is
   package P2 is
      ...
   end P2;
   ...
end P1;
```

then outside P1 we could write

```
use P1;  use P2;
```

or

```
use P1, P1.P2;
```

but not

```
use P1, P2;                                -- illegal
```

We could even write **use** P1.P2; to gain direct visibility of the entities in P2 but not those in P1.

Similar rules apply in the case of child packages. So assuming

```
package P1 is
   ...
end P1;
package P1.P2 is
   ...
end P1.P2;
```

then we can again write exactly the same use clauses as in the case of the nested packages.

Remember that a use clause in a context clause can only refer to packages mentioned in a with clause earlier in the context clause (and it must be a normal with clause and not a limited with or private with clause); moreover a use clause in a context clause does not take effect until the end of the context clause. So we cannot write something like

```
with P1.P2;  use P1;  use P2;                -- illegal
```

but have to spell it out in detail as

```
with P1.P2;  use P1;  use P1.P2;
```

Note that writing **with** P1.P2; implies **with** P1; but **use** P1.P2; does not imply **use** P1; as well. This is because dependency on the child implies dependency on the parent but there is no reason why we should not choose to have direct visibility of internal declarations as we wish.

We can repeat a with or use clause as well, thus we can even write

```
with P1, P1;  use P1;  with P1;  use P1, P1;
```

The main reason for permitting this is that it might be helpful if the text of a program is automatically generated from various components.

As mentioned in Section 12.3, there is also the use type clause which makes the primitive operators of a type directly visible. This avoids making all declarations directly visible but allows the convenience of infixed notation.

Although a use clause is much like a declaration nevertheless the range of its effect is not always the same. Thus a use clause in the visible part of a package has no effect outside the package whereas, of course, the declaration of an entity such as a variable in the visible part of a package is visible outside the package. But a use clause in a block or subprogram has a similar scope to normal declarations in a block or subprogram.

Finally, there is the package Standard. This contains all the predefined entities and moreover every root library unit should be thought of as being declared as a child of Standard. This explains why an explicit with clause for Standard is not required. Another important consequence is that, provided we do not hide the name Standard by redefining it, a library unit P can always be referred to as Standard.P. Hence, in the absence of anonymous blocks, loops and overloading, every identifier in the program has a unique name commencing with Standard. We could even refer to the predefined operators in this way

```
Four: Integer := Standard."+" (2, 2);
```

It is probably good advice not to redefine Standard; indeed declaring a package Standard simply results in Standard.Standard which is very confusing.

We conclude this section by remarking that the existence of packages plays no part in the accessibility rules. Packages are static scope walls whereas accessibility concerns dynamic lifetimes; packages can therefore be ignored in considering levels of accessibility.

13.7 Renaming

Renaming enables another name to be given to an existing entity. This is often convenient for providing a local abbreviation.

There is a strong school of thought that use clauses are bad for you. Consider the case of a large program with many library units and suppose that the unit we are in has with clauses for several packages This, That and The_Other. If we have use clauses for all these packages then it is not clear from which package an arbitrary identifier such as X has been imported; it might be This.X, That.X or The_Other.X. In the absence of use clauses we have to use the full dotted notation and the origin of everything is then obvious. However, it is also commonly accepted that long meaningful identifiers should be generally used. A long meaningful package name followed by the long meaningful name of an entity in the package is often too much. However, we can introduce an abbreviation by renaming such as

```
V: Float renames Aircraft_Data.Current_Velocity;
```

and then compactly use V in local computation and yet still have the full identification available in the text of the current unit.

Subprograms can also be renamed. Thus suppose we wish to use both the function Inner and the equivalent operator "*" of Chapter 10 without declaring two distinct subprograms. We can write one of

```
function "*" (X, Y: Vector) return Float renames Inner;
```

or the reverse

```
function Inner(X, Y: Vector) return Float renames "*";
```

according to which we declare first. Overriding indicators can also be given if desired.

Renaming of operators can be used to avoid the use of prefixed notation in the absence of a use clause (remember that many consider use package clauses to be evil); the alternative of a use type clause should be considered as described in Section 12.3.

Renaming is also useful in the case of library units. Thus we might wish to have two or more overloaded subprograms and yet compile them separately. This cannot be done directly since library units must have distinct names. However, differently named library units could be renamed so that the user sees the required effect. The restriction that a library unit cannot be an operator can similarly be overcome. The same tricks can be done with subunits.

Note also that a library unit can be renamed as another library unit. Ada 83 did not have child units and the text input–output package was simply Text_IO. In Ada 95, Ada.Text_IO was renamed as Text_IO for backward compatibility.

If a subprogram is renamed, the number, types and modes of the parameters (and result if a function) must be the same. This is called mode conformance.

Rather strangely, any constraints on the parameters or result in the new subprogram are ignored; those on the original still apply. This is because calls using the renaming are simply compiled as calls of the original subprogram. Thus we might have

```
procedure P(X: in Positive);
procedure Q(Y: in Natural) renames P;
...
Q(0);                          -- raises Constraint_Error
```

The call of Q raises Constraint_Error because zero is not an allowed value of Positive. The constraint implied by Natural on the renaming is completely ignored.

But the same does not apply to null exclusion in the case of parameters of access types. There is a general philosophy that 'null exclusions never lie'. In other words if we give a null exclusion then the entity must exclude null; however, if no null exclusion is given then the entity might nevertheless exclude null. (A good example of this occurs in Section 14.4 when we discuss access parameters and tagged types.) In the case of renaming we might have

```
procedure P(X: not null access T);
procedure Q(Y: access T) renames P;              -- legal
...
Q(null);                                   -- raises Constraint_Error
```

The call of Q raises Constraint_Error because the parameter excludes null even though there is no explicit null exclusion on the renaming. On the other hand

```
procedure P(X: access T);
procedure Q(Y: not null access T) renames P;     -- illegal
```

is illegal because the null exclusion in the renaming is a lie.

Although the number of parameters must match, the presence, absence or value of default parameters do not have to match. So renaming can be used to introduce, change or delete default expressions; the default parameters associated with the new name are those shown in the renaming declaration. Hence renaming cannot be used as a trick to give an operator default values. Similarly, parameter names do not have to match but naturally the new names must be used for named parameters of calls using the new subprogram name.

We can also provide the body of a subprogram as simply a renaming of another subprogram. The text looks exactly the same as when renaming a subprogram specification. Thus in the body of the package Simple_IO of Section 2.2 we might implement Put for the type String as simply

```
procedure Put(S: in String) renames Ada.Text_IO.Put;
```

However, since Put in Float_Text_IO has extra parameters, we could not write

```
procedure Put(F: in Float) renames Ada.Float_Text_IO.Put;
```

even though the additional (formatting) parameters have defaults.

There are, moreover, two extra rules when renaming is used for subprogram bodies. All profile subtypes must statically match and the calling conventions must be the same; this is called subtype conformance and is the same as that required for access to subprogram types. The reason for requiring subtype conformance is so that a simple jump to the old subprogram can be compiled as the call to the new one.

Table 13.1 Conformance matching of profiles.

Level	*Matches*	*Used for*
type	types	hiding (see Section 10.6) overriding untagged primitive ops (12.3)
mode	+ modes	renaming as spec generic subprograms (19.2, 19.3)
subtype	+ subtypes statically + convention	renaming as body access to subprograms (11.8) overriding tagged primitive ops (14.1) requeue (20.9)
full	+ names + defaults + null exclusions	distinct specs and bodies (8.6) stubs and subunits (13.2) entry specs and bodies/accept (20.2, 20.4) repeated discriminants (18.1)

A subprogram renaming is treated as a renaming as body rather than as a renaming as specification only if a body is needed to complete an existing specification. Note that a renaming as body can occur in a private part; this is discussed further in the answer to Exercise 19.2(**4**).

Renamings as bodies may require an elaboration check and so if misused could raise Program_Error. More interestingly, it is possible to create circularities using renaming as bodies; some are illegal because of freezing rules (see Section 25.1), but some result in infinite loops.

The reader may be baffled by all the various levels of conformance we have mentioned from time to time. For convenience they are summarized in Table 13.1 which gives them in increasing order of strength.

The unification of subprograms and enumeration literals is further illustrated by the fact that an enumeration literal can be renamed as a parameterless function with the appropriate result. For example

```
function Ten return Roman_Digit renames 'X';
```

Beware that object renaming simply provides a new name for an existing object and like subprogram renaming any constraints are ignored. Thus we can have

```
Some_Day: Day;
To_Day: Weekday renames Some_Day;
```

then To_Day has the same constraints as the type Day (none) and the constraints implied by the subtype Weekday are ignored. (We cannot give an explicit constraint in an object renaming.)

Somewhat stricter rules apply to access types. In the case of anonymous access types we might write

```
Local_Ptr: access T renames Ptr;
Local_Const: access constant T renames ACT;
Local_Sub: access procedure (X: Integer) renames P;
```

In the case of access to object types, the designated subtypes must statically match and they must both be constant or not. In the case of access to subprogram types the profiles must have subtype conformance.

It will be recalled from Section 11.7 that if we copy an access parameter to a local variable then the accessibility information was lost in Ada 2005 although preserved in Ada 2012. Similarly, renaming does not lose accessibility information, so we can write

```
procedure P(Ptr: access T) is
   Local_Ptr: access T renames Ptr;
...
```

and then Local_Ptr retains the accessibility information of the parameter Ptr.

For all access to object type renamings (named or anonymous) any null exclusion given on the renaming must not lie.

Renaming can also be used to partially evaluate the name of an object. Suppose we have an array of records such as the array People in Section 8.7 and that we wish to scan the array and print out the dates of birth in numerical form. We could write

```
for S in People'Range loop
   Put(People(S).Birth.Day);  Put(" : ");
   Put(Month_Name'Pos(People(S).Birth.Month)+1);  Put(" : ");
   Put(People(S).Birth.Year);
end loop;
```

It is clearly painful to repeat People(S).Birth each time. We could declare a variable D of type Date and copy People(S).Birth into it, but this would be very wasteful if the record were at all large. We could also use an access variable to refer to it but this would mean making the variable aliased as well as introducing an access type which might otherwise not be necessary. A better technique is to use renaming thus

```
for S in People'Range loop
   declare
      D: Date renames People(S).Birth;
   begin
      Put(D.Day);  Put(" : ");
      Put(Month_Name'Pos(D.Month)+1);  Put(" : ");
      Put(D.Year);
   end;
end loop;
```

Beware that renaming does not correspond to text substitution – the identity of the object is determined when the renaming occurs. If any variable in the name subsequently changes then the identity of the object does not change. An important point is that any constraints implied by the subtype mark in the renaming declaration are totally ignored; those on the original object still apply.

Package renaming takes a very simple form and is particularly useful with long child names thus

```
package Rights renames OS.Internals.Access_Rights;
```

The abbreviated name could for example be used in a with clause and maybe as an alternative to a use clause. Sadly, the abbreviated name cannot be used as part of the name for declaring a further child. So we could not write

```
package Rights.Data is                    -- illegal
```

but have to spell it out in pitiless detail

```
package OS.Internals.Access_Rights.Data is
```

In summary, renaming can be applied to objects (variables and constants), components of composite objects (including slices of arrays), exceptions (see Chapter 15), subprograms and packages.

Although renaming does not directly apply to types an almost identical effect can be achieved by the use of a subtype

```
subtype S is T;
```

It can also be used with incomplete types as a convenient abbreviation

```
subtype Dept is Departments.Department;          -- see Section 13.5
```

but constraints and null exclusions cannot be used with incomplete types.

Note also that renaming cannot be applied to a named number. This is partly because renaming requires a type name and named numbers do not have an explicit type name (we will see in Chapter 17 that they are of so-called universal types which cannot be explicitly named). Thus the constant e (the base of natural logarithms) in the package Ada.Numerics cannot be given a local renaming for brevity (not that it is exactly long anyway!). However, there is no need, we can just declare another named number

```
E: constant := Ada.Numerics.e;
```

which is no disadvantage because the named numbers are not run-time objects anyway and so there is no duplication.

Finally note that renaming does not automatically hide the old name nor does it ever introduce a new entity; it just provides another way of referring to an existing entity or, in other words, another view of it (that is why new constraints in a renaming declaration are ignored). Renaming can be very useful at times but the indiscriminate use of renaming should be avoided since the aliases introduced can make understanding the program more difficult.

Exercise 13.7

1 Declare a renaming of the literal Mon of type Day from the package Diurnal of Section 12.1.

2 Declare a renaming of Diurnal.Next_Work_Day.

3 Declare Pets as a renaming of part of the second Farmyard of Section 8.5.

4 Rename the operator "+" from the package Complex_Numbers of Section 12.2 so that it can be used in infix notation without a use or use type clause.

5 Declare a renaming to avoid the repeated evaluation of World(I, J) in the answer to Exercise 11.4(**2**).

13.8 Programs, partitions and elaboration

We mentioned in Section 13.1 the idea that a program starts execution by some external magic calling the main subprogram. In fact, in the general case, a complete program comprises a number of partitions each of which can have its own main subprogram. Partitions typically have their own address space and communication between partitions is restricted although the typing rules are enforced across the whole program. The details of this topic are outside the core language and are covered by the Distributed Systems annex discussed very briefly in Section 26.3. We will restrict ourselves here to programs consisting of just one partition.

The first thing that happens when a program starts is that the library units have to be elaborated. This has to be done before the main subprogram is called because it might depend upon the library units. (Incidentally, there need not be a main subprogram at all, see Section 27.3.)

Elaboration of some library units may be trivial, but in the case of a package the initial values of top level objects must be evaluated and if the body has an initialization part then that must be executed.

The order of these elaborations is not precisely specified but it must be consistent with the dependencies between the units. In addition, the pragma Elaborate (or Elaborate_All) can be used to specify that a body is to be elaborated before the current unit. This may be necessary to prevent ambiguities or even to prevent Program_Error. Consider the situation mentioned at the end of Section 12.1 thus

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

package body P is
   function A return Integer is
   begin
      return ... ;
   end A;
end P;

with P;
package Q is
   I: Integer := P.A;
end Q;
```

The three units can be compiled separately and the dependency requirements are that the body of P and the specification of Q must both be compiled after the specification of P is entered into the library. But there is no need for the body of P to be compiled before the specification of Q. However, when we come to elaborate the three units it is important that the body of P be elaborated before the specification of Q otherwise Program_Error will be raised by the attempt to call the function A in the declaration of the integer I.

The key point is that the dependency rules only partially constrain the order in which the units are elaborated and so we need some way to impose order at the library level much as the linear text imposes elaboration order within a unit. This can be done by various pragmas. Thus writing

```
with P;
pragma Elaborate(P);
package Q is
   I: Integer := P.A;
end Q;
```

specifies that the body of P is to be elaborated before the specification of Q. The pragma goes in the context clause and can refer to one or more of the library units mentioned earlier in the context clause. The pragma can be used even if the with clause is a private with clause but not a limited with clause.

The related pragma Elaborate_All specifies that all library units needed by the named units are elaborated before the current unit (and is thus transitive).

The pragma Elaborate_Body specifies that a body is to be elaborated immediately after the corresponding specification. We mentioned in Section 12.1 that a library package can only have a body if required by some rule. If we want the package Stack.Data of Section 13.4 to have a body in order to initialize Top then we can use the pragma Elaborate_Body to force a body to be required, thus

```
private package Stack.Data is
   …
   pragma Elaborate_Body(Stack.Data);
end;
```

Note that Elaborate_Body is placed in or after the corresponding specification; the parameter can be omitted.

If the dependencies and any elaboration pragmas are such that no consistent order of elaboration exists then the program is illegal; if there are several possible orders and the behaviour of the program depends on the particular order then it will not be portable.

There are three other pragmas relating to elaboration. The first is Preelaborate and appears inside the unit concerned. It essentially states that the unit can be elaborated before the program executes; generally this means that it is all static and has no code. It doesn't mean that it necessarily will be elaborated before the program runs but simply that it could be if the implementation were up to it; but certainly all such preelaborable units will be elaborated before other units which are not marked as preelaborable. This concept is important for certain real-time and distributed systems.

The second pragma is Pure and this indicates that the unit is not only preelaborable but also has no state. As noted in Chapter 4, this applies to the package Ada

```
package Ada is
   pragma Pure(Ada);
end Ada;
```

Pure units can only depend upon other pure units; preelaborable units can only depend upon other preelaborable units including pure units.

Remember that properties such as Pure are known as aspects. Thus the package Ada has the Pure aspect. Many aspects can be given by using a pragma or by an aspect specification as outlined in Section 5.6. Thus we could write

```
package Ada
   with Pure is
end Ada;
```

although that seems a bit curious perhaps because of the consequent juxtaposition of **is** and **end**. Thus properties such as Pure and Preelaborate are preferred to be set by a pragma.

The third pragma is Preelaborable_Initialization. It is used in situations like the following

```
package Q is
   pragma Preelaborate(Q);
   type T is private;
   pragma Preelaborable_Initialization(T);
private
   type T is
      record
         C: Integer := 7;
      end record;
end Q;

with Q;
package P is
   pragma Preelaborate(P);
   Obj: Q.T;
end P;
```

The package Q is preelaborable because it does not declare any objects. But the package P does declare an object and is only permitted to be preelaborable because the object has static initialization. But the object is of a private type and so this fact would not normally be visible to P. However, the package Q includes a pragma Preelaborable_Initialization for the type T and this promises that T will indeed be preelaborable. Without this pragma Preelaborable_Initialization in Q, the package P would not be preelaborable and the pragma Preelaborate in P would be illegal.

Many examples of the use of these three pragmas, Pure, Preelaborate and Preelaborable_Initialization, will be found in the predefined library discussed in Chapter 23.

A somewhat related pragma is Restrictions. This states that certain aspects of the language are not used. Most restrictions concern the Real-Time Systems or High Integrity annexes but a few relate to the core language.

The restriction No_Dependence asserts that the partition does not use a particular library unit. Thus we might write

```
pragma Restrictions(No_Dependence => Ada.Text_IO);
```

The parameter need not be a predefined unit such as Ada.Text_IO but could be any unit. Care is needed to spell the name correctly. Thus writing No_Dependence => Supperstring will not guard against using the package Superstring.

We can also write

```
pragma Restrictions(No_Implementation_Pragmas,
                    No_Implementation_Attributes);
```

These do not apply to the whole partition but only to the compilation or library environment concerned. This helps us to ensure that implementation parts of a program are identified. There is also a similar restriction No_Obsolescent_Features which ensures that we do not use features that are now deemed obsolescent such as the package ASCII.

A full list of all the predefined restrictions will be found in Appendix 1.

Exercise 13.8

1 The library package P has a Boolean variable B in its visible part. Library packages Q and R both have initialization parts; that in Q sets P.B to True whereas that in R sets P.B to False. Prevent any ambiguities by the use of elaborate pragmas.

Checklist 13

A library unit cannot be compiled until other library units mentioned in its with clause are entered into the library.

A body cannot be compiled until the corresponding specification is entered into the library.

A subunit cannot be compiled until its parent body is entered into the library.

A child cannot be compiled until the specification of its parent is entered into the library.

A package specification and body form a single declarative region.

A context clause on a specification also applies to the body and any children. Similarly a context clause on a body also applies to any subunits.

A library package can only have a body if it needs one to satisfy other language rules.

Do not attempt to redefine Standard.

Renaming is not text substitution.

Limited with and private with clauses were added in Ada 2005.

The pragma Preelaborable_Initialization was added in Ada 2005.

The Restrictions identifiers No_Dependence, No_Implementation_Attributes, No_Implementation_Pragmas and No_Obsolescent_Features were added in Ada 2005.

New in Ada 2012

An incomplete type can be completed by a private type in Ada 2012.

The parameter mechanism for subprogram parameters does not need to be known when the subprogram specification is declared, but only when a call is made or the body declared.

Many new restrictions identifiers are added in Ada 2012. See Appendix 1.

Program 3

Rational Reckoner

This program illustrates the use of a hierarchical library to create an abstract data type and its associated operations. It is based on the type Rational of Exercise 12.2(**3**).

The root package Rational_Numbers declares the type Rational and the usual arithmetic operations. The private child package Rational_Numbers.Slave contains the functions Normal and GCD as in Exercise 13.4(**1**). Thus Normal cancels any common factors in the numerator and denominator and returns the normalized form. It is called by various operations in the parent package. It would be possible to restructure Normal so that it was a private child function.

Appropriate input–output subprograms are declared in the public child Rational_Numbers.IO.

In order to exercise the rational subsystem a simple interpretive calculator is added as the main subprogram. This enables the user to read in rational numbers, perform the usual arithmetic operations upon them and print out the result.

The numbers are held in a stack which is operated upon in the reverse Polish form (Polish after the logician Jan Łucasiewicz). The root package Rat_Stack contains subprograms Clear, Push and Pop. The private child package Rat_Stack.Data contains the actual data, and the public child procedure Rat_Stack.Print_Top prints the top item. This structure is somewhat similar to that described in Section 13.4 and enables the stack size to be changed with minimal effect on the rest of the system.

The main subprogram is decomposed into subunits Process and Get_Rational in order to isolate the dependencies and clarify the structure.

```
package Rational_Numbers is
   type Rational is private;

   -- unary operators
   function "+" (X: Rational) return Rational;
   function "-" (X: Rational) return Rational;

   -- binary operators
   function "+" (X, Y: Rational) return Rational;
   function "-" (X, Y: Rational) return Rational;
   function "*" (X, Y: Rational) return Rational;
   function "/" (X, Y: Rational) return Rational;

   -- constructor and selector functions
   function "/" (X: Integer; Y: Positive) return Rational;
   function Numerator(R: Rational) return Integer;
   function Denominator(R: Rational) return Positive;

private
   type Rational is
      record
         Num: Integer := 0;      -- numerator
         Den: Positive := 1;     -- denominator
      end record;
end;
```

```
private package Rational_Numbers.Slave is
   function Normal(R: Rational) return Rational;
end;

package body Rational_Numbers.Slave is

   function GCD(X, Y: Natural) return Natural is
   begin
      if Y = 0 then
         return X;
      else
         return GCD(Y, X mod Y);
      end if;
   end GCD;

   function Normal(R: Rational) return Rational is
      G: Positive := GCD(abs R.Num, R.Den);
   begin
      return (R.Num/G, R.Den/G);
   end Normal;

end Rational_Numbers.Slave;
```

```
with Rational_Numbers.Slave;
package body Rational_Numbers is
   use Slave;

   function "+" (X: Rational) return Rational is
   begin
      return X;
   end "+";

   function "–" (X: Rational) return Rational is
   begin
      return (–X.Num, X.Den);
   end "–";

   function "+" (X, Y: Rational) return Rational is
   begin
      return Normal((X.Num*Y.Den + Y.Num*X.Den,
                                        X.Den*Y.Den));
   end "+";

   function "–" (X, Y: Rational) return Rational is
   begin
      return Normal((X.Num*Y.Den – Y.Num*X.Den,
                                        X.Den*Y.Den));
   end "–";

   function "*" (X, Y: Rational) return Rational is
   begin
      return Normal((X.Num*Y.Num, X.Den*Y.Den));
   end "*";

   function "/" (X, Y: Rational) return Rational is
      N: Integer := X.Num*Y.Den;
      D: Integer := X.Den*Y.Num;
   begin
      if D < 0 then D := –D;  N := –N; end if;
      return Normal((Num => N, Den => D));
   end "/";

   function "/" (X: Integer; Y: Positive)
                                    return Rational is
   begin
      return Normal((Num => X, Den => Y));
   end "/";

   function Numerator(R: Rational)
                                     return Integer is
   begin
      return R.Num;
   end Numerator;

   function Denominator(R: Rational)
                                    return Positive is
   begin
      return R.Den;
   end Denominator;

end Rational_Numbers;
```

```
package Rational_Numbers.IO is
   procedure Get(X: out Rational);
   procedure Put(X: in Rational);
end;
```

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada;
with Rational_Numbers.Slave;
package body Rational_Numbers.IO is

   procedure Get(X: out Rational) is
      N: Integer;              -- numerator
      D: Integer;              -- denominator
      C: Character;
      EOL: Boolean;            -- end of line
   begin
      -- read the (possibly) signed numerator
      -- this also skips spaces and newlines
      Integer_Text_IO.Get(N);
      Text_IO.Look_Ahead(C, EOL);
      if EOL or else C /= '/' then
         raise Text_IO.Data_Error;
      end if;
      Text_IO.Get(C);     -- remove the / character
      Text_IO.Look_Ahead(C, EOL);
      if EOL or else C not in '0' .. '9' then
         raise Text_IO.Data_Error;
      end if;
      -- read the unsigned denominator
      Integer_Text_IO.Get(D);
      if D = 0 then
         raise Text_IO.Data_Error;
      end if;
      X := Slave.Normal((N, D));
   end Get;

   procedure Put(X: in Rational) is
   begin
      Integer_Text_IO.Put(X.Num, 0);
      Text_IO.Put('/');
      Integer_Text_IO.Put(X.Den, 0);
   end Put;

end Rational_Numbers.IO;
```

```
with Rational_Numbers;
use Rational_Numbers;
package Rat_Stack is
   Error: exception;
   procedure Clear;
   procedure Push(R: in Rational);
   function Pop return Rational;
end;
```

```
private package Rat_Stack.Data is
   Max: constant := 4;          -- stack size
   Top: Integer := 0;
   Stack: array (1 .. Max) of Rational;
end Rat_Stack.Data;
```

```
with Rat_Stack.Data;
package body Rat_Stack is
   use Data;

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

   procedure Push(R: in Rational) is
   begin
      if Top = Max then
         raise Error;
      end if;
      Top := Top + 1;
      Stack(Top) := R;
   end Push;

   function Pop return Rational is
   begin
      if Top = 0 then
         raise Error;
      end if;
      Top := Top – 1;
      return Stack(Top + 1);
   end Pop;

end Rat_Stack;
```

```
with Rational_Numbers.IO;
with Ada.Text_IO;
private with Rat_Stack.Data;
procedure Rat_Stack.Print_Top is
   use Data;
begin
   if Top = 0 then
      Ada.Text_IO.Put("Nothing on stack");
   else
      Rational_Numbers.IO.Put(Stack(Top));
   end if;
   Ada.Text_IO.New_Line;
end Rat_Stack.Print_Top;
```

```
with Rat_Stack;
with Ada.Text_IO;  use Ada.Text_IO;
procedure Rational_Reckoner is
   C: Character;
   Control_Error, Done: exception;
   procedure Process(C: Character) is separate;

begin
   Put("Welcome to the Rational Reckoner");
   New_Line(2);
   Put_Line("Operations are + – * / ? ! plus eXit");
   Put_Line("Input rational by #[sign]digits/digits");
   Rat_Stack.Clear;
   loop
      begin
         Get(C);
         Process(C);
      exception
         when Rat_Stack.Error =>
            New_Line;
            Put_Line("Stack overflow/underflow, " &
                     "stack reset");
            Rat_Stack.Clear;
         when Control_Error =>
            New_Line;
            Put_Line("Unexpected character, " &
                     "not # + – * / ? ! or X");
         when Done =>
            exit;
      end;
   end loop;
   New_Line;
   Put_Line("Finished");
   Skip_Line(2);
end Rational_Reckoner;
```

```
with Rat_Stack.Print_Top;  use Rat_Stack;
with Rational_Numbers;  use Rational_Numbers;
separate(Rational_Reckoner)
procedure Process(C: Character) is
   R: Rational;
   procedure Get_Rational(R: out Rational)
                                        is separate;
begin
   case C is
      when '#' =>
         Get_Rational(R);
         Push(R);
      when '+' =>
         Push(Pop + Pop);
      when '–' =>
         R := Pop;  Push(Pop – R);
      when '*' =>
         Push(Pop * Pop);
      when '/' =>
         R := Pop;  Push(Pop / R);
      when '?' =>
         Print_Top;
      when '!' =>
         Print_Top;  R := Pop;
      when ' ' =>
         null;
      when 'X' | 'x' =>
         raise Done;
      when others =>
         raise Control_Error;
   end case;
end Process;
```

```
with Rational_Numbers.IO;
separate(Rational_Reckoner.Process)
procedure Get_Rational(R: out Rational) is
begin
   loop
      begin
         IO.Get(R);
         exit;
      exception
         when Data_Error =>
            Skip_Line;  New_Line;
            Put_Line("Not a rational, try again ");
            Put('#');
      end;
   end loop;
end Get_Rational;
```

The procedures for input and output of rational numbers are in a distinct package since it is likely that alternative formats might be tried and this avoids changing the root package.

For simplicity the procedure Get uses the predefined procedure Get for reading integers in Ada.Integer_Text_IO. The expected format for a rational number consists of an optional sign, a sequence of digits, a solidus (/), and then another sequence of digits; for example –34/67. Using the integer Get for the denominator automatically causes any leading spaces and newlines to be skipped and any sign to be processed. It then uses the procedure Look_Ahead to look at the next characters (see Section 23.6) and raises the predefined exception Ada.IO_Exceptions.Data_Error (via the renaming in Ada.Text_IO, see Sections 23.5 and 23.6) if the numerator is not immediately followed by a solidus and a digit. It then reads the (unsigned) denominator and checks that it is not zero. The behaviour with regard to skipping spaces and newlines follows the same general pattern as for the predefined subprograms Get for the predefined types. Note, however, that by using the predefined Get for integers, we do actually allow the numerator and denominator to be in based notation and to have an exponent!

By contrast, Put is almost trivial although more elaborate formats could be devised.

The functions Normal and GCD are also in a distinct package. Again this enables alternative algorithms to be used without recompiling the entire system. Note that since the denominator is never zero, the test Y = 0 in the top level call of GCD never succeeds and so at least one recursion always occurs. The parameters could be interchanged to avoid this. An alternative approach is to use the iterative form as in the answer to Exercise 10.1(7).

A further discussion on the structure of these packages will be found on the web.

The stack subsystem has been divided into several units mainly so that Print_Top is distinct and alternative formats can be tried without disturbing the whole of the stack system. Thus Print_Top has a private with clause giving access to the data in the private package Rat_Stack.Data. Another approach might have been to put the data in the private part of the root package Rat_Stack but this would mean recompiling the whole system if the stack depth (currently 4) were changed.

The main subprogram has been decomposed into subunits largely to clarify the exception handling and avoid deep indentation; it also reduces the dependencies somewhat. Exception handling is discussed in detail in Chapter 15 but it is hoped that the simple structure shown here is quite clear. The calculator is driven by the procedure Process which manipulates the system according to a control character. The character # signifies that a rational number is to follow; the characters +, –, * and / cause the appropriate operations on the top two items of the stack (care has to be taken with – and / that the items are used in the correct order); the characters ? and ! cause the top item to be printed and ! also causes the top item to be deleted. Spaces are ignored. Thus the sequence

```
#3/4 #2/3 + !
```

results in 17/12 being output. The character X or x causes the system to terminate.

If the stack is misused then Rat_Stack.Error is raised and the stack reset to empty. An unexpected control character raises Control_Error and the system continues after a suitable warning. Note also that termination is indicated by the raising of the exception Done; this might be considered bad practice since termination in this case is not an exceptional situation. On the other hand it is quite convenient to bundle it in with the other exceptions.

The reading of a rational number is carried out by the subunit Get_Rational. Spaces and newlines are allowed between the # and the number but not within the number itself. If Data_Error is raised because the sequence following # is not recognized then the user is invited to resubmit a sequence and a further # is output as a prompt. Note also how a call of Skip_Line removes any unused characters.

The system could clearly be extended in various ways. For example, a prompt might be output whenever input is expected. Another improvement would be to handle Constraint_Error which is raised if any of the numeric operations overflow.

14 Object Oriented Programming

We now come to a discussion of the basic features of Ada which support what is generally known as object oriented programming or OOP. Ingredients of OOP include the ability to extend a type with new components and operations, to identify a specific type at run time and to select a particular operation depending on the specific type. A major goal is the reuse of existing reliable software without the need for recompilation.

This chapter concentrates on the fundamental ideas of type extension and polymorphism. Related topics are type parameterization (using discriminants) and genericity; these are discussed in Chapters 18 and 19 respectively. Finally, Chapter 20 considers how these various aspects of OOP all fit together especially with regard to multiple inheritance.

The reader might find it helpful to reread Sections 3.2 and 3.3 before considering this chapter in detail.

14.1 Type extension

In Section 12.3 we saw how it was possible to declare a new type as derived from an existing type and how this enabled us to use strong typing to prevent inadvertent mixing of different uses of similar types. We also saw that primitive operations were inherited by the derived type and could be overridden and, moreover, that further primitive operations could be added if the derivation were in a package specification.

We now introduce a more flexible form of derivation where it is possible to add additional components to a record type as well as additional operations. This gives rise to a possible tree of types where each type contains the components of its parent

plus other components as well. Since the types are clearly different, although with common properties, it is convenient to be able to deal with an object of any type in the tree and to determine the type of the object at run time. It is clear therefore that each object has to have some additional information indicating its type. This additional information is provided by a hidden component called the tag.

Accordingly, record types can be extended on derivation provided they are marked as tagged. Private types implemented as records can also be marked as tagged.

We saw a simple example of a hierarchy of tagged types plus associated primitive operations in Chapter 3 where we declared the types Object, Circle and so on. These could be declared in one or several packages as illustrated in the answer to Exercise 3.2(**1**).

```
package Objects is

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

   function Distance(O: Object) return Float;
   function Area(O: Object) return Float;

end Objects;

with Objects;  use Objects;
package Shapes is

   type Circle is new Object with
      record
         Radius: Float;
      end record;

   function Area(C: Circle) return Float;

   type Point is new Object with null record;

   type Triangle is new Object with
      record
         A, B, C: Float;
      end record;

   function Area(T: Triangle) return Float;

end Shapes;
```

In this example the type Object is the root of a tree of types and Circle, Point and Triangle are derived from it. Note how the extra components are indicated and that in the case of the type Point where no extra components are added we have to indicate this explicitly by **with null record**; this makes it clear to the reader that it is a tagged type since every tagged type has **tagged** or **with** (or **interface** as discussed in Section 14.3) in its declaration.

The primitive operations of Object are Area and Distance and these are inherited by Circle, Point and Triangle. Although Distance is appropriate for all the types, the function Area is redefined for Circle and Triangle. (As we saw in Section 3.3, the function Area for Object should really be abstract so that it cannot be accidentally inherited; see Section 14.3. Moreover, Distance would be better with a class wide parameter as noted in the next section.)

Type conversion is always allowed towards the root of the tree, but an extension aggregate is required in the opposite direction in order to give values for any additional components. So we can write

```
O: Object := (1.0, 0.5);
C: Circle := (0.0, 0.0, 34.7);
T: Triangle;
P: Point;
...
O := Object(C);
...
C := (O with 41.2);
T := (O with A => 3.0, B => 4.0, C => 5.0);
P := (O with null record);
```

An extension aggregate can use positional or named notation and the familiar **with null record** is required when there are no extra components.

The expression before **with** can be an expression of any appropriate ancestor type, it does not have to be of the immediate parent type. So moving into another dimension we might have

```
type Cylinder is new Circle with
   record
      Height: Float;
   end record;

Cyl: Cylinder;
```

and then we could write any of

```
Cyl := (O with Radius => 41.2, Height => 231.6);
Cyl := (C with Height => 231.6);
Cyl := (Object(T) with 41.2, 231.6);
```

In the last case we first convert the triangle T to an object and then extend it to give a cylinder. However, it should be noted that the type Cylinder is not sensible since a cylinder is not a form of circle at all. We will return to this topic in Section 21.1.

When we come to Section 14.7 we will find that it is sometimes not possible to give an expression of an ancestor type; as an alternative we can simply provide a subtype mark instead. In fact we can always do this and then the components corresponding to the ancestor type are initialized by default (if at all) as for any object of the type. So writing

```
C := (Object with Radius => 41.2);
```

will result in the circle having components as if we had written

```
Obj: Object;                    -- no initial value
C: Circle := (Obj with Radius => 41.2);
```

and so the coordinates of C are not defined.

This is clearly not very sensible in this case but we might have chosen to declare the type Object so that it was by default at the origin thus

```
type Object is tagged
  record
    X_Coord: Float := 0.0;
    Y_Coord: Float := 0.0;
  end record;
```

and then the circle would also by default be at the origin.

At this point we pause to consider the main commonalities and differences between type derivation with tagged types which we have just been discussing and type derivation with other types as discussed in Section 12.3.

The common points are

- existing components are inherited,
- inheritance, overriding and addition of primitive operations are allowed in the same places; additional operations are only allowed if the derivation occurs in a package specification,
- derivation can occur in the same package specification as the parent and inherits all its primitive operations declared so far.

On the other hand, the differences are

- derivation from a tagged type freezes it so that no further operations can be added,
- a tagged record type can have additional components,
- type conversion is only allowed towards the ancestor for a tagged type; both ways for untagged types,
- inherited operations of tagged types have the same convention as the parent type whereas inherited operations of untagged types are always intrinsic; remember that the Access attribute cannot be applied to intrinsic operations,
- if an inherited operation is overridden then the conformance requirements are different; in the case of a tagged type it must have subtype conformance, whereas for an untagged type it only has to have type conformance.

The reason for the last difference will become apparent later.

We now consider a more extensive example which illustrates the use of tagged types to build a system as a hierarchy of types. We will see how this allows the system to be extended without recompilation of its central part.

Our system concerns the processing of reservation requests for a mythical Ada Airlines (not low cost but naturally very reliable). There are presumably various aspects to this: the creation of a request; its processing by some central system; and then reporting back indicating success or failure. There are three categories of travel, Basic, Nice and Posh. The better categories have options which can be

requested when making the reservation. Nice passengers are given a choice of seat (Aisle or Window) and a choice of meal which can be Green (vegetarian), White (fish or fowl) or Red (for the carnivores). Posh passengers are also given onward personal ground transport (or Personal Onward Surface Help).

We concentrate on the part of the system that processes the requests and consider the hierarchy of types required. The package specification might be

```
package Reservation_System is

   type Position is (Aisle, Window);
   type Meal_Type is (Green, White, Red);

   type Reservation is tagged
      record
         Flight_Number: Integer;
         Date_Of_Travel: Date;
         Seat_Number: String(1 .. 3) := "   ";
      end record;

   procedure Make(R: in out Reservation);
   procedure Select_Seat(R: in out Reservation);

   type Basic_Reservation is new Reservation with null record;

   type Nice_Reservation is new Reservation with
      record
         Seat_Sort: Position;
         Food: Meal_Type;
      end record;

   overriding
   procedure Make(NR: in out Nice_Reservation);              -- overrides
   procedure Order_Meal(NR: in Nice_Reservation);

   type Posh_Reservation is new Nice_Reservation with
      record
         Destination: Address;
      end record;

   overriding
   procedure Make(PR: in out Posh_Reservation);              -- overrides
   procedure Arrange_Limo(PR: in Posh_Reservation);

end Reservation_System;
```

We start with the root type Reservation which contains the components common to all reservations. It also has a procedure Make which performs all those actions common to making a reservation of any category. The type Basic_Reservation is simply a copy of Reservation (note **with null record**;) and could be dispensed with; Basic_Reservation inherits the procedure Make from Reservation. The type Nice_Reservation extends Reservation and provides its own procedure Make thus overriding the inherited version. The type Posh_Reservation further extends Nice_Reservation and similarly provides its own procedure Make. The procedures Select_Seat, Order_Meal and Arrange_Limo are called by the various

procedures Make as required. The main purpose of Select_Seat is to fill in the seat number with a string such as "56A"; hence the parameter has in out mode.

Note the use of the reserved word **overriding** which precedes the overriding procedures Make. This is optional as explained in Section 12.3 and is an indication to the compiler (and anyone maintaining the program later) that this truly is an overriding of an existing inherited operation. This is a safeguard and the program will fail to compile if we accidentally spell Make incorrectly or give it the wrong number of parameters. We can also write **not overriding** when declaring a new operation such as Select_Seat or the first time we declare Make.

The package body might be as follows

```
package body Reservation_System is

   procedure Make(R: in out Reservation) is
   begin
      Select_Seat(R);
   end Make;

   procedure Make(NR: in out Nice_Reservation) is
   begin
      Make(Reservation(NR));              -- make as plain reservation
      Order_Meal(NR);
   end Make;

   procedure Make(PR: in out Posh_Reservation) is
   begin
      Make(Nice_Reservation(PR));         -- make as nice reservation
      Arrange_Limo(PR);
   end Make;

   procedure Select_Seat(R: in out Reservation) is separate;
   procedure Order_Meal(NR: in Nice_Reservation) is separate;
   procedure Arrange_Limo(PR: in Posh_Reservation) is separate;

end Reservation_System;
```

Each distinct body for Make contains just the code immediately relevant to the type and delegates other processing back to its parent using an explicit type conversion. This avoids repetition of code and simplifies maintenance. Note carefully that all type checking is static; the choice of Make is done with simple overload resolution based on the known type of the parameter. (The reader may feel concerned that the procedure Select_Seat does not have enough information to work for nice and posh passengers; do not worry – all will be revealed in Section 14.5.)

If at a later date the growing Ada Airlines purchases some secondhand Concordes (alas no more) then a new reservation type such as Supersonic_Reservation will be required. This can be added without recompiling (and perhaps more importantly, without retesting) the existing code.

```
with Reservation_System;
package Supersonic_Reservation_System is
```

```
   type Supersonic_Reservation is
                                  new Reservation_System.Reservation with
      record
         Champagne: Vintage;
         ...   -- other supersonic components
      end record;
   overriding
   procedure Make(SR: in out Supersonic_Reservation);
   ...
end Supersonic_Reservation_System;
```

We could have made this package a child of Reservation_System. This would have avoided the need for a with clause and emphasized that it was all really part of the same system. The entities in the parent package would also be immediately visible and use clauses or dotted names would be avoided. Furthermore, as we shall see in Section 14.6, the fact that the child package can see the private part of its parent can be very important.

Exercise 14.1

1 Declare a point P with the same coordinates as a given circle C.

2 Declare an object R of type Reservation and assign to it an appropriate flight number and date. Then declare an object NR of the type Nice_Reservation with common components the same as R and a window seat and vegetarian meal.

3 Rewrite the package Supersonic_Reservation_System as a child package of Reservation_System.

14.2 Polymorphism

The facilities we have seen so far have allowed us to define a new type as an extension of an existing one. We have introduced the different categories of Reservation as distinct but related types. What we also need is a means to manipulate any kind of Reservation and to process it accordingly. We do this through the introduction of the notion of class wide types which provide dynamic polymorphism.

Each tagged type T has an associated type denoted by T'Class. This type comprises the union of all the types derived from T. The values of T'Class are thus the values of T and all its derived types; the type T is known as the root type of the class. Moreover, a value of any type derived from T can always be converted to the type T'Class (implicitly in certain contexts).

So, for example, in the case of the type Reservation the types can be pictured as in Figure 14.1. A value of any of the reservation types can be converted to Reservation'Class. Note carefully that Nice_Reservation'Class is not the same as Reservation'Class; the former consists just of Nice_Reservation and Posh_Reservation.

Each value of a class wide type has a tag which identifies its particular type at run time. Thus the tag acts as a hidden component as mentioned earlier.

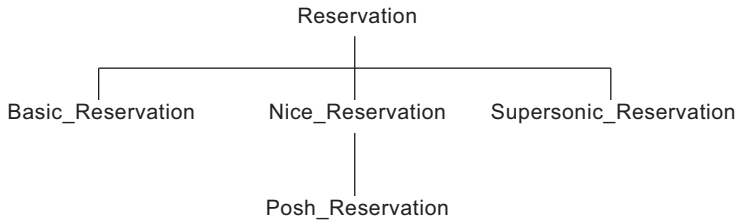


Figure 14.1 A tree of types.

The type T'Class is treated as an indefinite type (like an unconstrained array type); this is because we cannot possibly know how much space could be required by any value of a class wide type because the type might be extended. As a consequence, although we can declare an object of a class wide type, it must be constrained; however, the constraint is not given explicitly but by initializing it with a value of a specific type and it is then constrained by the tag of that type. So we could write

```
NR: Nice_Reservation;
...
RC: Reservation'Class := NR;
```

although this is not very helpful. Of more importance is the fact that a formal parameter can be of a class wide type and the actual parameter can then be of any specific type in the class. (Note again the analogy with arrays; a formal can be of an unconstrained array type and the actual can then be constrained.)

We now continue our example by considering how we might queue a series of reservation requests and process them in sequence by some central routine. The essence of the problem is that such a routine cannot assume knowledge of the individual types since we want it to work (without recompilation) even if we extend the system by adding a new reservation type to it.

The central routine could thus take a class wide value as its parameter so we might have

```
procedure Process_Reservation(RC: in out Reservation'Class) is
   ...
begin
   ...
   Make(RC);                -- dispatch according to tag
   ...
end Process_Reservation;
```

In this case we do not know which procedure Make to call until run time because we do not know which specific type the reservation belongs to. However, RC is of a class wide type and so its value includes a tag indicating the specific type of the value. The choice of Make is then determined by the value of this tag; the parameter is then implicitly converted to the appropriate specific reservation type before being passed to the appropriate procedure Make.

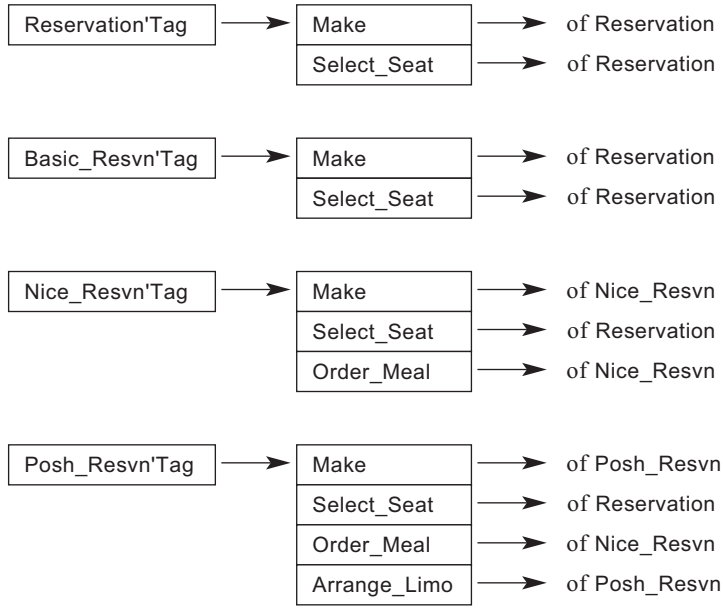


Figure 14.2 Tags and dispatch tables.

This run-time choice of procedure is called dispatching and is key to the flexibility of class wide programming. It is important to realize that dispatching can be implemented very efficiently. A possible implementation in this case is to make the tag point to a dispatch table each entry of which in turn points to the code of the body of a primitive operation. Each value of a tagged type will have the tag at a standard place such as at the beginning of the value. This model is illustrated in Figure 14.2 which shows how the various operations are inherited, replaced or added. (For simplicity, we have omitted the predefined primitive operations such as equality.)

The reason that dispatching is efficient is that there is never any need to check anything at run time. The dispatch table is arranged so that the displacements are the same for all types in the class. Also we know that every operation in the class is present because operations cannot be removed on derivation; only replaced or added. Moreover, as mentioned in the previous section, they always have the same convention and have subtype conformance so that the call always works dynamically.

(It should be added that the simple model described here needs to be extended when we come to include the effect of multiple inheritance to be discussed in Section 14.8. But it illustrates the general ideas.)

We now see the importance of the distinction between Reservation'Class and Nice_Reservation'Class. We can only dispatch to Order_Meal from the latter class wide type since only the latter has Order_Meal as a primitive operation of every type in the class.

Note that a procedure with a class wide parameter such as Process_Reservation is not a primitive operation of the root type and is never inherited and so cannot be overridden. It is often an advantage to use a class wide rather than a primitive operation if by its very nature it will apply to all types in the class. Thus the function Distance applying to objects is better written as

```
function Distance(O: Object'Class) return Float;
```

since unlike Area it applies unchanged to all types derived from Object.

We continue by considering how the various reservation requests might be held on a heterogeneous list awaiting processing. We can declare an access type referring to a class wide type. So we can write

```
type Reservation_Ptr is access all Reservation'Class;
```

in which case an access variable of this type could designate any value of the class wide type. We cannot change the specific type of the object referred to at any time into another type but we can from time to time refer to objects of different specific types. (This is much as the access variable R of Section 11.3 can refer to matrices of different sizes since the type Matrix is unconstrained; we cannot change the size of a particular matrix but R can refer to different sized matrices at different times.) The flexibility of access types is a key factor in class wide programming.

A heterogeneous list can be made in the obvious way using

```
type Cell is
   record
      Next: access Cell;
      Element: Reservation_Ptr;
   end record;
```

and the central routine can then manipulate the reservations using an access value as parameter

```
procedure Process_Reservation(RP: in Reservation_Ptr) is
   ...
begin
   ...
   Make(RP.all);              -- dispatch to appropriate Make
   ...
end Process_Reservation;
...
List: access Cell;            -- list of reservations
...
while List /= null loop       -- process the list
   Process_Reservation(List.Element);
   List := List.Next;
end loop;
```

In this case, the value of the object referred to by RP is of a class wide type and so includes a tag indicating the specific type. The parameter RP.all is thus dereferenced, the value of the tag gives the choice of Make and the parameter is then implicitly converted before being passed to the chosen procedure Make as before.

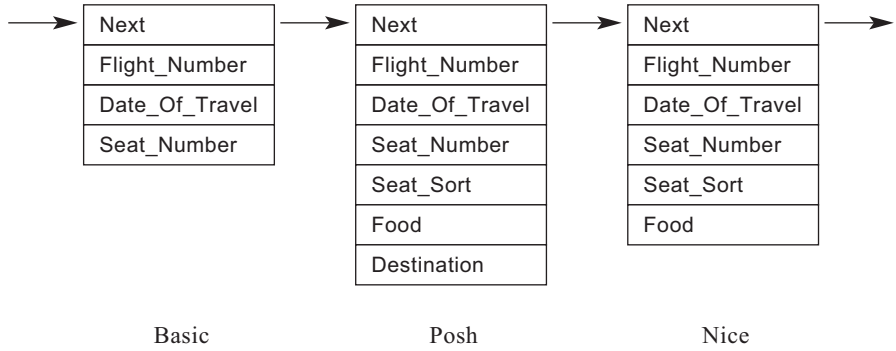


Figure 14.3 A heterogeneous list.

It is fundamental in class wide programming to manipulate objects via references; this is largely because the objects may be of different sizes. As a consequence, references to the objects will typically be on a list of some kind and will move from list to list as they are processed within the system. (The objects themselves will not move and so, as mentioned in Section 12.5, can be of a limited type.) If, as is likely, each object is only on one list at a time, then we can adopt a rather neater approach to chaining the objects together.

The general idea is that we first declare some root type which contains the pointer to the next item in the list and then the types to be placed on the list can be extended from the root type. Consider

```
type Element is tagged;                          -- tagged incomplete
type Element_Ptr is access all Element'Class;

type Element is tagged
   record
      Next: Element_Ptr;
   end record;
```

We have used a form of incomplete type which states that the full type will be tagged and so permits the use of the Class attribute. Objects of any type in the class Element'Class can be linked together through the one common element. We can now modify the reservation system so that the type Reservation is

```
type Reservation is new Element with
   record
      Flight_Number: Integer;
      Date_Of_Travel: Date;
      Seat_Number: String(1 .. 3) := "   ";
   end record;
```

with the rest of the system as before. The various reservations can now be joined together to form a list as illustrated in Figure 14.3.

The manipulation of such lists or queues is very common in object oriented programming and it is convenient to have a package providing appropriate standard operations. So, somewhat like Exercise 12.5(**3**), we might have

```
package Queues is
   Queue_Error: exception;
   type Queue is limited private;
   type Element is tagged private;
   type Element_Ptr is access all Element'Class;
   procedure Join(Q: access Queue; E: in Element_Ptr);
   function Remove(Q: access Queue) return Element_Ptr;
   function Length(Q: Queue) return Integer;
private
   type Element is tagged
      record
         Next: Element_Ptr;
      end record;
   type Queue is limited
      record
         First, Last: Element_Ptr;
         Count: Integer := 0;
      end record;
end Queues;
```

This package illustrates many points. We have hidden away the inner workings from the user by making both the type Queue and the type Element private. The type Queue is also limited since assignment must not be allowed (it would mess up the internal pointers); we have also made the full type explicitly limited just to ensure that the implementation of the body does not inadvertently attempt to assign a Queue either. (This is a useful safeguard because remember that we might later write a child package and that could see the private part also.) The type Element is given as tagged private. This means that the full type must also be tagged and permits the user to extend from the type without knowing its details.

The subprograms Join and Remove take an access parameter rather than an in out parameter. One advantage in Ada 2005 is that Remove can then be a function which is perhaps more convenient to use. A possible disadvantage is that we have to specifically create a reference to the queue and this means marking it as aliased or creating the queue with an allocator.

Using this package we can now declare the root type Reservation and so on

```
with Queues;
package Reservation_System is
   ...
   type Reservation is new Queues.Element with
      record
         ...
      end record;
   ...
end Reservation_System;
```

and then create and place reservations on a queue by statements such as

```
The_Queue: access Queue := new Queue;
...
New_Resvn: Reservation_Ptr := new Nice_Reservation;
...
Join(The_Queue, Element_Ptr(New_Resvn));
...
```

Removing a reservation from the queue can later be done by

```
Next_Resvn: Reservation_Ptr;
...
Next_Resvn := Reservation_Ptr(Remove(The_Queue));
Process_Reservation(Next_Resvn);
```

Note very carefully that we have to explicitly convert the result of Remove to the type Reservation_Ptr. This is because Remove returns a result of the type Element_Ptr. Nothing would have prevented us from putting any type derived from Element on the queue and so there is no guarantee that it is indeed a reservation. The conversion performs a check that the type referred to is indeed a member of Reservation'Class and Constraint_Error is raised if the check fails.

In Section 21.4 we shall see how we can ensure that only reservations are placed on such a queue.

In Ada 2012, the parameter Q of Join and Remove can be of mode in out since a function can have in out parameters. The_Queue can then be of type Queue rather than an access type.

Having introduced class wide types and access types to tagged types this seems a good moment to summarize the rules concerning type conversions and tagged types. The general principle is that we can only convert towards an ancestor, and run-time checks may be needed to ensure this if the source is a class wide type; if the check fails then Constraint_Error is raised. In detail

- Conversion between two specific types is only permitted towards the ancestor.
- Conversion from a specific type to a class wide type of any ancestor type is allowed. Conversion to a class wide type that is not of an ancestor is not allowed.
- Conversion from a class wide type to a specific type is allowed provided the type of the actual value is a descendant of the specific type. A dynamic check may be required.
- Conversion between two class wide types is allowed provided the actual value is in the target class. A dynamic check may be required.

(The terms ancestor type and descendant type include the type itself.)

Note that conversion from a specific type to a class wide type is allowed implicitly on initialization and parameter passing (which are similar contexts) but that an explicit named conversion must be given on assignment. This rule is relaxed in Ada 2012 if no check is needed on the conversion.

Conversion between access types is allowed provided the designated types can be converted in the same direction. If the conversion requires a check at run time that could fail then the conversion must be to a named access type. This is in contrast to checks concerning null exclusion where the destination type can always be an anonymous access type. See Section 11.7. Of course, all that happens when we convert between access types is that we get a different view of the same object and so the new view must be an allowed interpretation. We will come back to the topic of views and conversions in more detail in Section 14.5.

The fact that a specific type is essentially a subtype of a class wide type (we say the class wide type covers the specific type) is also relevant in extended return statements which were introduced in Section 10.1. We might have a function

```
function Get_Next( ... ) return Reservation'Class is
begin
   ...
   return NR: Nice_Reservation do
      ...
   end return;
   ...
end Get_Next;
```

This is permitted since the type of NR is covered by the return type of the function.

We conclude by noting that tagged type parameters are always passed by reference and considered aliased. The similarities and differences between in out and access parameters were discussed in Section 11.6. Note also that dispatching occurs with all modes of formal parameters including access parameters (and access results) but does not occur with parameters and results of a named access type.

Finally, note that the model of tagged types that we have introduced so far only permits strict trees of types and thus single inheritance. In the next section we shall introduce interfaces which permit multiple inheritance as described in Section 14.8. The very simple model of a dispatch table given in Figure 14.2 then has to be extended but the details need not concern the user.

Exercise 14.2

1 Declare a procedure that will print the area of any geometrical object of a type derived from the type Object.

2 In a traditional world women do not have beards and men do not bear children. However all persons have a date of birth. Declare a type Person with the common component Birth of type Date (as in Section 8.7) and then derived types Man and Woman that have additional components as appropriate indicating whether they have a beard or not and how many children they have borne respectively.

3 Declare procedures Print_Details for Person, Man and Woman which output information regarding the current values of their components. Then declare a procedure Analyse_Person which takes a parameter of the class wide type Person'Class and calls the appropriate procedure Print_Details.

4 Write the body of the package Queues.

14.3 Abstract types and interfaces

It is sometimes convenient to declare a type solely to act as the foundation upon which other types can be built by derivation and not in order to declare objects of the type itself. We can do this by marking a tagged type as abstract. An abstract type can have abstract primitive subprograms; these have no body and cannot be called but simply act as placeholders for operations to be added later. We thus might write

```
package P is
   type T is abstract tagged
      record ... end record;
   procedure Op(X: T) is abstract;
   ...
end P;
```

It is a very important rule that it is not possible to create an object of an abstract type. Hence no object can ever have a tag corresponding to an abstract type and so it is never possible to attempt to dispatch to a primitive subprogram of an abstract type. Note that an abstract type can have components and can also have concrete operations.

An interface is much like an abstract type but is more restricted. An interface cannot have components and it cannot have concrete operations (operations that are not abstract) except for null procedures which, it will be recalled from Section 12.3, behave as if they have a null body. Like an abstract type, we cannot declare objects of an interface. So we might write

```
package Q is
   type Int is interface;
   procedure Op(X: Int) is abstract;
   procedure N(X: Int) is null;
   procedure Action(X: Int'Class);          -- class wide operation
   ...
end Q;
```

Although an interface cannot have any concrete operations other than null procedures, we can declare a concrete procedure with a class wide parameter such as Action. Remember that such operations are not primitive and so not inherited.

Generally speaking, interfaces are more useful than abstract types because they permit multiple inheritance as explained in Section 14.8.

A concrete type (one that is not abstract) can be derived from an interface or from an abstract type provided that all inherited abstract subprograms are replaced by concrete subprograms which therefore have bodies (it is in this sense that the abstract subprograms are placeholders). We often speak of the concrete type as implementing the interface or abstract type.

We can now reformulate the example of processing reservations so that the root type Reservation is an interface and then build the specific types upon it. This enables us to program and compile all the infrastructure routines, such as Process_Reservation in the previous section, that deal with reservations in general without any concern at all for the individual reservation types and indeed before deciding what they should contain.

The baseline package can then simply become

```
package Reservation_System is
   type Reservation is interface;
   type Reservation_Ptr is access all Reservation'Class;
   procedure Make(R: in out Reservation) is abstract;
end Reservation_System;
```

in which we have declared the type Reservation as an interface with just the procedure Make as an abstract subprogram; remember that it does not have a body and hence the package also has no body.

We can now develop the reservation infrastructure and then later add the normal reservation system containing the three types of reservations. We introduce a child package as follows

```
package Reservation_System.Subsonic is

   type Position is (Aisle, Window);
   type Meal_Type is (Green, White, Red);

   type Basic_Reservation is new Reservation with
      record
         Flight_Number: Integer;
         Date_Of_Travel: Date;
         Seat_Number: String(1 .. 3) := "   ";
      end record;

   -- now provide concrete subprogram for abstract Make
   -- and add other subprograms as necessary

   overriding
   procedure Make(BR: in out Basic_Reservation);
   procedure Select_Seat(BR: in out Basic_Reservation);

   type Nice_Reservation is new Basic_Reservation with
      record
         Seat_Sort: Position;
         Food: Meal_Type;
      end record;

   overriding
   procedure Make(NR: in out Nice_Reservation);
   procedure Order_Meal(NR: in Nice_Reservation);

   type Posh_Reservation is new Nice_Reservation with
      record
         Destination: Address;
      end record;

   overriding
   procedure Make(PR: in out Posh_Reservation);
   procedure Arrange_Limo(PR: in Posh_Reservation);

end Reservation_System.Subsonic;
```

In this revised formulation we must provide a concrete procedure Make for the concrete type Basic_Reservation in order to implement the interface Reservation. The procedure Select_Seat now takes a parameter of type Basic_Reservation and the type Nice_Reservation is more naturally derived from Basic_Reservation.

Note carefully that we did not make Select_Seat an abstract subprogram in the package Reservation_System. There was no need; it is only Make that is required by the general infrastructure such as the procedure Process_Reservation and to add anything else would weaken the abstraction and clutter the interface with unnecessary operations.

We have chosen to include the overriding indicator **overriding** whenever we declare a further Make. This ensures that if we make a simple typographical error such as misspelling Make or getting the parameters wrong then the compiler will detect the error and not just add a bogus operation. We could of course add **not overriding** to the declarations of Select_Seat and so on.

We also have to make corresponding changes to the package body. This is left as an exercise for the reader.

When we now add the Supersonic_Reservation we can choose to derive this from the baseline Reservation as before or perhaps from some other point in the tree picking up the existing facilities of one of the other levels.

As an example of the use of an abstract type rather than an interface, consider the types Person, Man and Woman of Exercise 14.2(**2**). We really do not want to be able to declare objects of the type Person because they are incomplete; we want all real persons to be either of the type Man or Woman and the type Person is merely a convenience for the common properties such as the component Birth of type Date. We can prevent the declaration of objects of the type Person by making it abstract

```
type Person is abstract tagged
   record
      Birth: Date;
   end record;
```

and then the types Man and Woman can be derived as before.

Despite the type Person being abstract there is no reason why we should not continue to declare the concrete procedure Print_Details of Exercise 14.2(**3**)

```
procedure Print_Details(P: in Person) is
begin
   Print_Date(P.Birth);
end Print_Details;
```

Although we will never dispatch to this procedure because there can never be an object of the type Person itself, nevertheless it can be used to print the common information of the types derived from Person. We will return to this topic in Section 14.5.

It is possible to derive an abstract type from a concrete type or from another abstract type. In either case we can replace inherited concrete subprograms by abstract ones or concrete ones, add additional abstract or concrete subprograms and so on. The overall rule is simply that a concrete type cannot have abstract subprograms. It is possible to derive an interface from one or more other interfaces but not from types whether abstract or concrete. See Section 14.8.

Although we can declare a concrete subprogram with parameters of an abstract type such as Print_Details above, a function returning an abstract type (or an access to an abstract type) must always be abstract. A related situation occurs in the case of a primitive function that returns a concrete type when that type is extended. Clearly the function cannot return a value of the extended type (since it does not know how to provide values for the new components) and so the function requires overriding with a new definition providing an appropriate result (an example is the type Text_Map in Section 24.10). If the derived type is declared as abstract then we do not need to provide a new definition – the inherited one just becomes abstract. These rules do not apply if the extension is in fact null since the function can be inherited in the normal way.

We conclude with a final remark on the syntax. A tagged type declaration always has just one of **interface**, **tagged** and **with** (it has none if it is not a tagged type).

Exercise 14.3

1 Reformulate the type Object as an abstract type. Make the function Area abstract and make Distance take a class wide parameter. (Remember that it is best to use a class wide parameter rather than inheritance if by its very nature the operation will apply to all types in the class without change.) Then declare types Point and Circle etc. from the type Object.

2 Declare a function Further that takes two parameters of type Object and returns the one further from the origin by comparing their distances. Can this be inherited and thus applied to objects of the types Point and Circle?

3 Declare a function Further that takes two parameters of type Object'Class and returns the further one by comparing their distances.

4 Repeat the previous two exercises for a function Bigger that bases the comparison on the areas of the objects.

5 Write the body of the package Reservation_System.Subsonic.

14.4 Primitive operations and tags

In this section we consider in a little more detail some of the fundamental properties of tagged types and their operations.

It will be remembered from Section 12.3 that the primitive operations of a type are

- various predefined operations such as assignment and equality,
- those inherited from its ancestors and,
- those declared in the same package specification as the type itself.

We have said ancestors here rather than parent because of the existence of multiple inheritance to be discussed in Section 14.8.

For the sake of discussion we assume that we have the following declarations of packages Root and Shapes much as in the answer to Exercise 14.3(**1**)

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

   function Area(O: Object) return Float is abstract;
   function MI(O: Object) return Float is abstract;
   function Distance(O: Object'Class) return Float;

end Root;

package body Root is
   function Distance(O: Object'Class) return Float is
   begin
      return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
   end Distance;
end Root;

with Root;
package Shapes is
   type Circle is new Root.Object with
      record
         Radius: Float;
      end record;

   function Area(C: Circle) return Float;
   function MI(C: Circle) return Float;

   type Triangle is new Root.Object with
      record
         A, B, C: Float;        -- lengths of sides
      end record;

   function Area(T: Triangle) return Float;
   function MI(T: Triangle) return Float;

   ...   -- and so on for other types such as Square
end Shapes;
```

Then (apart from the predefined operations) the primitive operations of Root are Area and MI. The function Distance is not a primitive operation because it is class wide. Similarly the primitive operations of Circle and Triangle are the overridden versions of Area and MI.

We can call these operations by the usual technique of giving the subprogram name and then the parameters in parentheses (only one parameter in these cases). Thus we might write

```
A := Shapes.Area(A_Circle);
D := Root.Distance(A_Triangle);
M := Shapes.MI(A_Square);
```

These calls all mention the name of the package containing the operation which means that Distance has to be called differently from the others. Moreover, we might later decide to restructure the hierarchy into a set of packages thus

```
package Geometry is
   type Object is abstract ...
   ...   -- functions Area, Mi, Distance
end Geometry;

package Geometry.Circles is
   type Circle is new Object with...
   ...   -- functions Area, MI
end Geometry.Circles;

package Geometry.Triangles is
   type Triangle is new Object with...
   ...   -- functions Area, MI
end Geometry.Triangles;
```

This is a much more elegant structure and avoids having to write Root.Object when doing the extensions. But the assignments now become

```
A := Geometry.Circles.Area(A_Circle);
D := Geometry.Distance(A_Triangle);
M := Geometry.Squares.MI(A_Square);
```

This is really all rather a nuisance and unnecessarily emphasizes the location of the operations. In object oriented programming it is the object that should be dominant and not the location of the operations. Accordingly, an alternative notation is allowed for calling operations of tagged types.

The rule is that if an operation Op on a type T is declared in a package P and X is of type T, then a call

```
P.Op(X, other paras)        -- package P mentioned
```

can be replaced by

```
X.Op(other paras)           -- package P not mentioned
```

provided that

- T is a tagged type,
- Op is a primitive (dispatching) or class wide operation of T,
- X is the first parameter of Op.

The reason there is never any need to mention the package is that, by starting from the object, we can identify its type and thus the primitive operations of the type. Note that a class wide operation can be called in this way only if it is declared at the same place as the primitive operations of T (or one of its ancestors).

Using this notation the various assignments now become

```
A := A_Circle.Area;
D := A_Triangle.Distance;
M := A_Square.MI;
```

and of course since there are no extra parameters in these cases, there are no parentheses either. It is important to note that these calls are independent of the package structure which helps with program maintenance.

This prefixed notation also has some other advantages. One is that it unifies the notation for calling a function with a single parameter and directly reading a component of the object. Thus we can write uniformly

```
X := A_Circle.X_Coord;
A := A_Circle.Area;
```

(Of course if we were foolish and had a visible component Area as well as a function Area then we could not call the function in this way.)

Another advantage is that explicit dereferencing is often not necessary when using access types. Suppose we have

```
type Pointer is access all Geometry.Object'Class;
...
This_One: Pointer := A_Circle'Access;
```

Then using the package notation we have to write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(Geometry.Area(This_One.all));
```

whereas using the prefixed notation we can uniformly write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(This_One.Area);
```

It is important to note that the first parameter of an operation plays a special role since it has to be of the tagged type concerned. Treating the first parameter especially can seem odd in some circumstances such as when there is symmetry among the parameters. Accordingly, we shall only use the prefixed notation in examples when it seems appropriate.

We will now turn to the rules for dispatching. A parameter of a primitive operation which is of the type concerned is known as a controlling parameter and a corresponding actual parameter is a controlling operand. The basic principle is that dispatching only occurs when calling a primitive operation and a controlling operand is class wide. Thus the call

```
Make(RC);                    -- RC of type Reservation'Class
```

in the procedure Process_Reservation in Section 14.2 is a dispatching call. The value of the tag of RC is used to determine which procedure Make to call and this is determined at run time.

On the other hand a call such as

```
Make(Reservation(NR));     -- NR of type Nice_Reservation
```

in the package body of Reservation_System in Section 14.1 is not a dispatching call because the type of the operand is the specific type Reservation as a result of the explicit type conversion.

The subprograms Make, Select_Seat and so on have just a single controlling parameter. However, a primitive operation can have several controlling parameters but they must all be of the same type.

Thus it would not be possible to declare

```
procedure Something(C: Circle; T: Triangle);
```

in the specification of the package Shapes above containing the declarations of the types Circle and Triangle. Such a procedure could of course be declared in a different package but it would then not be a primitive operation of either Circle or Triangle. It could also be declared as a child subprogram because child subprograms are not primitive operations either. Another possibility is to make one parameter class wide so that it then becomes a primitive operation of the other type. (Incidentally, this restriction only applies to tagged types – a subprogram can be a primitive operation of more than one type provided at most one is tagged.)

A controlling parameter may be of any mode and it can also be an access parameter of the type. A function can also have a controlling result in which case the type must be the same as that of any controlling parameters.

The principle that all controlling operands and results of a call must be of the same type is very important. If they are statically determined then this is checked at compile time. If they are dynamically determined (for example, variables of a class wide type) then again the actual values must all be of the same specific type and of course this check has to be made at run time (the tags are compared) and Constraint_Error is raised if the check fails. In order to avoid confusion a mixed situation whereby some operands are static and some are dynamic is not allowed.

As an example consider the function

```
function Is_Further(X, Y: Object) return Boolean;
```

which we assume is primitive together with

```
T1, T2: Triangle;
C1, C2: Circle;
Obj_1: Object'Class := ... ;      -- must be initialized
Obj_2: Object'Class := ... ;      -- because a class wide type
```

Then we can write calls such as

```
Is_Further(T1, T2)            -- non-dispatching, type Triangle
Is_Further(C1, C2)            -- non-dispatching, type Circle
Is_Further(Obj_1, Obj_2)      -- dispatching
```

and in the last case a check is made before the call that Obj_1'Tag equals Obj_2'Tag. On the other hand the following are illegal for the reasons stated

```
Is_Further(T1, C2)          -- illegal – mixed specific types
Is_Further(Obj_1, T2)       -- illegal – mixed static and dynamic
```

and both these situations are detected at compile time. A type conversion could be used to overcome the restriction in the second case, thus

```
Is_Further(Obj_1, Object'Class(T2))          -- legal
```

It is also possible to dispatch on the result of a function when the context of the call determines the specific type. Suppose we have the following further primitive operations

```
function Unit return Object;
function Double(X: Object) return Object;
function Is_Bigger(X: Object; Y: Object := Unit) return Boolean;
```

(We assume now that Object is not abstract.) These rather unlikely functions might behave as follows. Unit returns an object of unit size such as a circle of unit diameter or a triangle of unit side. Double returns an object with twice the linear size as that of the parameter. Is_Bigger compares the area of two objects and by default takes a unit object as the second parameter.

The functions Unit and Double have controlling results. Consider

```
Is_Bigger(T1, Unit)         -- non-dispatching, type Triangle
Is_Bigger(Obj_1, Unit)      -- dispatching
```

In the first case, the controlling operand T1 is static and determines that the call of Unit is also static; the call of Unit is thus chosen at compile time to be the Unit with result of type Triangle. In the second case, the controlling operand Obj_1 is dynamic and determines the type at run time; in this case the call of Unit dispatches to the particular Unit with the same tag as Obj_1; there is no run-time check because only one controlling operand is used to determine the type. The call of Unit is thus like a chameleon and adapts to the circumstances; we say that it is *tag indeterminate*.

The situation can be nested, for example

```
Is_Bigger(Obj_1, Double(Obj_2))          -- dispatching
```

in which case the tags of Obj_1 and Obj_2 are checked to ensure that they are the same; Constraint_Error is raised if they are not. The more elaborate expression Double(Unit) is also tag indeterminate so we can have

```
Is_Bigger(C1, Double(Unit))         -- non-dispatching
Is_Bigger(Obj_1, Double(Unit))      -- dispatching
```

In the second case the call of Double is then determined by the specific type of Obj_1 and this in turn determines the call of Unit.

We can also use a call of a function such as Unit to determine a default value. Thus we can have

```
Is_Bigger(T1)               -- non-dispatching
Is_Bigger(Obj_1)            -- dispatching
```

and in the first case the default call of Unit is statically determined to be that of type Triangle whereas in the second case the call of Unit is dynamically determined by the specific type of the value of Obj_1.

It is interesting to note that a default expression for a controlling operand has to be tag indeterminate and so has to be a call of a function such as Unit or an expression such as Double(Unit). The reason is that we need to be able to use the default expression in both dispatching and non-dispatching contexts.

Note that an overridden operation does not have to have the same pattern of defaults as its parent; defaults are in the eye of the caller. Overload resolution identifies the declaration relevant to a call whether dispatching or not and it is the text of that declaration which gives any default expressions.

The use of overload resolution explains why the following call with two different class wide parameters fails even though the tags might be the same

```
Is_Further(Obj_1, CC)       -- illegal if CC is of type Circle'Class
```

This fails because neither the declaration of Is_Further for type Object nor that for type Circle is applicable. Such a mixed call would only be acceptable if there was a function Is_Further with at least one class wide formal parameter.

Similarly, if all controlling operands and any result are tag indeterminate then the situation is ambiguous and thus illegal. For example

```
Is_Further(Unit, Unit)      -- illegal
```

We are assuming that both Triangle and Circle are visible and so we do not know which Is_Further and Unit to apply. Again the overload resolution fails.

On the other hand, if only the type Circle is visible then such examples are not ambiguous and the type of Unit is *statically* determined to be Circle. An interesting example is that we could use a tag indeterminate expression as the initial value for a class wide object thus

```
CC: Circle'Class := Unit;    -- non-dispatching
```

The above discussion may have seemed a bit tedious but the underlying ideas are really quite simple and are aimed to make things as explicit as possible so that surprises are minimized or at least show up at compile time. A further example of the use of the various rules will be found in Section 14.9.

Assignment is a dispatching operation in the general case as we shall see in Section 14.7 when we consider controlled types. All the above rules apply and so both sides must have the same type and be statically or dynamically determined together.

So we can write

```
Obj_1 := Obj_2;
```

and there is a run-time check to ensure that the tags are the same (remember that a class wide object is constrained by its initial value). But, we cannot write

```
T1 := Obj_1;                -- illegal
Obj_2 := C2;                -- illegal
```

because in both cases we are mixing static and dynamic tags. We therefore have to provide an appropriate conversion such as

```
Obj_2 := Object'Class(C2);        -- run-time check
```

On the other hand, *implicit* conversion to a class wide type is permitted when providing an initial value in a declaration or as an actual parameter.

Equality is also a primitive operation but has somewhat different rules. If we compare two class wide values and they have different tags then the result False is returned rather than raising Constraint_Error which would occur with assignment or any other operation with two controlling operands. There is a strong analogy with arrays. We can only assign one array to another if the lengths are the same but equality simply returns False if the lengths are different and never raises Constraint_Error. But operations such as **and** and **or** on one-dimensional arrays raise Constraint_Error if the lengths are different.

Equality is also different with regard to inheritance; this is because it is normally predefined and so is always expected to be available and have sensible properties. On the other hand we might wish to redefine equality; directly inheriting a redefined version could bring surprises. Suppose we decide that two values of the type Object are to be considered equal if they are located within some small distance, epsilon, of the same point. We might declare

```
function "=" (A, B: Object) return Boolean is
begin
   return (A.X_Coord – B.X_Coord)**2
         + (A.Y_Coord – B.Y_Coord)**2 < Epsilon**2;
end "=";
```

The normal rules for inheritance would mean that this would be inherited unchanged by Point and Circle. This would be very surprising for the type Circle since it would ignore the Radius component completely. On the other hand, without redefinition of "=" for the type Object, the predefined equality for Circle would have applied predefined equality to all its components including the Radius. Because this drastic change of behaviour would be so surprising, what actually happens is that the equality for Object is not inherited for Circle but simply incorporated into the predefined equality for Circle. So two circles would be equal if their centres were within epsilon and their radii exactly equal.

The full rule for predefined equality of a type extension is that the primitive operation (possibly redefined) is used for the parent part and for any tagged components in the extension whereas predefined equality is always used for untagged components in the extension.

Recalling the discussion in Section 12.4, we see that tagged types (and indeed all record types in Ada 2012) always compose for equality. We are guaranteed that all private types in the predefined library that export equality compose properly – many will be implemented as tagged types.

Having redefined equality for Object, if we now wanted to redefine equality for Circle so that two circles are equal if they are at exactly the same location but their radii are within epsilon then we can either use the underlying coordinates directly

or we can retrieve the original equality for the type Object if we had had the foresight to rename it before the redefinition (a two pass algorithm!) by writing

```
function Old_Equality(A, B: Object) return Boolean renames "=";
```

(Such a renaming is sometimes called a squirrelling renaming.) Of course a renaming after the redefinition will refer to the new operation. All renamings in a package specification are genuine primitive operations and have their own slots in the dispatch table which are initialized with the operation that is current at the point of the renaming. These new names thus denote primitive operations in their own right and can themselves be overridden on later inheritance. This might seem peculiar because the principle of renaming is that no new entity is ever created. But this is still true, the new slots just give further ways of referring to existing entities.

We mentioned earlier that an access parameter can also be a controlling parameter. However, it is an important rule that a controlling operand can never be null since null has no tag and the tag is needed for dispatching or checking. So we should write

```
procedure Whatever(Ptr: not null access Object);
```

if Whatever is a primitive operation. If we attempt to call Whatever with a null parameter then Constraint_Error will be raised.

We can actually omit **not null** for compatibility with Ada 95 in the case of an access parameter that is a controlling parameter but the behaviour remains as if it were given. We recall the rule when discussing renaming in Section 13.7 that 'null exclusions never lie'. So if present it must be heeded, but if absent a null exclusion might still apply for other reasons as in this case.

Another example of this rule concerns renaming, null exclusions, and primitive operations (this example is strictly for the collector of Ada curios). Consider

```
package P is
   type T is tagged ...
   procedure One(X: access T);                    -- excludes null

   package Inner is
      procedure Deux(X: access T);                -- includes null
      procedure Trois(X: not null access T);      -- excludes null
   end Inner;

   use Inner;

   procedure Two(X: access T) renames Deux;       -- illegal
   procedure Three(X: access T) renames Trois;    -- legal
   ...
```

The procedure One is a primitive operation of T and its parameter X is therefore a controlling parameter and so excludes null even though this is not explicitly stated. However, the declaration of Two is illegal. It is trying to be a primitive operation of T and therefore its controlling parameter X has to exclude null. But Two is a renaming of Deux whose corresponding parameter does not exclude null and so the renaming is illegal. (Remember that primitive operations have to be declared immediately inside the package concerned and so subprograms declared in an inner

package are not primitive.) On the other hand the declaration of Three is permitted because the parameter of Trois does exclude null.

We have mentioned the existence of the tag of an object as being effectively a hidden component and we have seen how the value of the tag of a class wide object is used for dispatching. An object thus has the property of being self-describing; it carries an indication of the identity of its type with it.

In the previous section we considered the formulation of the type Person as

```
type Person is abstract tagged
   record
      Birth: Date;
   end record;

type Man is new Person with
   record
      Bearded: Boolean;
   end record;

type Woman is new Person with
   record
      Children: Integer;
   end record;
```

Note that, perhaps worryingly, there is no explicit component indicating the sex of a person; it might appear as if there was no way to find out one's sex! But luckily this information is not lost since it is implicit in the tag. The tag can be implicitly tested by membership tests such as

```
if P in Woman then
   ...                          -- special processing for Women
end if;
```

where P is of the class wide type Person'Class.

Indeed, it is also possible to test the tag explicitly using the attribute Tag which can be applied to a value of a class wide type and to a tagged type itself. So we could alternatively have written

```
if P'Tag = Woman'Tag then
```

The value of the attribute Tag is of the private (but nonlimited) type Tag declared in the package Ada.Tags whose visible part has the form

```
package Ada.Tags is
   pragma Preelaborate(Tags);
   type Tag is private;
   pragma Preelaborable_Initialization(Tag);
   No_Tag: constant Tag;
   function Expanded_Name(T: Tag) return String;
   function Parent_Tag(T: Tag) return Tag;
   ...      -- other functions
   function Is_Abstract(T: Tag) return Boolean;
   Tag_Error: exception;
private
```

We can declare variables of the type Tag in the usual way. The function Expanded_Name returns the full dotted name of the type of the tag as a string (in upper case and starting with a root library unit). So if Obj is an object of type Object'Class whose value happens to have the specific type Circle where Circle is declared in the library package Objects then

```
Expanded_Name(Obj'Tag)
```

would return the string "OBJECTS.CIRCLE". There are also versions of Expanded_Name that return the result as a Wide_String or Wide_Wide_String.

The function Is_Abstract tells us whether the tag is that of an abstract type. It was added in Ada 2012.

It is important to remember that the attribute Tag cannot be applied to a value of a specific type but only to a value of a class wide type or to the *name* of a specific type. (The reason is explained in the next section.)

We could also write

```
if P in Woman'Class then
```

and this would then cover any types derived from Woman as well. It is possible to do the corresponding test using tags by using the function Parent_Tag also declared in Ada.Tags. But it is perhaps a little more elegant to use membership tests wherever possible.

The package Ada.Tags has a number of other functions which will be explained in Sections 21.6 and 23.7.

Exercise 14.4

1 Define "=" for the type Circle so that two circles are equal if they are within epsilon of the same location and their radii are within epsilon of each other. Assume that "=" for Object has been appropriately redefined.

2 Now define "=" for the type Circle so that they have to be at the same location but their radii are within epsilon of each other. Assume that Old_Equality was declared and renames the original equality on the type Object.

14.5 Views and redispatching

We now consider the rules for type conversion once more. Recall that the basic rule is that type conversion is only allowed towards the root of a tree of tagged types and so we can convert a Nice_Reservation into a Reservation. On the other hand we cannot convert a specific type away from the root; we have to use an extension aggregate even if there are no extra components.

We can however convert a value of a class wide type to a specific type thus

```
NR := Nice_Reservation(RC);
```

where RC is of the type Reservation'Class. In such a case there is a run-time check that the current value of the class wide object RC is of a specific type for which the conversion is possible. Hence it must be of the type Nice_Reservation or derived from it so that the conversion is not away from the root of the tree. In other words

there is a check that the value of RC is actually in Nice_Reservation'Class. Constraint_Error is raised if the check fails.

Some conversions are what is known as view conversions. This means that the underlying object is not changed but we merely get a different view of it. (Much as the private view and full view of a type are just different views; the type is still the same.) We met an example of a view conversion in Section 10.3 when we called the procedure Increment with the view conversion Integer(F) as parameter.

Most conversions of tagged types are view conversions. For example the conversion in

```
Make(Reservation(NR));
```

is a view conversion. The value passed to the call of Make (with parameter of type Reservation) is in fact the same value as held in NR (tagged types are *always* passed by reference) but we can no longer see the components relating to the type Nice_Reservation. And in fact the tag still relates to the underlying value and this might even be the tag for Posh_Reservation because it could have been view converted all the way towards the root of the tree.

Moreover, if we did an assignment as in

```
NR := Nice_Reservation(PR);
```

then the tag of NR is of course not changed. All that happens is that the components appropriate to the type of NR are copied from the object PR. We can also have a view conversion as the destination of an assignment

```
Nice_Reservation(PR) := NR;
```

and the components copied are then just those appropriate to the type of the view. We could even have a conversion on both sides such as

```
Object(A_Circle) := Object(A_Triangle);
```

This makes the X_Coord and Y_Coord of the circle the same as those of the triangle. Other components, and in particular the tag, are not changed.

It is indeed an important principle that the tag of an object (both specific and class wide) is never changed. In particular, the fact that a view conversion does not change the tag is very important for what is called redispatching.

It often happens that after one dispatching operation we apply a further common (and inherited) operation and so need to dispatch once more to an operation of the original type. If the original tag were lost then this would not be possible. An example of the seeds of this difficulty already lies in our reservation system for Ada Airlines.

Consider again

```
procedure Make(NR: in out Nice_Reservation) is
begin
   Make(Reservation(NR));      -- make as plain reservation
   Order_Meal(NR);
end Make;
```

in which there is a call of the procedure Order_Meal. This call is not a dispatching call because the parameter is of a specific type and indeed there is only one procedure Order_Meal. Inside the body of Order_Meal we would expect to deal just with an order from a nice reservation and would not anticipate having to take account of the fact that the order might have originated from a posh passenger. (Although one would hope that posh meals are indeed better than nice meals.)

Actually we *could* write

```
procedure Order_Meal(NR: Nice_Reservation) is
   NRC: Nice_Reservation'Class := NR;
begin
   if NRC in Posh_Reservation then
      ...                        -- order a posh meal
   else
      ...                        -- order a nice meal
   end if;
end Order_Meal;
```

where we have regained the original type by converting to the class wide type Nice_Reservation'Class.

We could avoid the burden of the assignment by putting the conversion in the test itself

```
if Nice_Reservation'Class(NR) in Posh_Reservation then
```

or by introducing a renaming

```
NRC: Nice_Reservation'Class renames Nice_Reservation'Class(NR);
```

although this is rather a mouthful.

Note also that we could alternatively have written the test as

```
if NRC'Tag = Posh_Reservation'Tag then
```

but remember that we cannot apply the attribute Tag to an object of a specific type. So we could not have avoided the introduction of the class wide variable by writing

```
if NR'Tag = Posh_Reservation'Tag then      -- illegal
```

This is disallowed because it would be very confusing to allow NR'Tag since we would naturally expect this always to be Nice_Reservation'Tag and to find that it had some other value would be strange.

However, it is of course against the spirit of the game to mess about inside Order_Meal to see if it was actually ordered by a posh passenger. The whole idea of programming by extension is that one should be able to write the body for Order_Meal without considering how the type system might be extended later. Indeed, the type Posh_Reservation (like Supersonic_Reservation) might be in a later package in which case we could not here refer to the type Posh_Reservation at all.

The proper approach is to use redispatching. We write a distinct procedure

```
procedure Order_Meal(PR: in Posh_Reservation);
```

for posh passengers and redispatch in the body of Make as follows

```
procedure Make(NR: in out Nice_Reservation) is
begin
   Make(Reservation(NR));       -- make as plain reservation
   Order_Meal(Nice_Reservation'Class(NR));       -- redispatch
end Make;
```

Redispatching occurs because we have converted the parameter to the class wide type Nice_Reservation'Class (remember that dispatching occurs if the actual parameter is class wide and the formal is specific). So it works properly and our posh passenger will now get a posh meal instead of just a nice one.

We will now leave our reservation system noting that there is an analogous possible difficulty with Select_Seat; it seems likely that the requests of nice and posh passengers for an aisle or window seat might be overlooked. We leave the consideration of this as an exercise.

The fact that view conversion does not change the tag explains why we can safely and usefully declare a concrete procedure for an abstract type such as the procedure Print_Details for the type Person. We can never declare an object of the type Person and so a dispatching call of this procedure is not possible. A static call is however possible with a view conversion as in

```
procedure Print_Details(W: in Woman) is
begin
   Print_Details(Person(W));                    -- view conversion
   Print_Integer(W.Children);
end;
```

but since the tag never changes no harm can arise.

As another example consider the innocuous looking

```
procedure Swap(X, Y: in out Object) is
   T: Object := X;
begin
   X := Y;  Y := T;
end Swap;
```

and consider its inheritance by the type Circle.

This does not work as we might have hoped because it only swaps the Object part of the Circle. Remember that inheritance only applies to the subprogram specification and the code of the body is completely unchanged unless overridden. All references to the type Object in the body remain as references to the Object view of the Circle. So the inherited version is effectively

```
procedure Swap(X, Y: in out Circle) is
   T: Object := Object(X);
begin
   Object(X) := Object(Y);  Object(Y) := T;
end Swap;
```

One solution is to write

```
procedure Swap(X, Y: in out Object'Class) is
   T: Object'Class := X;
begin
   X := Y;  Y := T;
end Swap;
```

where the assignments are effectively done by dispatching. Being a class wide operation it cannot be overridden. Alternatively

```
procedure Swap(X, Y: in out Object) is
   T: Object'Class := Object'Class(X);
begin
   Object'Class(X) := Object'Class(Y);  Object'Class(Y) := T;
end Swap;
```

can be overridden if desired and again does the assignments according to the specific type. Both solutions work if Object is abstract whereas the original would not even compile in that case.

The above examples can be summarized by considering the case of a type T with primitive operations A and B. Suppose furthermore that the implementation of A is conveniently performed by calling B so that the body of A has the form

```
procedure A(X: T) is
begin
   ...
   B(X);
   ...
end A;
```

Now suppose that a new type TT is declared as an extension of T and that the inherited operation A appears satisfactory but the inherited operation B needs to be overridden by a new version

```
procedure B(X: TT) is ...
```

This raises the problem of whether the inherited version of A applying to the type TT should internally call B of the type T or the overridden B for the type TT. The answer will clearly depend upon the particular application. If we want it to continue to call B for the type T then the code as above is correct. However, if we want it to call the overridden B applicable to TT then we must change the call of B in the body of A to

```
B(T'Class(X));
```

so that it then dispatches to the relevant B. Of course the behaviour of A for the type T remains unchanged since it just dispatches to the same specific operation as before. The danger in all this is that the (potential) error lies in the body of A for T and only shows up when the type is extended.

Looking back at the previous examples, the operations Make and Order_Meal play the roles of A and B in the first example whereas Swap and := play these roles in the second example.

Care should always be taken when inheriting any operation with a controlling operand which is an **out** parameter to ensure that the inherited version does provide appropriate values for any additional components. There is no corresponding risk with a function with a controlling result since such a function requires overriding when the type is extended as we saw in Section 14.3.

It is hoped that the discussion in this and the previous section has not seemed overly complex and detailed. Object oriented programming may be very flexible but it has its pitfalls and it is important that the reader be aware of these. Ada strives for clarity. The basic rule is that dispatching is only used if the actual parameter is of a class wide type and this is always clear at the point of the call. This simple rule coupled with the fact that the original tag is never lost should cause fewer surprises than the more obscure rules of some languages.

We conclude this section with a brief summary of the main points regarding tagged types.

- Record (and private) types can be tagged. Values of tagged types carry a tag with them. The tag indicates the specific type. A tagged type can be extended on derivation with additional components. The tag of an object can never be changed.
- The primitive operations of a type are those implicitly declared, plus, in the case of a type declared in a package specification, all subprograms with a parameter or result of that type also declared in the package specification.
- Primitive operations are inherited on derivation and can be overridden. If the derivation occurs in a package specification then further primitive operations can be added.
- Types and subprograms can be declared as abstract. An abstract subprogram does not have a body but one can be provided on derivation. An interface is a form of abstract type that has no components and no concrete types other than null procedures. Only interfaces and abstract tagged types can have abstract primitive subprograms.
- T'Class denotes the class wide type rooted at T. It is an indefinite type. An appropriate access type can designate any value of T'Class.
- Objects of a class wide type may be declared but they must be initialized and are then constrained by the tag of the initial value. Formal parameters of a class wide type are similarly constrained by the actual parameter.
- Type conversion must always be towards an ancestor. Implicit conversion from a specific type to an ancestor class wide type is allowed when passing parameters and on initialization.
- Parameters of tagged types are always passed by reference. They are also considered aliased so that the Access attribute can be applied.
- Calling a primitive operation with an actual parameter of a class wide type results in dispatching: that is the run-time selection of the operation according to the tag.

Finally, do remember that the tag is an intrinsic part of an object of a tagged type and can be thought of as a hidden component. However, unlike other components, the tag of an object can never be changed.

Exercise 14.5

1 In the revised procedure Make for Nice_Reservation why could we not write

```
Order_Meal(Reservation'Class(NR));      -- redispatch
```

2 Add a procedure Select_Seat appropriate for nice passengers (and better) and rewrite the procedure Make for the type Reservation to dispatch on Select_Seat.

14.6 Private types and extensions

We now come to a detailed consideration of the interaction between type extension, private types and child packages.

In Section 14.2 we saw that the type Element in the package Queue was declared as

```
type Element is tagged private;
```

The full type then also has to be tagged. The partial view could in fact be declared as abstract thus

```
type Element is abstract tagged private;
```

and this would be an advantage in this case since there is no point in the user being able to create objects of the type Element.

The rules regarding matching of full and partial views are as expected from the general principle that the full view must deliver the properties promised by the partial view.

If the partial (external) view is tagged then the external client can do type extension and so the full view must also be tagged but the reverse is not true. The partial view can be untagged and the full view can be tagged. Of course the external client cannot use any of the properties of type extension in such a situation. A similar pattern applies to the property of being abstract. If the partial view is abstract then the full view need not be but, on the other hand, if the partial view is not abstract then the full view cannot be abstract either.

Note that there is no such thing as a private interface. This is largely because interfaces have no components and so nothing to hide.

Limited types are somewhat different if both views are tagged; both partial and full view must be limited or not together. Another rule is that a tagged record can only have a limited component if the record is explicitly limited, or in the case of a type extension, if the ultimate ancestor is explicitly limited. These rules prevent potential difficulties with extension and dispatching.

Abstract and private types pose small problems with aggregates. We can only give a normal aggregate if all the components are visible. However, we can always use an extension aggregate if the ancestor part is private

```
Some_Element: Element;
...
(Some_Element with ... )
```

or (and this is especially useful if the type is abstract so that the object Some_Element cannot be declared), we can simply use the subtype name as explained in Section 14.1 thus

```
(Element with ... )
```

and components corresponding to the type Element will be default initialized. An alternative technique (perhaps a dirty trick) is to use a view conversion of some existing object of a concrete type derived from Element. Thus we might have

```
(Element(An_Object) with ... )
```

The type Element illustrates that we can extend from a private tagged type with additional components visible to the user even though the original components are hidden from the user. The reverse is also possible as we saw in Section 3.2; we can extend an existing type with additional components which are hidden from the external user. We could declare a type Shape using a private extension declaration and make visible the fact that the type Shape is derived from Object and yet keep the additional components hidden. Thus

```
package Hidden_Shape is
   type Shape is new Object with private;  -- private extension
   ...
private
   type Shape is new Object with              -- full type declaration
      record
         ...                    -- the private components
      end record;
   function Area(S: Shape) return Float;
end Hidden_Shape;
```

It is not necessary for the full declaration of Shape to be derived directly from the type Object. There might be one or more intermediate types such as Circle; all that matters is that Shape is ultimately derived from Object. If there are no extra components then we write **with null record**; as expected.

Of course, the type Shape could never be an existing type. Writing

```
type Shape is new Circle with null record;
```

in the private part does not make Shape the same as Circle but derived from it.

We can declare and override primitive operations (such as Area) in the private part or in the visible part. New and overridden operations in the visible part behave as expected; they are visible to both client and server. But operations in the private part bring interesting possibilities.

A new operation in the private part will have a new slot in the dispatch table even though it is not visible for all views of the type. It is an important principle that there is only one dispatch table for a type and it may be that some operations are not visible for some views.

A minor point is that an abstract type is not allowed to have private abstract operations because it would not be possible to override them and so it would be impossible to extend the type. This would be a serious violation of the abstraction

because the poor external user should be totally unaware of any private operations. A similar restriction is that a function with a controlling result cannot be a private operation because again it could not be overridden.

The final case is where we override an inherited visible operation such as Area for Shape in the private part. Despite not being directly visible, nevertheless the overridden operation will be used whether called directly or indirectly through dispatching. It is an important principle that the same operation is called both directly and indirectly for the same tag – this allows a fragment to be tested with static binding and then we know we will still get the same effect if the final program does dispatching.

As an example of the use of private extensions we will rewrite the basic geometry system as a collection of abstract data types. We declare each derived type in its own child package and make these packages into a hierarchy matching the type hierarchy. This ensures that each body can see all the components even though they are hidden from the external client. Remember that the private part and body of a child package can see the private part of the parent package. We might write

```
package Geometry is
   type Object is abstract tagged private;
   function Area(O: Object) return Float is abstract;
   function Distance(O: Object'Class) return Float;

   procedure Move(O: in out Object'Class; X, Y: in Float);
   function X_Coord(O: Object'Class) return Float;
   function Y_Coord(O: Object'Class) return Float;
private
   type Object is abstract tagged
      record
         X_Coord: Float := 0.0;
         Y_Coord: Float := 0.0;
      end record;
end Geometry;

package Geometry.Circles is
   type Circle is new Object with private;
   function Area(C: Circle) return Float;
   function Make_Circle(Radius: Float) return Circle;
   function Get_Radius(C: Circle) return Float;
private
   type Circle is new Object with
      record
         Radius: Float;
      end record;
end Geometry.Circles;

package Geometry.Triangles is
   type Triangle is new Object with private;
   function Area(T: Triangle) return Float;
   function Make_Triangle(A, B, C: Float) return Triangle;
   procedure Get_Sides(T: in Triangle; A, B, C: out Float);
```

```
private
   ...
end Geometry.Triangles;
```

and so on. Since the types are now private we have provided class wide selector functions for accessing the coordinates (these can have the same identifier as the hidden components) and specific constructor and selector subprograms such as Make_Circle, Get_Radius and so on.

We have not provided individual procedures for setting the two coordinates but a single procedure Move. This ensures that both coordinates get changed consistently. Thus if we wish to move an object from its default location at the origin (0.0, 0.0) to say (3.0, 4.0) then we write

```
A_Triangle.Move(3.0, 4.0);                    -- using prefixed notation
```

whereas if we had used two procedure calls such as

```
A_Triangle.Set_X_Coord(3.0);
A_Triangle.Set_Y_Coord(4.0);
```

then it might seem as if the triangle was transitorily at the point (3.0, 0.0). There are other risks as well – we might forget to set one component or accidentally set the same component twice. Other more serious risks will become apparent when we deal with tasking in Chapter 20.

This example also reveals another advantage of the prefixed notation. We can still write

```
X := A_Triangle.X_Coord;
```

irrespective of whether the component is read directly or via a function.

Constructor functions pose a problem. Suppose we decide to introduce a distinct type for equilateral triangles by declaring a type Equilateral_Triangle in a further child package Geometry.Triangles.Equilateral. The first interesting point is that this introduces a different form of specialization in which we do not need any additional components but rather a constraint between existing components, namely that the sides are equal. We need a new constructor function that takes just a single parameter.

But the real problem is that, as currently written, the new type will inherit the function Make_Triangle from its parent type (moreover, it does not require overriding because there are no additional components). But it will have three parameters and any overridden function will also need three parameters contrary to our requirements. We just do not want Make_Triangle for the new type at all, we want Make_Equilateral with a single parameter.

In order to solve this problem we have to arrange that Make_Triangle is not a primitive operation. This can be done in a number of ways. It could return the class wide type Triangle'Class – this works but is clumsy because a type conversion would be required after each call in order to obtain the specific type Triangle. We could put the function in a child package – it could then still see the private part but would no longer be primitive.

But perhaps the best solution is to make it a child function of the package Geometry.Triangles thus

```
function Geometry.Triangles.Make_Triangle(A, B, C: Float)
                                                    return Triangle;
```

An advantage of this approach is that a child function can be accessed without any extra use clause just as if it were a primitive operation. Moreover, we do not have to think of a name for a containing package.

The same approach using child subprograms can be taken with selector subprograms but there is often no need. In conclusion the package and function for equilateral triangles might be just

```
package Geometry.Triangles.Equilateral is
   type Equilateral_Triangle is new Triangle with private;
private
   type Equilateral_Triangle is new Triangle with null record;
end;
```

```
with Geometry.Triangles.Make_Triangle;
function Geometry.Triangles.Equilateral.Make_Equilateral(Side: Float)
                                                  return Equilateral_Triangle is
begin
   return (Make_Triangle(Side, Side, Side) with null record);
end;
```

Note how we have used an extension aggregate in the return statement. Note also that the package has no body.

Clearly this is a trivial example but it illustrates important principles. A constructor function such as Make_Triangle will often have to check that its parameters are consistent and perhaps raise an exception declared in the root package if they are not. The reader might care to consider the introduction of a type Isosceles and how we might convert between different forms of triangle. We will encounter a further example of this kind in Section 18.4.

We will now reconsider the reservation system using private extensions of the interface Reservation but by contrast will continue to declare all the types in a single package. Consider

```
package Reservation_System is
   type Reservation is interface;
   procedure Make(R: in out Reservation) is abstract;
   type Basic_Reservation is new Reservation with private;
   type Nice_Reservation is new Reservation with private;
   type Posh_Reservation is new Reservation with private;
private
   type Basic_Reservation is new Reservation with
      record
         Flight_Number: Integer;
         Date_Of_Travel: Date;
         Seat_Number: String(1 .. 3) := "   ";
      end record;
```

```
      overriding
      procedure Make(BR: in out Basic_Reservation);          -- overrides
      not overriding
      procedure Select_Seat(BR: in out Basic_Reservation);   -- new

      type Position is (Aisle, Window);
      type Meal_Type is (Green, White, Red);

      type Nice_Reservation is new Basic_Reservation with
        record
          Seat_Sort: Position;
          Food: Meal_Type;
        end record;

      procedure Make(NR: in out Nice_Reservation);           -- overrides
      procedure Select_Seat(NR: in out Nice_Reservation);    -- ditto
      procedure Order_Meal(NR: in Nice_Reservation);         -- new

      type Posh_Reservation is new Nice_Reservation with
        record
          Destination: Address;
        end record;

      procedure Make(PR: in out Posh_Reservation);           -- overrides
      procedure Order_Meal(PR: in Posh_Reservation);         -- ditto
      procedure Arrange_Limo(PR: in Posh_Reservation);       -- new

    end Reservation_System;
```

Externally all that is visible is that there are the various types and there is a procedure Make – this is all that is necessary in order to write a procedure such as Process_Reservation. The relationships between the types are not visible since they are just shown as deriving from Reservation; this is a good illustration of the fact that the full declaration of a private type need not be directly derived from the ancestor type given in the private extension.

Note also that Select_Seat and Order_Meal are private primitive operations (we have properly provided the various versions so that passengers get their appropriate choices of seat and meal). Moreover, although the redeclarations of Make are also in the private part nevertheless they are externally callable because the original operation Make is visible.

In Section 12.3 we mentioned that we cannot add further primitive operations to a type after a type has been derived from it. This is a consequence of the freezing rules which are discussed in Chapter 25; these rules concern when the representation of a type is determined or frozen and the effect of it being frozen. The two rules that concern us here are that first, we cannot add more primitive operations to a type once its representation is frozen (since this determines the dispatch table) and secondly, declaring a derived type freezes its parent (if not already frozen). Luckily this second rule applies only to the full type declaration and not to a private extension. Otherwise we could not declare the private dispatching operations such as Select_Seat. The full declaration of Basic_Reservation then freezes the type Reservation and prevents us from adding further primitive operations to it. Another point is that we cannot give the full type declaration corresponding to a type extension until after the full declaration of its

parent. It is therefore important that the various types and operations are declared in the proper order especially if they are all in the same package.

We can now add the supersonic package

```
package Reservation_System.Supersonic is
   type Supersonic_Reservation is new Reservation with private;
private
   type Supersonic_Reservation is new ... with
      record
         ...
      end record;

   procedure Make(SR: in out Supersonic_Reservation);
   procedure Select_Seat(SR: in out Supersonic_Reservation);
   ...
end Reservation_System.Supersonic;
```

The type Supersonic_Reservation can now be an extension of any member of the tree as appropriate and, being in a child package, it can see the full details of the various types. Note that Select_Seat overrides properly despite being a private primitive operation of Reservation.

We did not bother to give overriding indicators in all of the above but clearly there would have been no difficulty in doing so. But sometimes it is not known whether a subprogram is going to override and that is why they are optional (apart from considerations of compatibility with Ada 95). Consider

```
package P is
   type NT is new T with private;
   procedure Op(X: NT);
private
```

Now suppose the type T does not have an operation Op. Then clearly it would be wrong to write

```
package P is
   type NT is new T with private;          -- T has no Op
   overriding                              -- illegal
   procedure Op(X: NT);
private
```

because that would violate the information known in the partial view.

But suppose that in fact it turns out that in the private part the type NT is actually derived from TT (itself derived from T) and that TT does have an operation Op.

```
private
   type NT is new TT with ...              -- TT has Op
end P;
```

In such a case it turns out in the end that Op is in fact overriding after all. We can then put an overriding indicator on the body of Op since at that point we do know that it is overriding.

Equally of course we should not specify **not overriding** for Op in the visible part because that might not be true either (since it might be that TT does have Op). However, if we did put **not overriding** on the partial view then that would not in itself be an error but would simply constrain the full view not to be overriding and thus ensure that TT does not have Op.

Of course if T itself has Op then we could and indeed should put an overriding indicator in the visible part since we know that to be the truth at that point.

The general rule is not to lie. But the rules are slightly different for overriding and not overriding. For overriding it must not lie at the point concerned. For not overriding it must not lie anywhere.

This asymmetry is a bit like presuming the prisoner is innocent until proved guilty. We sometimes start with a view in which an operation appears not to be overriding and then later on we find that it is overriding after all. But the reverse never happens – we never start with a view in which it is overriding and then later discover that it was not. So the asymmetry is real and justified.

We cannot expect to find all the bugs in a program through syntax and static semantics; the key goal of overriding indicators is to provide a simple way of finding most of them.

We conclude this section by expanding on the remark made in Section 12.5 that the properties of a type visible from a given point depend upon the declaration of that type which is visible from that point. Consider

```
package P is
   type T is tagged private;
private
   type T is tagged
      record
         X: Integer;
      end record;
end;

with P;  use P;
package Q is
   type DT is new T with
      record
         Y: Integer;
      end record;
end;
```

Now suppose that the body of P has a with clause for Q and consider visibility of an object A of the type DT from within the body of P. We might write

```
with Q;  use Q;
package body P is
   A: DT;
   ...
begin
   A.X := 5;                -- illegal
   T(A).X := 5;             -- legal
end P;
```

Although DT is extended from T and the full declaration of T is visible from within the body of P, nevertheless the component X of A of type DT is not visible. The visibility is controlled by what we can see at the point where DT is declared and at that point we cannot see the component X. However, within the body of P, we could carry out a view conversion of A to the type T and then access the component X.

One consequence is that the additional component of DT could also have the identifier X. However, this would not be possible if DT were declared in a child package of P because then the private part and the body of the child would have direct visibility of both components and a clash of names would arise.

Exercise 14.6

1 Could we declare the functions Further of Exercises 14.3(**2**) and 14.3(**3**) for the type Shape in the private part of the package Hidden_Shape?

14.7 Controlled types

A very interesting example of the use of type extension is provided by considering the facilities for controlled types. These allow a user complete control over the initialization and finalization of objects and also provide the capability for user-defined assignment.

The general principle is that there are three distinct primitive activities concerning the control of objects

- initialization after creation,
- finalization before destruction,
- adjustment after assignment,

and the user is given the ability to provide appropriate procedures which are called to perform whatever is necessary at various points in the life of an object. These procedures are Initialize, Finalize and Adjust and they take the object as a parameter of mode in out.

To see how this works, consider

```
declare
   A: T;                    -- create A, Initialize(A)
begin
   ...
   A := E;                  -- Finalize(A), copy value, Adjust(A)
   ...
end;                        -- Finalize(A)
```

After A is declared and any normal default initialization carried out, the Initialize procedure is called. On an assignment, Finalize is first called to tidy up the old object about to be overwritten and thus destroyed, the physical copy is then made and finally Adjust is called to do whatever might be required for the new copy. At the end of the block Finalize is called once more before the object is destroyed.

Note, of course, that the user does not physically write the calls of the three control procedures, they are called automatically by the compiled code.

In the case of a declaration with an initial value

```
A: T := E;                    -- create A, copy value, Adjust(A)
```

the calls of Initialize and Finalize that would occur with distinct declaration and assignment are effectively cancelled out (but see the note below for the case when the expression E is an aggregate).

There are many other situations where the control procedures are invoked such as when calling allocators, evaluating aggregates and so on; the details are omitted but the principles will be clear.

In the case of a nested structure where inner components might themselves be controlled, such inner components are initialized and adjusted before the object as a whole and on finalization everything is done in the reverse order.

In order for a type to be controlled it has to be extended from one of two tagged types declared in the library package Ada.Finalization whose specification is as follows

```
package Ada.Finalization is
   pragma Preelaborate(Finalization);
   pragma Remote_Types(Finalization);

   type Controlled is abstract tagged private;
   pragma Preelaborable_Initialization(Controlled);

   procedure Initialize(Object: in out Controlled) is null;
   procedure Adjust(Object: in out Controlled) is null;
   procedure Finalize(Object: in out Controlled) is null;

   type Limited_Controlled is abstract tagged limited private;
   pragma Preelaborable_Initialization(Limited_Controlled);

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

We see that there are distinct abstract types for nonlimited and limited types. Naturally enough the Adjust procedure does not exist in the case of limited types because they cannot be copied.

(See Section 26.3 for a brief discussion on the Remote_Types pragma which applies to this package.)

This package also provides an example of null procedures. Remember that if a procedure specification has **is null** then the procedure behaves as if it has a null body. That is, if it is called, nothing happens.

As a simple example, suppose we wish to declare a type and keep track of how many objects (values) of the type are in existence and also record the identity number of each object in the object itself. We could declare

```
with Ada.Finalization;  use Ada.Finalization;
package Tracked_Things is

   type Thing is new Controlled with
      record
         Identity_Number: Integer;
         ...       -- other data
      end record;

   overriding
   procedure Initialize(Object: in out Thing);

   overriding
   procedure Adjust(Object: in out Thing);

   overriding
   procedure Finalize(Object: in out Thing);

end Tracked_Things;

package body Tracked_Things is

   The_Count: Integer := 0;
   Next_One: Integer := 1;

   procedure Initialize(Object: in out Thing) is
   begin
      The_Count := The_Count + 1;
      Object.Identity_Number := Next_One;
      Next_One := Next_One + 1;
   end Initialize;

   procedure Adjust(Object: in out Thing)
         renames Initialize;

   procedure Finalize(Object: in out Thing) is
   begin
      The_Count := The_Count - 1;
   end Finalize;

end Tracked_Things;
```

In this example we have considered each value of a thing to be a new one and so Adjust is the same as Initialize and we can conveniently use a renaming declaration to provide the body as was mentioned in Section 13.7. An alternative approach might be to consider new things to be created only when an object is first declared (or allocated). This variation is left as an exercise.

Another point is that we have used overriding indicators for the three overridden operations. This is good practice in this important area.

The observant reader will note that the identity number is visible to users of the package and thus liable to abuse. We can overcome this by using a child package in which we extend the type. This enables us to provide different views of a type and effectively allows us to create a type with some components visible to the user and some components hidden.

Consider

```
package Tracked_Things is
   type Identity_Controlled is abstract tagged private;
private
   type Identity_Controlled is abstract new Controlled with
      record
         Identity_Number: Integer;
      end record;
   overriding
   procedure Initialize ...
   ...   -- etc.
end Tracked Things;

package Tracked_Things.User_View is
   type Thing is new Identity_Controlled with
      record
         ...   -- visible data
      end record;
end Tracked_Things.User_View;
```

In this arrangement we first declare a private type Identity_Controlled just containing the component Identity_Number (this component being hidden from the user) and then in the child package we further extend the type with the visible data which we wish the user to see. Note carefully that the type Identity_Controlled is abstract so objects of this type cannot be declared and moreover the user cannot even see that it is actually a controlled type. We also declare Initialize, Adjust and Finalize in the private part and so they are also hidden from the user.

Using a child package allows any subprograms declared in it to access the hidden identity number. Note also that several different types could be derived from Identity_Controlled all sharing the same control mechanism. Other arrangements are also possible.

We finish this section with a few observations on the package Ada.Finalization. The procedures Initialize, Adjust and Finalize are null and this will often be an appropriate default. The types Controlled and Limited_Controlled are of course abstract.

A key reason for making the default procedures null is because it will often be the case that the Finalize procedure for a type will naturally include a call of the Finalize procedure for the parent type. We might write

```
type T is new Parent with
   record
      ...   -- additional components
   end record;
...
procedure Finalize(Object: in out T) is
begin
   ...      -- operations to finalize additional components
   Finalize(Parent(Object));
end Finalize;
```

This is particularly relevant if the parent is a generic formal parameter (see Section 19.3). So all we might know is that the parent is some controlled type; since the default Finalize is null it can always be called with impunity.

In some obscure circumstances Finalize can be called twice for the same object unless it is of a limited type. (It can happen with the abort statement; see Chapter 18.) It is therefore advisable to write Finalize so that any second call does not matter. Another thing to beware is that Finalize must not raise an exception – this would cause a bounded error.

Controlled types provide a good example of the use of the form of extension aggregate where the ancestor part is just given by a subtype mark. We can typically write

```
X: T := (Controlled with ... );
```

Note that we cannot easily give an expression for the ancestor part since its type is abstract. We could however, do a dirty trick with a view conversion by writing Controlled(A_Thing) for the ancestor part as explained in the previous section.

When an aggregate provides an initial value as in this example, the value is created directly in X and Adjust is not called for the object as a whole. (This is similar to the use of aggregates to initialize limited types as described in Section 11.5.) This behaviour is vital for the case of declaring an initialized object in the same package as the type T since calling any new Adjust for T at this stage would otherwise raise Program_Error because the body of the new Adjust will not yet have been elaborated.

We conclude with an important warning. Remember to spell Finalize with -ize and not with -ise. Declaring a procedure Finalise will simply add a new primitive operation leaving the inherited Finalize still applicable. This is a good example of the dangers inherent in the flexibility of dynamic binding. Misspelling when overriding a primitive operation can cause an error which can only be detected by debugging whereas misspelling a statically called subprogram usually causes a compile-time error. But, if we consistently use overriding indicators as recommended then writing

```
overriding
procedure Finalise( ... );                    -- illegal
```

will indeed be detected at compile time because the procedure Finalise is not an overriding but a new operation.

(The author is delighted that those who spell incorrectly are in danger. Those who recall their classical Greek will appreciate that -ize is etymologically correct. It is correct British English too as stated clearly by the *Oxford English Dictionary*, but sadly many British publishers as well as even *The Times* now use -ise inherited from our Gallic friends across the sea. Maybe this illustrates that it is always best to inherit from the ultimate ancestor!)

Exercise 14.7

1 Rewrite Initialize, Adjust and Finalize as necessary for the situation where we only consider new things to be created when an object is declared or allocated.

2 Show how to modify the key manager of Section 12.6 so that Return_Key is automatically called on scope exit.

14.8 Multiple inheritance

We now come to a very important facility introduced in Ada 2005. In Ada 95, the inheritance model was strictly single inheritance. In the examples so far we have always had a strict tree of types descended from a single root type or interface. Some other languages such as C++ have embraced full multiple inheritance and got into a bit of a mess. However, Java introduced the notion of interfaces which work well and interfaces in Ada are the key to multiple inheritance.

We have already seen some examples of interfaces when discussing the reservation system. In this section we give a more detailed description of the rules regarding interfaces and one simple example of multiple inheritance but more elaborate examples must wait until Chapter 21 since it is helpful to have explained the generic mechanism as well.

General multiple inheritance has problems. Suppose that we have a type T with some components and operations. Perhaps

```
type T is tagged
   record
      A: Integer;
      B: Boolean;
   end record;

procedure Op1(X: T);
procedure Op2(X: T);
```

We then derive two new types from T thus

```
type T1 is new T with
   record
      C: Character;
   end record;

procedure Op3(X: T1);
                              -- Op1 and Op2 inherited, Op3 added
type T2 is new T with
   record
      C: Colour;
   end record;

procedure Op1(X: T2);
procedure Op4(X: T2);
                              -- Op1 overridden, Op2 inherited, Op4 added
```

Now suppose that we were able to derive a further type from both T1 and T2 by perhaps writing

```
type TT is new T1 and T2 with null record;          -- illegal
```

This is about the simplest example one could imagine. We have added no further components or operations. But what would TT have inherited from its two parents?

There is a general rule that a record cannot have two components with the same identifier so presumably it has just one component A and one component B. But what about C? Does it inherit the character or the colour? Or is it illegal because of the clash? Suppose T2 had a component D instead of C. Would that be allowed? Would TT then have four components?

And then consider the operations. Presumably it has both Op1 and Op2. But which implementation of Op1? Is it the original Op1 inherited from T via T1 or the overridden version inherited from T2? Clearly it cannot have both. But there is no reason why it cannot have both Op3 and Op4, one inherited from each parent.

The problems arise when inheriting *components* from more than one parent and inheriting different *implementations* of the same operation from more than one parent. There is no problem with inheriting the same specification of an operation from two parents.

These observations provide the essence of the solution. At most one parent can have components and at most one parent can have concrete operations – for simplicity they have to be the same parent. But abstract operations can be inherited from several parents. This can be phrased as saying that this kind of multiple inheritance is about merging contracts to be satisfied rather than merging algorithms or state.

As we have already seen, an interface is an abstract tagged type with no components and no concrete operations except for null procedures. And hence there is no problem with inheriting from several interfaces plus one normal tagged type.

So the main type derivation rule in Ada is that a tagged type can be derived from zero or one conventional tagged types plus zero or more interface types. Thus

```
type NT is new T and Int1 and Int2 with ... ;
```

where Int1 and Int2 are interface types. The normal tagged type if any has to be given first in the declaration. The first type is known as the parent so the parent could be a normal tagged type or an interface. The other types are known as progenitors. Additional components and operations are allowed in the usual way.

The term progenitors may seem strange but the term ancestors in this context is confusing and so a new term is introduced. Progenitors comes from the Latin *progignere*, to beget, and so is very appropriate.

Suppose that the interface Int1 and type T are in packages with operations thus

```
package P1 is
   type Int1 is interface;
   procedure Op1(X: Int1) is abstract;
   procedure N1(X: Int1) is null;
end P1;

package P is
   type T is tagged record ... end record;
   procedure Opt1( ... );
   procedure Opt2( ... );
end P;
```

We could now write

```
with P1, P;
package PNT is
   type NT is new P.T and P1.Int1 with ... ;
   procedure Op1(X: NT);                    -- concrete procedure
   ...   -- possibly other ops of NT
end PNT;
```

We must of course provide a concrete procedure for Op1 inherited from the interface Int1 since we have declared NT as a concrete type. We could also provide an overriding for N1 but if we do not then we simply inherit the null procedure of Int1. We could also override the inherited operations Opt1 and Opt2 from T in the usual way and possibly add some quite new ones.

Interfaces can be composed from other interfaces thus

```
type Int2 is interface;
...
type Int3 is interface and Int1;
...
type Int4 is interface and Int1 and Int2;
```

When we compose interfaces in this way we can add new operations so that the new interface such as Int4 will have all the operations of both Int1 and Int2 plus possibly some others declared specifically as operations of Int4. All these operations must be abstract or null and there are fairly obvious rules regarding what happens if two or more of the progenitors have the same operation. Thus a null procedure overrides an abstract one with the same (that is conformant) profile and repeated operations with the same profile must have the same convention.

We refer to all the interfaces in an interface list as progenitors. So Int1 and Int2 are the progenitors of Int4. The first one is not a parent – that term is only used when deriving a type as opposed to composing an interface. Note that the term ancestor covers all generations whereas parent and progenitors are first generation only.

Similar rules apply when a tagged type is derived from another type plus one or more interfaces as in the case of the type NT which was

```
type NT is new T and Int1 and Int2 with ... ;
```

In this case it might be that T already has some of the operations of Int1 and/or Int2. If so then the operations of T must match those of Int1 or Int2.

We informally speak of a specific tagged type as implementing an interface from which it is derived (directly or indirectly). Thus in the above example the tagged type NT must implement all the operations of the interfaces Int1 and Int2. If the type T already implements some of the operations then the type NT will automatically implement them because it will inherit the implementations from T. It could of course override such inherited operations in the usual way.

The normal rules apply in the case of functions. Suppose one operation is a function F thus

```
package P2 is
   type Int2 is interface;
   function F(Y: Int2) return Int2 is abstract;
end P2;
```

then the type NT must provide a concrete function to override F (unless the type extension of NT is null) even though T might already have such a conforming operation as in

```
package P is
   type T is tagged record ...
   function F(X: T) return T;
   ...
end P;
```

The new function F will then override the functions F of both T and Int2.

Class wide types also apply to interface types and an interface can have class wide operations such as

```
procedure Action(X: Int1'Class);
```

In the body of Action we would typically have calls of the primitive operations of the interface and these would be dispatching calls thus

```
procedure Action(X: Int1'Class) is
begin
   ...
   Op1(X);                -- dispatching call
   ...
   N1(X);                 -- dispatching call
end Action;
```

An example of this important structure will be found in Section 21.2.

Interfaces can also be used in private extensions. Thus

```
   type PT is new T and Int2 and Int3 with private;
   ...
private
   type PT is new T and Int2 and Int3 with ... ;
```

An important rule regarding private extensions is that the full view and the partial view must agree with respect to the set of interfaces they implement. Thus although the parent in the full view need not be T but can be any type derived from T, the same is not true of the interfaces which must be such that they both implement the same set exactly. This rule is important in order to prevent a client type from overriding private operations of the parent if the client implements an interface added in the private part. This is sometimes known as the rule that there are 'no hidden interfaces'.

We can compose mixtures of limited and nonlimited interfaces but if any one of them is nonlimited then the resulting interface must not be specified as limited. This is because it must implement the equality and assignment operations implied by the

nonlimited interface. Similar rules apply to types which implement one or more interfaces.

If an interface is limited then it must be stated explicitly

```
type LI is limited interface;                -- limited
type NLI is interface;                       -- nonlimited
type I is limited interface and LI;          -- limited
```

This contrasts with the situation regarding normal types

```
type LT is tagged limited ...                -- limited
type NLT is tagged ..                        -- nonlimited
type T is new LT with ...                    -- limited
```

In the case of types, limitedness is inherited from the parent type and does not have to be stated (this is for compatibility with Ada 83 and Ada 95) whereas in the case of interfaces it always has to be stated. However, **limited** can optionally be given on a normal type derivation and is always required if we derive a limited type from a limited interface thus

```
type T is limited new LT with ...            -- limited optional
type T is limited new LI with ...            -- limited mandatory
```

These rules all really come down to the same thing. If any parent or progenitor is nonlimited then the descendant must be nonlimited. In other words if a type (including an interface) is limited then all its ancestors must also be limited.

We conclude with an amusing example. Suppose we wish to inherit from both the type Geometry.Object and a type People.Person. This might be in order to simulate the Flatlanders of the book *Flatland* written by Edwin Abbott in 1884. This book is a satire on class structure and concerns a world in which people are flat geometrical objects. The working classes are triangles, the middle classes are other polygons and the aristocracy are circles. Sadly, all females are two-sided and thus simply a line segment. We would therefore like to write something like

```
type Flatlander is new People.Person and Geometry.Object with ...
```

where the type Person might be in a package People thus

```
package People is
  type Person is abstract tagged
    record
      Birth: Date;
      ...
    end record;
  ...                              -- various operations on Person
end People;
```

The earlier type Object had components. We cannot inherit from two types with components and so one of them must be turned into an interface. We can turn the type Object into an interface by writing

```
package Geometry is
  type Object is interface;

  procedure Move(O: in out Object; X, Y: in Float) is abstract;
  function X_Coord(O: Object) return Float is abstract;
  function Y_Coord(O: Object) return Float is abstract;
  function Area(O: Object) return Float is abstract;
  ...
end Geometry;
```

We can now declare an abstract type Flatlander. Remember that the first type is the only one that can have components and so the type Person has to be first in the list. In other words the type Person is the parent type and the type Object is a progenitor type. We write

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

  procedure Move(F: in out Flatlander; X, Y: Float);
  function X_Coord(F: Flatlander) return Float;
  function Y_Coord(F: Flatlander) return Float;

private
  type Flatlander is abstract new Person and Object with
    record
      X_Coord, Y_Coord: Float := 0.0;
    end record;
end;
```

The type Flatlander will now inherit the components such as Birth from the type Person, any operations of the type Person and the abstract operations of the type Object. Moreover, it is convenient to declare the coordinates as components since we need to do that eventually and we can then override the inherited abstract operations Move, X_Coord and Y_Coord with concrete ones. The type Flatlander is abstract since we are not yet in a position to provide a concrete version of a function such as Area. The package body is straightforward.

We can now declare a type Square suitable for Flatland (when originally written the book was published anonymously under the pseudonym A_Square) as follows

```
package Flatland.Squares is

  type Square is new Flatlander with
    record
      Side: Float;
    end record;

  function Area(S: Square) return Float;

end;

package body Flatland.Squares is

  function Area(S: Square) is
```

```
    begin
      return S.Side**2;
    end Area;

end Flatland.Squares;
```

and all the operations are thereby implemented. By way of illustration we have made the the extra component Side of the type Square directly visible. We can now declare

```
A_Square: Square := (Flatlander with Side => 4.00);
```

and he will have all the properties of a square and a person and will be located by default at the origin. Incidentally, Dr Abbott had a proper education at St John's College, Cambridge. He read theology, classics and mathematics just like the type Student of Section 8.7.

This concludes a basic description of the use of interfaces for multiple inheritance. Another example will be found in the next section and more elaborate examples will be found in Chapter 21. Note also that there are other forms of interfaces, namely synchronized interfaces, task interfaces and protected interfaces. They will be described in Chapter 22.

14.9 Multiple implementations

We conclude this chapter on the fundamentals of OOP in Ada with a discussion on an important characteristic of object oriented programming – the ability to provide different implementations of a single abstraction. Of course one can do this statically by writing a package with alternative bodies but only one body can appear in one program.

Interfaces and inheritance enable different types to be treated as different realizations of a common abstraction. The tag of an object indicates its implementation and allows a dynamic binding between the client and the appropriate implementation.

We can thus develop different implementations of a single abstraction, such as a family of set types, as in the next example. We naturally start with an interface type

```
package Abstract_Sets is

   type Set is interface;

   function Empty return Set is abstract;
   function Unit(E: Element) return Set is abstract;
   function Union(S, T: Set) return Set is abstract;
   function Intersection(S, T: Set) return Set is abstract;
   procedure Take(From: in out Set; E: out Element) is abstract;
end Abstract_Sets;
```

This package provides an abstract specification of sets where we assume that Element is some discrete subtype such as Integer or Colour. The type Set is an

interface. The package also defines a set of abstract primitive operations for the interface Set. An implementation of the abstraction can then be created by deriving a concrete type from the interface Set and providing concrete operations for that implementation.

The primitive operations are Empty which returns the null set, Unit which builds a set of one element, Union and Intersection as expected, and Take which removes one (arbitrary) element from the set.

One possible implementation would be to use a Boolean array as in Section 8.6 where each element represents the presence or absence of a member in the set as follows.

```
with Abstract_Sets;
package Bit_Vector_Sets is

   type Bit_Set is new Abstract_Sets.Set with private;

   function Empty return Bit_Set;
   function Unit(E: Element) return Bit_Set;
   function Union(S, T: Bit_Set) return Bit_Set;
   function Intersection(S, T: Bit_Set) return Bit_Set;
   procedure Take(From: in out Bit_Set; E: out Element);

private

   type Bit_Vector is array (Element) of Boolean;

   type Bit_Set is new Abstract_Sets.Set with
      record
         Data: Bit_Vector;
      end record;
end Bit_Vector_Sets;

package body Bit_Vector_Sets is

   function Empty return Bit_Set is
   begin
      return (Data => (others => False));
   end;

   function Unit(E: Element) return Bit_Set is
      S: Bit_Set := Empty;
   begin
      S.Data(E) := True;
      return S;
   end;

   function Union(S, T: Bit_Set) return Bit_Set is
   begin
      return (Data => S.Data or T.Data);
   end;
      ...                        -- Intersection
      ...                        -- and Take
end Bit_Vector_Sets;
```

Such an implementation is only appropriate if the number of values in the subtype Element is not too large (note that we would typically pack the array anyway using the aspect Pack described in Chapter 25).

An alternative implementation more appropriate to sparse sets might be based on using a linked list containing the elements present in a set. For such an implementation we would have to redefine equality and assignment as well as the abstract operations; we will return to this in a moment.

But the really interesting thing is that we could then write a program which contained both forms of sets; we could convert from one representation to any other by using

```
procedure Convert(From: in Set'Class; To: out Set'Class) is
   Temp: Set'Class := From;
   E: Element;
begin
   -- build target set, element by element
   To := Empty;
   while Temp /= Empty loop
      Take(Temp, E);
      To := Union(To, Unit(E));
   end loop;
end Convert;
```

This works by extracting the elements one at a time from the source set From and then building them up into the target set To. It is instructive to consider the fine details of how it dispatches onto the appropriate operations according to the specific type of its parameters using the rules described in Section 14.4.

The first action is to copy the original set into the class wide variable Temp. This avoids damaging the original set. Remember that all variables of class wide types such as Temp have to be initialized since class wide types are indefinite.

We then have

```
To := Empty;
```

which illustrates the case of a tag being indeterminate. The function Empty has a controlling result but no controlling operands to determine the tag; the choice of function to call has to be determined by the tag of the class wide parameter To which is the destination of the assignment.

We then come to

```
while Temp /= Empty loop
```

where the dispatching equality operator has two controlling operands. Since Empty is tag indeterminate the tag of Temp is used to determine which Empty and then which equality operator to call.

The statement

```
Take(Temp, E);
```

has the single controlling operand Temp and so dispatches according to the tag of Temp which is that of From.

Finally the statement

```
To := Union(To, Unit(E));
```

also causes dispatching to Unit and Union according to the tag of To.

An interesting property of the procedure Convert is that nothing can go wrong; it is not possible for two controlling operands to have different tags and thus raise Constraint_Error. Indeed there are no checks but just enough information to do the dispatching correctly. The algorithm used by Convert also works if the *formal* parameters are specific types in which case no dispatching occurs since all the types are known statically – the variable Temp would also have to be of the appropriate specific type as well.

We mentioned above that assignment is also a dispatching operation although this is not often apparent. In this example, however, if the type of From were a linked list then a deep copy would be required otherwise the original value could be damaged when the copy is decomposed. Such a deep copy can be performed by using a controlled type as described in Section 14.7.

The general idea is that the set is implemented as a record containing a single component of the controlled type Inner which itself contains a single component of the type Cell_Ptr which is an access to the usual linked list containing the actual elements. Assigning the record also assigns the inner component and this invokes the procedure Adjust for the controlled component which then copies the whole list. The implementation might be as follows

```
with Abstract_Sets;
with Ada.Finalization;  use Ada.Finalization;
package Linked_Sets is

   type Linked_Set is new Abstract_Sets.Set with private;
   ...   -- the various operations on Linked_Set

private
   type Cell;
   type Cell_Ptr is access Cell;

   type Cell is
      record
         Next: Cell_Ptr;
         E: Element;
      end record;

   type Inner is new Controlled with
      record
         The_Set: Cell_Ptr;
      end record;

   procedure Adjust(Object: in out Inner);

   type Linked_Set is new Abstract_Sets.Set with
      record
         Component: Inner;
      end record;

end Linked_Sets;
```

```
package body Linked_Sets is
   function Copy(P: Cell_Ptr) return Cell_Ptr is      -- deep copy
   begin
      if P = null then
         return null;
      else
         return new Cell'(Copy(P.Next), P.E);
      end if;
   end Copy;

   procedure Adjust(Object: in out Inner) is
   begin
      Object.The_Set := Copy(Object.The_Set);
   end Adjust;
   ...
end Linked_Sets;
```

The procedure Adjust for Inner performs a deep copy by calling the function Copy with its single component The_Set as parameter. Performing an assignment on the type Linked_Set (such as Temp := From;) causes Adjust to be called on its inner component thereby making the deep copy; Figure 14.4 shows the situation (a) just after the bitwise copy is made but before Adjust is called, and (b) after Adjust is called.

Note that the type Linked_Set as a whole is not controlled because it is simply derived from Abstract_Sets.Set which is not. Nevertheless it behaves rather as if it were controlled because of its controlled component and so exhibits a hidden form of multiple inheritance; see Section 21.1. But of course none of the mechanism is visible to the user. Observe that we do not need to provide a procedure Initialize and

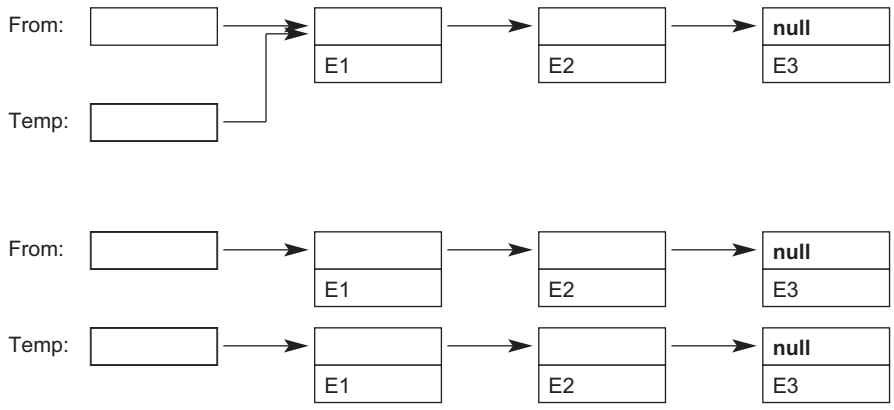


Figure 14.4 Two stages in the deep copy assignment.

that we have not bothered to provide Finalize although it would be wise to do so in order to discard unused space; see Section 25.4.

We could alternatively have used multiple inheritance so that the full type is

```
type Linked_Set is new Controlled and Abstract_Sets.Set with
  record
    The_Set: Cell_Ptr;
  end record;
```

and this avoids the introduction of the type Inner. Again the type Linked_Set is not visibly controlled. The procedure Adjust then becomes

```
procedure Adjust(Object: in out Linked_Set) is
begin
   Object.The_Set := Copy(Object.The_Set);
end Adjust;
```

The details of the remaining subprograms are left to the imagination of the reader. Some thought is necessary in order to avoid excessive manipulation. It is probably best to arrange that elements are not duplicated. It might also be advisable to keep the elements on the list in some canonical order otherwise the equality operation will be tedious.

As another example we might have an interface defining a stack thus

```
package Abstract_Stacks is

   type Stack is interface;

   function Empty return Stack is abstract;
   procedure Push(S: in out Stack; E: in Element) is abstract;
   procedure Pop(S: in out Stack; E: out Element) is abstract;
end Abstract_Stacks;
```

This could also be implemented in various ways and again we could define a procedure Convert which changed one representation into another. Moreover we could even have a type which is both a set and a stack. We would write

```
with Abstract_Sets, Abstract_Stacks;
package Mixture is
   type Strange is new Abstract_Sets.Set and
                                    Abstract_Stacks.Stack with private;

   function Empty return Strange;
   function Unit(E: Element) return Strange;
   function Union(S, T: Strange) return Strange;
   function Intersection(S, T: Strange) return Strange;
   procedure Take(From: in out Strange; E: out Element);

   procedure Push(S: in out Strange; E: in Element);
   procedure Pop(S: in out Strange; E: out Element);

private
   ...
end;
```

An interesting feature here is that the function Empty is an abstract operation of both progenitors. Since they have the same profile the one operation of the type Strange can implement both. The reader might wonder how the type Strange can be both a set and a stack. That is entirely up to the writer of the package Mixture. It could be essentially a stack each of whose elements is a set or indeed vice versa. All that matters from the point of view of the language is that the various operations are implemented in a manner consistent with their profiles.

Exercise 14.9

1 Given

```
B1, B2: Bit_Set;
L1: Linked_Set;
C1: Set'Class := ... ;
C2: Set'Class := ... ;
```

then which of the following are legal and which involve a run-time check?

(a) Union(B1, B2)
(b) Union(B1, L1)
(c) Union(B1, C1)
(d) Union(C1, C2)

2 The procedure Convert will convert between any two sets but is somewhat inefficient if the two sets happen to have the same representation. Modify it to overcome this inefficiency.

3 Declare a procedure Convert that will convert between stacks of different representations. Sketch two representations of a stack using an array and linked lists much as in Sections 12.4 and 12.5.

Checklist 14

Additional primitive operations can only be added if derivation is in a package specification.

Additional primitive operations cannot be added to the parent type after a full type is derived from it.

Conversion of tagged types can only be towards the root.

Objects of a class wide type must be initialized.

Dispatching only occurs with a class wide actual parameter and a specific formal parameter.

It is not possible to derive from a class wide type.

Tagged type parameters are always passed by reference.

Tagged formal parameters are considered aliased.

Objects of an abstract type or interface are not allowed.

The operations of an interface must be abstract or null procedures.

Functions returning an abstract type must themselves be abstract.

Functions returning a tagged type require overriding on extension unless the extension is null.

The attribute `Tag` cannot be applied to an object of a specific type.

Equality has special rules on extension.

The tag of an object can never be changed.

An abstract type cannot have private abstract operations.

Controlling operands are never **null**.

Extension at an inner level is permitted in Ada 2005.

Incomplete types can be marked as tagged in Ada 2005.

Functions do not require overriding in Ada 2005 if the extension is null.

The prefixed notation was added in Ada 2005.

Interfaces were added in Ada 2005.

New in Ada 2012

The only significant change in this area concerns synchronized interfaces which are discussed in Section 22.3.

15 Exceptions

At various times in the preceding chapters we have said that if something goes wrong when the program is executed, then an exception, often Constraint_Error, will be raised. In this chapter we describe the exception mechanism and show how remedial action can be taken when an exception occurs. We also show how we may define and use our own exceptions. Exceptions concerned with interacting tasks are dealt with in Chapter 20.

15.1 Handling exceptions

We have seen that if we break various language rules then an exception may be raised when we execute the program. There are four predefined exceptions (declared in the package Standard) of which we have met three so far

Constraint_Error	This generally corresponds to something going out of range; this includes when something goes wrong with arithmetic such as an attempt to divide by zero.
Program_Error	This will occur if we attempt to violate the control structure in some way such as running into the **end** of a function, breaking the accessibility rules or calling a subprogram whose body has not yet been elaborated – see Sections 10.1, 11.7 and 12.1.
Storage_Error	This will occur if we run out of storage space, as for example if we called the recursive function Factorial with a large parameter – see Section 10.1.

The other predefined exception is Tasking_Error. This is concerned with tasking and so is dealt with in Chapter 20.

Note that for historical reasons the exception Constraint_Error is also renamed as Numeric_Error in the package Standard. Moreover, Numeric_Error is considered obsolescent and so should be avoided.

If we anticipate that an exception may occur in a part of a program then we can write an exception handler to deal with it. For example, suppose we write

```
begin
   ...   -- sequence of statements
exception
   when Constraint_Error =>
      ...   -- do something
end;
```

If Constraint_Error is raised while we are executing the sequence of statements between **begin** and **exception** then the flow of control is interrupted and immediately transferred to the sequence of statements following the =>. The clause starting **when** is known as an exception handler.

As an illustration we could compute Tomorrow from Today by writing

```
begin
   Tomorrow := Day'Succ(Today);
exception
   when Constraint_Error =>
      Tomorrow := Day'First;
end;
```

If Today is Day'Last (that is, Sun) then the attempt to evaluate Day'Succ(Today) causes the exception Constraint_Error to be raised. Control is then transferred to the handler for Constraint_Error and the statement Tomorrow := Day'First; is executed. Control then passes to the end of the block.

This is really a bad example. Exceptions should be used for rarely occurring cases or those which are inconvenient to test for at their point of occurrence. By no stretch of the imagination is Sunday a rare day. Over 14% of all days are Sundays. Nor is it difficult to test for the condition at the point of occurrence. So we should really have written

```
if Today = Day'Last then
   Tomorrow := Day'First;
else
   Tomorrow := Day'Succ(Today);
end if;
```

However, it is a simple example with which to illustrate the mechanism.

Several handlers can be written between **exception** and **end**. Thus we might have

```
begin
   ...   -- sequence of statements
exception
```

```
      when Constraint_Error | Program_Error =>
         Put("Constraint Error or Program Error occurred");
         ...
      when Storage_Error =>
         Put("Ran out of space");
         ...
      when others =>
         Put("Something else went wrong");
         ...
   end;
```

In this example a message is output according to the exception. Note the similarity to the case statement. Each **when** is followed by one or more exception names separated by vertical bars. As usual we can write **others** but it must be last and on its own; it handles any exception not listed in the previous handlers.

Note that we can mention the same exception twice in the same handler; this is useful when exceptions are renamed such as in the case of historic programs which might have a common handler for Constraint_Error and its renaming Numeric_Error.

Exception handlers can appear at the end of a block, extended return statement, subprogram body, package body (also task body and accept statement, see Chapter 20) and have access to all entities declared in that unit. The examples have shown a degenerate block in which there is no **declare** and declarative part; the block was introduced just to provide somewhere to hang the handlers. We could rewrite the bad example to determine tomorrow as a function thus

```
function Tomorrow(Today: Day) return Day is
begin
   return Day'Succ(Today);
exception
   when Constraint_Error =>
      return Day'First;
end Tomorrow;
```

It is important to realize that control can never be returned directly to the unit where the exception was raised. The sequence of statements following => replaces the remainder of the unit containing the handler and thereby completes execution of the unit. Hence a handler for a function must generally contain a return statement in order to provide the 'emergency' result.

Note that a goto statement cannot transfer control from a unit into one of its handlers or vice versa or from one handler to another. However, the statements of a handler can otherwise be of arbitrary complexity. They can include blocks, calls of subprograms and so on. A handler of a block could contain a goto statement which transferred control to a label outside the block and it could contain an exit statement if the block were inside a loop.

Observe that a handler at the end of a package body applies only to the initialization sequence of the package and not to any subprograms in the package. Such subprograms must have individual handlers if they are to deal with exceptions.

We now consider the question of what happens if a unit does not provide a handler for a particular exception. The answer is that the exception is propagated

dynamically. This simply means that the unit is terminated and the exception is raised at the point where the unit was executed.

In the case of a block we therefore look for a handler in the unit containing the block and similarly for an extended return. In the case of a subprogram, the call is terminated and we look for a handler in the unit which called the subprogram. In the case of a package body we consider the unit containing the declaration of the package.

This unwinding process is repeated until either we reach a unit containing a handler for the particular exception or come to the top level. If we find a unit containing a relevant handler then the exception is handled at that point. However, if we reach the main subprogram and have still found no handler then the main subprogram is abandoned and we can expect the run-time environment to provide us with a suitable diagnostic message. (Unhandled exceptions in tasks are dealt with in Sections 20.8 and 22.5.) The other possibility is that an unhandled exception might occur during the elaboration of a library unit in which case the program is abandoned before the main subprogram is called.

It is important to understand that exceptions are propagated dynamically and not statically. That is, an exception not handled by a subprogram is propagated to the unit calling the subprogram and not to the unit containing the declaration of the subprogram – these may or may not be the same.

If the statements in a handler themselves raise an exception then the unit is terminated and the exception propagated to the calling unit; the handler does not loop.

Exercise 15.1

Note: these are exercises to check your understanding of exceptions. They do not necessarily reflect good Ada programming techniques.

1 Assuming that calling Sqrt with a negative parameter and attempting to divide by zero both raise Constraint_Error, rewrite the procedure Quadratic of Section 10.3 without explicitly testing D and A.

2 Rewrite the function Factorial of Section 10.1 so that if it is called with a negative parameter (which would normally raise Constraint_Error) or a large parameter (which would normally raise Storage_Error or Constraint_Error) then a standard result of say –1 is returned. Hint: declare an inner function Slave which actually does the work.

15.2 Declaring and raising exceptions

Relying on the predefined exceptions to detect unusual but anticipated situations is usually bad practice because they do not provide a guarantee that the exception has in fact been raised because of the anticipated situation. Something else may have gone wrong instead.

For example, consider the package Stack of Section 12.1. If we call Push when the stack is full then the statement Top := Top + 1; will raise Constraint_Error and if we call Pop when the stack is empty then Top := Top – 1; will also raise Constraint_Error. Since Push and Pop do not have exception handlers, the exception will be propagated to the unit calling them. So we could write

```
declare
   use Stack;
begin
   ...
   Push(M);
   ...
   N := Pop;
   ...
exception
   when Constraint_Error =>
      ...   -- stack manipulation incorrect?
end;
```

Misuse of the stack now results in control being transferred to the handler for Constraint_Error. However, there is no guarantee that the exception has arisen because of misuse of the stack; something else could have gone wrong instead.

A better solution is to raise an exception specifically declared to indicate misuse of the stack. Thus the package could be rewritten as

```
package Stack is
   Error: exception;
   procedure Push(X: Integer);
   function Pop return Integer;
end Stack;

package body Stack is
   Max: constant := 100;
   S: array (1 .. Max) of Integer;
   Top: Integer range 0 .. Max;

   procedure Push(X: Integer) is
   begin
      if Top = Max then
         raise Error;
      end if;
      Top := Top + 1;
      S(Top) := X;
   end Push;

   function Pop return Integer is
   begin
      if Top = 0 then
          raise Error;
      end if;
      Top := Top – 1;
      return S(Top + 1);
   end Pop;

begin
   Top := 0;
end Stack;
```

The exception Error is declared in a similar way to a variable. Several exceptions could be declared in the same declaration.

An exception can be raised by an explicit raise statement naming the exception. The handling and propagation rules for user-defined exceptions are just as for the predefined exceptions. We can now write

```
declare
   use Stack;
begin
   ...
   Push(M);
   ...
   N := Pop;
   ...
exception
   when Error =>
      ...   -- stack manipulation incorrect
   when others =>
      ...   -- something else went wrong
end;
```

We have now successfully separated the handler for misusing the stack from the handler for other exceptions.

Note that if we had not provided a use clause then we would have had to refer to the exception in the handler as Stack.Error; the usual dotted notation applies.

What could we expect to do in the handler in the above case? Apart from reporting that the stack manipulation has gone wrong, we might also expect to reset the stack to an acceptable state although we have not provided a convenient means of doing so. A procedure Reset in the package Stack would be useful. A further thing we might do is to relinquish any resources that were acquired in the block and might otherwise be inadvertently retained. Suppose for instance that we had also been using the package Key_Manager of Section 12.6. We might then call Return_Key to ensure that a key declared and acquired in the block had been returned. Remember that Return_Key does no harm if called unnecessarily.

In the case of controlled types discussed in Section 14.7, we do not have to call Finalize since that is done automatically when we leave the block anyway.

We would probably also want to reset the stack and return the key in the case of any other exception as well; so it would be sensible to declare a procedure Clean_Up to do all the actions required. So our block might look like

```
declare
   use Stack, Key_Manager;
   My_Key: Key;

   procedure Clean_Up is
   begin
      Reset;
      Return_Key(My_Key);
   end;
```

```
begin
   Get_Key(My_Key);
   ...
   Push(M);
   ...
   Action(My_Key, ... );
   ...
   N := Pop;
   ...
   Return_Key(My_Key);
exception
   when Error =>
      Put("Stack used incorrectly");
      Clean_Up;
   when others =>
      Put("Something else went wrong");
      Clean_Up;
end;
```

Note that we could make the return of the key automatic by using a controlled type for the key as in Exercise 13.7(**2**). Leaving a unit via a handler will invoke any Finalize procedures properly.

We have rather assumed that Reset is a further procedure declared in the package Stack but note that we could write our own curious procedure externally as follows

```
procedure Reset is
   Junk: Integer;
   use Stack;
begin
   loop
      Junk := Pop;
   end loop;
exception
   when Error =>
      null;
end Reset;
```

This works by repeatedly calling Pop until Error is raised. We then know that the stack is empty. The handler needs to do nothing other than prevent the exception from being propagated; so we merely write **null**. This procedure seems a bit like trickery; it would be far better to have a reset procedure in the overall package Stack.

Sometimes the actions that require to be taken as a consequence of an exception need to be performed on a layered basis. In the above example we returned the key and then reset the stack but it is probably the case that the block as a whole cannot be assumed to have done its job correctly. We can indicate this by raising an exception as the last action of the handler, thus

```
exception
   when Error =>
      Put("Stack used incorrectly");
      Clean_Up;
      raise Another_Error;
   when others =>
      ...
end;
```

The exception Another_Error will then be propagated to the unit containing the block. We could put the statement

```
raise Another_Error;
```

in the procedure Clean_Up.

Sometimes it is convenient to handle an exception and then propagate the same exception. This can be done by just writing

```
raise;
```

This is particularly useful when we handle several exceptions with the one handler. So we might have

```
   when others =>
      Put("Something else went wrong");
      Clean_Up;
      raise;
end;
```

The current exception will be remembered even if the action of the handler raises and handles its own exceptions such as occurred in our trick procedure Reset. However, note that there is a rule that we can only write **raise**; directly in a handler and not for instance in a procedure called by the handler such as Clean_Up.

The stack example illustrates a legitimate use of exceptions. The exception Error should rarely, if ever, occur and it would also be inconvenient to test for the condition at each possible point of occurrence. To do that we would presumably have to provide an additional parameter to Push of type Boolean and mode **out** to indicate that all was not well, and then test it after each call. In the case of Pop we would also have to recast it as a procedure since a function cannot take a parameter of mode **out**.

The package specification would then become

```
package Stack is
   procedure Push(X: in Integer; B: out Boolean);
   procedure Pop(X: out Integer; B: out Boolean);
end;
```

and we would have to write

```
declare
   use Stack;
   OK: Boolean;
```

```
begin
   ...
   Push(M, OK);
   if not OK then ...     end if;
   ...
   Pop(N, OK);
   if not OK then ...     end if;
end;
```

It is clear that the use of an exception provides a better structured program.

Note that nothing prevents us from explicitly raising one of the predefined exceptions. We recall that in Section 10.1 when discussing the function Inner we stated that probably the best way of coping with parameters whose bounds were unequal was to explicitly raise Constraint_Error.

Ada 2012 introduced conditional expressions as described in Chapter 9. Another feature added to Ada 2012 after it became a standard (see AI12-22) is the ability to explicitly raise an exception within an expression. So just as we can write

```
if X > Y then
   Z := +1;
elsif X < Y then
   Z := –1;
else
   raise Some_Error;
end if;
```

we can equally write

```
Z := (if X < Y then +1 elsif X > Y then –1 else raise Some_Error);
```

An exception can similarly be explicitly raised as part of a case expression or indeed as part of any expression. But a raise expression cannot be just **raise** on its own.

In Section 9.3 we wondered whether the case statement assigning values to Object_Ptr in Program 1 might be better done as a case expression. Indeed, we could have a raise expression to handle the exceptional case of an incorrect character. But we would then need a handler to exit the loop which would not really be an improvement.

Exercise 15.2

1 Rewrite the package Random of Exercise 12.1(**1**) so that it declares and raises an exception Bad if the initial value is not odd.

2 Rewrite the answer to Exercise 15.1(**2**) so that the function Factorial always raises Constraint_Error if the parameter is negative or too large.

3 Declare a function "+" which takes two parameters of type Vector and returns their sum using sliding semantics by analogy with the predefined one-dimensional array operations described in Section 8.6. Use type Vector from Section 8.2. Raise Constraint_Error if the arrays do not match.

4 Are we completely justified in asserting that Stack.Error could only be raised by the stack going wrong?

5 Assuming that the exception Error is declared in the specification of Stacks, rewrite procedures Push and Pop of Section 12.5 so that they raise Error rather than Storage_Error and Constraint_Error.

15.3 Checking and exceptions

In the previous section we came to the conclusion that it was logically better to check for the stack overflow condition ourselves rather than rely upon the built-in check associated with the violation of the subtype of Top. At first sight the reader may well feel that this would reduce the execution efficiency of the program. However this is not necessarily so, assuming a reasonably intelligent compiler, and this example can be used to illustrate the advantages of the use of appropriate subtypes.

We will concentrate on the procedure Push; similar arguments apply to the function Pop.

First consider the original package Stack of Section 12.1. In that we had

```
S: array (1 .. Max) of Integer;
Top: Integer range 0 .. Max;

procedure Push(X: Integer) is
begin
   Top := Top + 1;
   S(Top) := X;
end Push;
```

If the stack is full (that is Top = Max) and we call Push then it is the assignment to Top that raises Constraint_Error. This is because Top has a range constraint. However, the only run-time check that needs to be compiled is that associated with checking the upper bound of Top. There is no need to check for violation of the lower bound since the expression Top + 1 could not be less than 1 (assuming that the value in Top is always in range). Note also that no checks need be compiled with respect to the assignment to S(Top). This is because the value of Top at this stage must lie in the range 1 .. Max (which is the index range of S) – it cannot exceed Max because this has just been checked by the previous assignment and it cannot be less than 1 since 1 has just been added to its previous value which could not have been less than 0. So just one check needs to be compiled in the procedure Push.

The type Integer is itself considered to be constrained as we shall see in the next chapter. So if the variable Top was simply declared as

```
Top: Integer;
```

then a check would still be applied to the assignment to Top (it might be done by the hardware), and checks would also have to be compiled for the assignment to S(Top) in order to ensure that Top lay within the index range of S. So three checks would be necessary in total, one to prevent overflow of Top and two for the index range.

So applying the range constraint to Top actually reduces the number of checks required from three to one. This is typical behaviour given a compiler with a moderate degree of flow analysis. The more you tell the compiler about the properties of the variables (and assuming the constraints on the variables match their usage), the better the object code.

Now consider what happens when we add our own test as in the previous section (and we assume that Top now has its range constraint)

```
procedure Push(X: Integer) is
begin
   if Top = Max then
      raise Error;
   end if;
   Top := Top + 1;
   S(Top) := X;
end Push;
```

Clearly, we have added a check of our own. As a consequence, there is now no need for the compiler to insert the check on the upper bound of Top in the assignment statement

```
Top := Top + 1;
```

because our own check will have caused control to be transferred away via the raising of the Error exception for the one original value of Top that would have caused trouble. So the net effect of adding our own check is simply to replace the one compiler check by one of our own; the net result is the same and the object code is not less efficient.

There are two morals to this tale. The first is that we should tell the compiler the whole truth about our program; the more it knows about the properties of our variables, the more likely it is to be able to keep the checks to the appropriate minimum. In fact this is just an extension of the advantage of strong typing discussed in Section 6.3 where we saw how arbitrary run-time errors can be replaced by easily understood compile-time errors.

The second moral is that introducing our own exceptions rather than relying upon the predefined ones need not reduce the efficiency of our program. In fact it is generally considered bad practice to rely upon the predefined exceptions for steering our program and especially bad to raise the predefined exceptions explicitly ourselves. It is all too easy to mask an unexpected genuine error that needs fixing.

It should also be noted that we can always ask the compiler to omit the run-time checks by using the pragma Suppress. This is described in more detail in Section 15.5.

Finally, an important warning. Our analysis of when checks can be omitted depends upon all variables satisfying their constraints at all times. Provided checks are not suppressed we can be reasonably assured of this apart from one nasty loophole. This is that we are not obliged to supply initial values in the declarations of variables in the first place. So they can start with a junk value which does not satisfy any constraints and may not even be a value of the base type. If such a variable is read before being updated then our program has a bounded error and all

our analysis is worthless. We should therefore ensure that all variables are initialized or updated before they are read. See Section 26.6. Beware however, that giving variables gratuitous initial values can obscure flow errors and so is not generally a good idea.

Exercise 15.3

1 Consider the case of the procedure Push with explicit raising of Error but suppose that there is no range constraint on Top. How many checks would then be required?

15.4 Exception occurrences

There are a number of auxiliary facilities which enable a more detailed analysis of exceptions and their cause. In the clean-up clause in Section 15.2 we wrote

```
    when others =>
      Put("Something else went wrong");
      Clean_Up;
    raise;
  end;
```

This is not very helpful since we would really like to record what actually happened. We can do this using an exception occurrence which identifies both the exception and the instance of its being raised (that is, the circumstances associated with the particular error condition).

The limited type Exception_Occurrence is declared in the child package Ada.Exceptions together with various subprograms including functions Exception_Name, Exception_Message and Exception_Information. These functions take an exception occurrence as their single parameter and return a string (which has lower bound of 1).

As their names suggest, Exception_Name returns the name of the exception (the full dotted name in upper case) and Exception_Message and Exception_Information return two levels of more detailed information which identify the cause and location. The result of Exception_Message should be a one-line message and not contain the exception name whereas Exception_Information will include both the name and the message and might provide full details of a trace back as well. Although the details are dependent upon the implementation the general intent is that the messages are suitable for output and analysis on the system concerned.

To get hold of the occurrence we write a ‘choice parameter’ in the handler and this behaves as a constant of the type Exception_Occurrence. We can now write

```
    when Event: others =>
      Put("Unexpected exception: ");
      Put_Line(Exception_Name(Event));
      Put(Exception_Message(Event));
      Clean_Up;
      raise;
  end;
```

The object Event of the type Exception_Occurrence acts as a sort of marker which enables us to identify the current occurrence; its scope is the handler. Such a choice parameter can be placed in any handler.

We can also supply our own message. One way to do this is by an extended form of raise statement in which a string is supplied. Consider

```
declare
   Trouble: exception;
begin
   ...
   raise Trouble with "Doom";
   ...
   raise Trouble with "Gloom";
   ...
exception
   when Event: Trouble =>
      Put(Exception_Message(Event));
end;
```

This extended raise statement raises the exception Trouble with the string attached as the message. The call of Put in the handler will output "Doom" or "Gloom" according to which occurrence of Trouble was raised. Note that our own messages work consistently with the predefined messages; for example Exception_Information will include our message if we have supplied one. A message can also be attached to a raise expression.

Note the difference between

```
raise An_Error;          -- message is implementation-defined
raise An_Error with "";  -- message is null
```

The package Ada.Exceptions contains other types and subprograms which provide further facilities. Its specification is

```
package Ada.Exceptions is
   pragma Preelaborate(Exceptions);
   type Exception_Id is private;
   pragma Preelaborable_Initialization(Exception_Id);
   Null_Id: constant Exception_Id;

   function Exception_Name(Id: Exception_Id) return String;
   ...   -- also Wide_ and Wide_Wide_Exception_Name

   type Exception_Occurrence is limited private;
   pragma Preelaborable_Initialization(Exception_Occurrence);
   type Exception_Occurrence_Access is
                                        access all Exception_Occurrence;
   Null_Occurrence: constant Exception_Occurrence;

   procedure Raise_Exception(E: in Exception_Id;
                             Message: in String := "")
      with No_Return;
   function Exception_Message(X: Exception_Occurrence) return String;
```

```
   procedure Reraise_Occurrence(X: in Exception_Occurrence);

   function Exception_Identity(X: Exception_Occurrence)
                                                  return Exception_Id;
   function Exception_Name(X: Exception_Occurrence) return String;
   ...   -- also Wide_ and Wide_Wide_Exception_Name
   function Exception_Information(X: Exception_Occurrence) return String;

   procedure Save_Occurrence(Target: out Exception_Occurrence;
                             Source: in Exception_Occurrence);
   function Save_Occurrence(Source: Exception_Occurrence)
                                  return Exception_Occurrence_Access;
  ... -- also procedures for streaming occurrences, see Section 23.7
private
   ...
end Ada.Exceptions;
```

The two subprograms Save_Occurrence enable exception occurrences to be saved for detailed later analysis. Note that the type Exception_Occurrence is limited; using subprograms rather than allowing the user to save values through assignment gives better control over the use of storage for saved exception occurrences (which could be large since they may contain extensive trace back information).

The procedure Save_Occurrence copies the occurrence from the Source to the Target. It may truncate the message associated with the occurrence to 200 characters; this corresponds to the minimum size of line length required to be supported as mentioned in Section 5.3. On the other hand, the function Save_Occurrence copies the occurrence to a newly created object and returns an access value to the new object. It is not permitted to truncate the message.

So we might have some debugging package containing an array in which perhaps up to 100 exception occurrences might be stored. A fragment of the body of a crude implementation might be

```
Dump: array (1 .. 100) of Exception_Occurrence;
Dump_Index: Integer := 0;
...
procedure Dump_Ex(E: Exception_Occurrence) is
begin
   Dump_Index := Dump_Index + 1;
   Save_Occurrence(Dump(Dump_Index), E);
end Dump_Ex;

procedure Analyse_Ex is
begin
   Put_Line("Analysis of saved occurrences:");
    for I in 1 .. Dump_Index loop
      Put_Line(Exception_Information(Dump(I)));
   end loop;
   Dump_Index := 0;          -- reset dump
end Analyse_Ex;
```

and then a handler could save an occurrence for later analysis by writing

```
    when Event: others =>
       Dump_Ex(Event);
       Clean_Up;
       raise;
end;
```

Care is clearly needed to ensure that the dumping package does not itself raise exceptions and thereby get the system into a mess.

An occurrence may be reraised by calling the procedure Reraise_Occurrence. This is precisely equivalent to reraising an exception by a raise statement without an exception name and does not create a new occurrence (this ensures that the original cause is not lost). An important advantage of Reraise_Occurrence is that it can be used to reraise an occurrence that was stored by one of the subprograms Save_Occurrence.

Another way of raising an exception with a message is to call the somewhat redundant procedure Raise_Exception. In order to do this we need some way of referring to the exception. However, exceptions are not proper types and cannot be passed as parameters in the normal way. So instead we pass a value of the type Exception_Id which can be thought of as a global enumeration type whose literals represent the individual exceptions. The identity of an exception is provided by the attribute Identity.

So the statement

```
Raise_Exception(Trouble'Identity, "Doom");
```

is identical to

```
raise Trouble with "Doom";
```

The reason for the existence of Raise_Exception is that the form of raise statement with a message did not exist in Ada 95 and so Raise_Exception had to be used instead. The aspect No_Return applies to Raise_Exception; this is explained in the next section.

We mentioned above that the exceptions can be thought of as representing the literals of some type Exception_Id and that the Identity attribute enabled conversion from an exception to its identity. We can sort of go in the reverse direction by calling the function Exception_Name applied to an exception identity; this returns the full dotted name of the exception as a string. So

```
Exception_Name(Error'Identity)
```

might return the string "STACK.ERROR". (This is the same form as the string returned by Expanded_Name applied to tags; see Section 13.4.)

The identity of an exception associated with an occurrence is obtained by calling the function Exception_Identity. If the function Exception_Identity is applied to Null_Occurrence then it returns Null_Id and thereby provides a useful test for Null_Occurrence.

Applying Exception_Name, Exception_Message or Exception_Information to Null_Occurrence raises Constraint_Error. Similarly applying Raise_Exception to Null_Id also raises Constraint_Error.

Note finally that the following are equivalent

```
Exception_Name(Exception_Identity(Event))
Exception_Name(Event)
```

since there are two overloadings of the function Exception_Name.

Exercise 15.4

1 Revise the package Stack of Section 15.2 so that appropriate messages are attached to the raising of the exception Error. Change the handler to output the specific message.

15.5 Exception pragmas and aspects

There are a number of pragmas and aspects associated with exceptions. (Remember from Section 5.6 that certain properties of entities are known as aspects. Sometimes they are set by pragmas and sometimes by aspect specifications.)

First, consider the pragma Assert and the associated pragma Assertion_Policy. These enable a program to check that some condition holds and to raise an exception with a message if it does not. The exception raised is Assertion_Error in the package Ada.Assertions. Thus we might write

```
pragma Assert(Status = Green, "Help");
```

The first parameter of Assert is thus a Boolean expression and the second (and optional) parameter is a string. The parameter of Assertion_Policy is an identifier which controls the behaviour of the pragma Assert. Two policies are defined by the language, namely, Check and Ignore. Further policies may be defined by the implementation. It is also implementation defined as to which policy applies by default so we should always set the policy if we are using assertions.

The specification of the package Ada.Assertions is

```
package Ada.Assertions is
   pragma Pure(Assertions);
   Assertion_Error: exception;
   procedure Assert(Check: in Boolean);
   procedure Assert(Check: in Boolean; Message: in String);
end Ada.Assertions;
```

The pragma Assert can be used wherever a declaration or statement is allowed. Thus it might occur in a list of declarations such as

```
N: constant Integer := ... ;
pragma Assert(N > 1);
```

```
A: Real_Matrix(1 .. N, 1 .. N);
EV: Real_Vector(1 .. N);
```

and in a sequence of statements such as

```
pragma Assert(Transpose(A) = A, "A not symmetric");
EV := Eigenvalues(A);
```

(These examples use types and subprograms from the Numerics annex, see Section 26.5.) If the policy set by Assertion_Policy is Check then the above pragmas are equivalent to

```
if not N > 1 then
   raise Assertion_Error;
end if;
```

and

```
if not Transpose(A) = A then
   raise Assertion_Error with "A not symmetric";
end if;
```

Remember from the previous section that a raise statement without any explicit message is not the same as one with an explicit null message. In the former case a subsequent call of Exception_Message returns implementation defined information whereas in the latter case it returns a null string. This same behaviour thus occurs with the Assert pragma as well – providing no message is not the same as providing a null message.

If the policy set by Assertion_Policy is Ignore then the Assert pragma is ignored at run time – but of course the syntax of the parameters is checked during compilation.

The two procedures Assert in the package Ada.Assertions have an identical effect to the corresponding Assert pragmas except that their behaviour does not depend upon the assertion policy. Thus the call

```
Assert(Some_Test);
```

is always equivalent to

```
if not Some_Test then
   raise Assertion_Error;
end if;
```

In other words we could define the behaviour of

```
pragma Assert(Some_Test);
```

as equivalent to

```
if policy_identifier = Check then
   Assert(Some_Test);                    -- call of procedure Assert
end if;
```

Note again that there are two procedures Assert, one with and one without the message parameter. These correspond to raise statements with and without an explicit message.

The pragma Assertion_Policy is a so-called configuration pragma and controls the behaviour of Assert throughout the units to which it applies. It is thus possible for different policies to be in effect in different parts of a partition. As well as controlling the behaviour of Assert, the pragma Assertion_Policy also controls the contract facilities such as pre- and postconditions described in the next chapter.

Another feature introduced in Ada 2005 with a related flavour is the ability to assert that a subprogram never returns by applying the aspect No_Return. In Ada 2005 this is done by using a pragma but in Ada 2012 this should be done by using an aspect specification since the pragma is now considered to be obsolete.

The aspect No_Return can be applied to a procedure (but not to a function) and asserts that the procedure never returns in the normal sense. Control can leave the procedure only by the propagation of an exception or it might loop forever (which is common among certain real-time programs). Thus we might have a procedure Fatal_Error which outputs some message and then propagates an exception which can be handled in the main subprogram. We can apply the aspect to the procedure specification using an aspect specification thus

```
procedure Fatal_Error(Message: in String)
  with No_Return;
```

or by using the obsolete pragma thus

```
procedure Fatal_Error(Message: in String);
pragma No_Return(Fatal_Error);
```

We might then have

```
procedure Fatal_Error(Message: in String) is
begin
  Put_Line(Message);
  ...                          -- other last wishes
  raise Death;
end Fatal_Error;
...
procedure Main is
  ...
  Put_Line("Program terminated successfully");
exception
  when Death =>
    Put_Line("Program terminated: known error");
  when others =>
    Put_Line("Program terminated: unknown error");
end Main;
```

There are two consequences of supplying the aspect No_Return.

- The implementation checks at compile time that the procedure concerned has no explicit return statements. There is also a check at run time that it does not

attempt to run into the final end – Program_Error is raised if it does as in the case of running into the end of a function.

- The implementation is able to assume that calls of the procedure do not return and so various optimizations can be made.

We might then have a call of Fatal_Error as in

```
function Pop return Integer is
begin
   if Top = 0 then
      Fatal_Error("Stack empty");              -- never returns
   elsif
      Top := Top – 1;
      return S(Top + 1);
   end if;
end Pop;
```

If No_Return applies to Fatal_Error then the compiler should not compile a jump after the call of Fatal_Error and should not produce a warning that control might run into the final end of Pop.

We could restructure the procedure Fatal_Error to raise the exception Death with the message thus

```
procedure Fatal_Error(Message: in String)
   with No_Return is
begin
   ...                              -- other last wishes
   raise Death with Message;
end Fatal_Error;
```

The exception handler for Death in the main subprogram can now use Exception_Message to print out the message.

If a subprogram has no distinct specification then the aspect specification is placed inside the body (as shown above). Note carefully that it goes before the reserved word **is**.

If several subprograms have the aspect No_Return, then using aspect specifications each must have its own. However, a pragma No_Return could apply to several subprograms declared in the same package specification. For further details of the rules regarding aspect specifications see Section 16.1.

It is vital that dispatching works correctly with procedures that do not return. A non-returning dispatching procedure can only be be overridden by a non-returning procedure and so the overriding one must also have the aspect No_Return thus

```
type T is tagged ...
procedure P(X: T; ... )
   with No_Return;
...
type TT is new T with ...
overriding
procedure P(X: TT; ... )
   with No_Return;
```

The reverse is not true, a procedure that does return can be overridden by one that does not. It is possible to give the aspect No_Return for an abstract procedure, but obviously not for a null procedure. (The aspect No_Return can also be given for a generic procedure. It then applies to all instances. See Chapter 19.)

There are also two pragmas Suppress and Unsuppress for controlling checks. Essentially, Suppress says that specific run-time checks which give rise to exceptions such as Constraint_Error can be turned off whereas Unsuppress says they must not be turned off. Thus Suppress is merely a recommendation whereas Unsuppress is an order. One reason why Suppress is merely a recommendation is that some checks may be done by hardware. Moreover, there is no guarantee that a suppressed exception will not be raised. Indeed it could be propagated from another unit compiled with checks.

The core language checks corresponding to the exception Constraint_Error are Access_Check (checking that an access value is not null), Discriminant_Check (checking that a discriminant value is consistent with the component being accessed or a constraint), Division_Check (checking the second operand of **/**, **rem** and **mod**), Index_Check (checking that an index is in range), Length_Check (checking that the number of components of an array match), Overflow_Check (checking for numeric overflow), Range_Check (checking that various constraints are satisfied) and Tag_Check (checking that tags are equal when dispatching). There are other checks relating to the specialized annexes.

The core language checks corresponding to Program_Error are Allocation_Check (checking an obscure situation regarding allocation), Elaboration_Check (checking that the body of a unit has been elaborated) and Accessibility_Check (checking an accessibility level).

The check corresponding to Storage_Error is Storage_Check (checking that space for a storage pool or task has not been exceeded).

The pragma takes the form

```
pragma Suppress(Range_Check);
```

We can also write **pragma** Suppress(All_Checks); which does the obvious thing.

The purpose of Unsuppress is to ensure that checks are performed so that we can rely upon an exception being raised. Suppose we have a type Sat_Int that we wish to behave as a saturated type. This means it never overflows but just takes the maximum or minimum value when appropriate. We could write the multiplication operator for such a type as

```
function "*" (Left, Right: Sat_Int) return Sat_Int is
   pragma Unsuppress(Overflow_Check);
begin
   return Integer(Left) * Integer(Right);
exception
   when Constraint_Error =>
      if (Left > 0 and Right > 0) or (Left < 0 and Right < 0) then
         return Sat_Int'Last;
      else
         return Sat_Int'First;
      end if;
end "*";
```

The pragma Unsuppress ensures that the code always works as intended even if checks are suppressed in the program as a whole.

We conclude with an important warning regarding Suppress. If we suppress a check and the situation nevertheless arises then the program is erroneous.

Exercise 15.5

1 Write a pragma Assert to check that an array A is sorted into ascending order. Use a quantified expression.

15.6 Scope of exceptions

To a large extent exceptions follow the same scope rules as other entities. An exception can hide and be hidden by another declaration; it can be referred to by the dotted notation and so on. An exception can be renamed

```
Help: exception renames Bank.Alarm;
```

Exceptions are, however, different in many ways. We cannot declare arrays of exceptions, and they cannot be components of records, parameters of subprograms and so on. In short, exceptions are not objects and so cannot be manipulated. They behave a bit like literals of some globally defined enumeration type Exception_Id as mentioned in Section 15.4.

A very important characteristic of exceptions is that they are not created dynamically as a program executes but should be thought of as existing throughout the life of the program. Indeed the full set of exceptions in a program is only known when the program is bound and so the representation of the type Exception_Id can be thought of as occurring at bind time. This aspect of exceptions relates to the way in which they are propagated dynamically up the chain of execution rather than statically up the chain of scope. An exception can be propagated outside its scope although of course it can then only be handled anonymously by **others**. This is illustrated by the following

```
procedure Main is
   procedure P is
      X: exception;
   begin
      raise X;
   end P;
begin
   P;
exception
   when others =>
         ...                    -- X handled here
end Main;
```

The procedure P declares and raises the exception X but does not handle it. When we call P, the exception X is propagated to the block calling P where it is handled anonymously.

We could of course rediscover the textual form of the exception by writing

```
when Event: others =>
```

and then Exception_Name(Event) would return the string "MAIN.P.X".

It is even possible to propagate an exception out of its scope, where it becomes anonymous, and then back in again where it can once more be handled by its proper name. There is an example on the web.

A further illustration of the nature of exceptions is afforded by a recursive procedure containing an exception declaration. Unlike variables declared in a procedure, we do not get a new exception for each recursive call. Each recursive activation refers to the same exception – they have the same Exception_Id. Consider the following example

```
procedure F(N: Integer) is
   X: exception;
begin
   if N = 0 then
      raise X;
   else
      F(N – 1);
   end if;
exception
   when X =>
      Put("Got it!");
      raise;
   when others =>
      null;
end F;
```

Suppose we execute F(4); we get recursive calls F(3), F(2), F(1) and finally F(0). When F is called with parameter zero, it raises the exception X, handles it, prints out a confirmatory message and then reraises it. The calling instance of F (which itself had N = 1) receives the exception and again handles it as X and so on. The message is therefore printed out five times in all and the exception is finally propagated anonymously. Observe that if each recursive activation had created a different exception then the message would only be printed out once.

In most of the examples we have seen so far exceptions have been raised in statements. An exception can however also be raised in a declaration. Thus

```
N: Positive := 0;
```

would raise Constraint_Error because the initial value of N does not satisfy the range constraint 1 .. Integer'Last of the subtype Positive. An exception raised in a declaration is not handled by a handler (if any) of the unit containing the declaration but is immediately propagated up a level. This means that in any handler we are assured that all declarations of the unit were successfully elaborated and so there is no risk of referring to something that does not exist.

Finally, a warning regarding parameters of mode **out** or **in out**. If a subprogram is terminated by an exception then any actual parameter of an elementary type will

not have been updated since such updating occurs on a normal return. On the other hand, a parameter of a tagged record type or an explicitly limited type is always passed by reference and so will always have been updated. For an array or other record type the parameter mechanism is not so closely specified and the actual parameter may or may not have its original value. A program assuming a particular mechanism may have a bounded error. As an example consider the procedure Withdraw of the package Bank in Exercise 12.6(**1**) and suppose that the type key were not explicitly limited. It would be incorrect to attempt to take the key away and raise an alarm as in

```
procedure Withdraw (K: in out Key; M: in out Money) is
begin
   if Valid (K) then
      if M > amount remaining then
         M := amount remaining;
         Free(K.Code) := True;                -- close account
         K.Code := 0;
         raise Alarm;
      else
         ...
      end if;
   end if;
end Withdraw;
```

If the parameter mechanism were implemented by copy then the bank would think that the key were now free but would have left the greedy customer with a copy. However, if the key were explicitly limited or a tagged type as in Exercise 14.7(**2**) then this problem would not arise since such types are always passed by reference.

Exercise 15.6

1 Rewrite the package Bank of Exercise 12.6(**1**) to declare an exception Alarm and raise it when any illegal banking activity is attempted.

2 Consider the following pathological procedure

```
procedure P is
begin
   P;
exception
   when Storage_Error =>
      P;
end P;
```

What happens when P is called? To be explicit suppose that there is enough stack space for only N simultaneous recursive calls of P but that on the N+1th call the exception Storage_Error is raised. How many times will P be called in all and what eventually happens?

Checklist 15

Do not use exceptions unnecessarily.

Use specific user declared exceptions rather than predefined exceptions where relevant.

Ensure that handlers return resources correctly.

Match the constraints on index variables to the arrays concerned.

Beware of uninitialized variables.

Out and in out parameters may not be updated correctly if a subprogram is terminated by an exception.

Numeric_Error is obsolescent.

The ability to add a message to raise statements was added in Ada 2005.

The pragmas Assert, Assertion_Policy, No_Return and Unsuppress were added in Ada 2005.

New in Ada 2012

Raise expressions are new in Ada 2012 (actually added as a Binding Interpretation by AI12-22).

The pragma No_Return is obsolete in Ada 2012 and replaced by the corresponding aspect specification.

16 Contracts

This chapter describes the mechanisms for defining and enforcing contracts introduced in Ada 2012. The general idea is that we add aspects defining certain requirements on the behaviour of an entity such as a subprogram and then, when the subprogram is executed, if any of these requirements are not met, an exception is raised.

As well as specifying pre- and postconditions on subprograms, we can also specify invariant properties of private types, and additional restrictions on the values of subtypes. These properties are all given by the use of aspect specifications briefly mentioned in Section 5.6.

Indeed, it was the introduction of pre- and postconditions that triggered the introduction of aspect specifications into Ada 2012. Moreover, as mentioned in Chapter 9, these in turn triggered the introduction of conditional and quantified expressions.

We start by giving further details of the rules regarding aspect specifications.

16.1 Aspect specifications

As we saw in Section 15.5 we can apply an aspect such as No_Return to a procedure using an aspect specification. We can supply several aspects by a series of aspect specifications as follows

```
procedure Do_It( ... )
   with Inline, No_Return;
```

In the general case an aspect specification supplies an expression for the aspect following =>. In the case of both Inline and No_Return, these aspects take a Boolean value so we could pedantically write

```
procedure Do_It ( ... )
   with No_Return => True,
        Inline => True;
```

There is a general rule that if an aspect requires a Boolean value then it can be omitted and by default is taken to be true. But this does not apply to the aspects Default_Value and Default_Component_Value, see Sections 6.8 and 8.2.

As we see, aspect specifications are introduced by the reserved word **with** and if there are several they are separated by commas. The word **with** is used for a number of purposes (we have already met with clauses and type extensions) and this might be considered confusing. However, with a sensible layout there should be no problems.

In the case of a subprogram without a distinct specification, the aspect specification goes in the subprogram body before **is** thus

```
procedure Do_It( ... )
   with Inline is
   ...
begin
   ...
end Do_It;
```

This arrangement is because the aspect specification is very much part of the specification of the subprogram. If a subprogram has a distinct specification then we cannot give a language-defined aspect specification on the body; this avoids problems of conformance.

If there is a stub but no specification then any aspect specification goes on the stub but not the body. Thus aspect specifications go on the first of specification, stub, and body but are never repeated. Note also that we can give aspect specifications on other forms of stubs and bodies such as package bodies (also task and entry bodies, see Chapter 20) but none are defined by the language.

In the case of a stub, abstract subprogram, or null subprogram which never have bodies, the aspect specification goes after **is separate**, **is abstract**, or **is null** thus

```
procedure Action(D: in Data) is separate
   with Convention => C;

procedure Enqueue( ... ) is abstract
   with Synchronization => By_Entry;

procedure Nothing is null
  with Something;
```

The aspect Convention is described in Section 25.5; the example of the use of Synchronization is from the package Synchronized_Queue_Interfaces, a child of Ada.Containers, see Section 24.8.

The same style is followed by expression functions described in Section 10.1 thus

```
function Inc (A: Integer) return Integer is (A + 1)
   with Inline;
```

As we have seen, many aspects (such as Inline) can now be set by aspect specifications in Ada 2012 and corresponding pragmas are obsolete. Others are preferred to be set by pragmas still such as Pure and Preelaborate, some can only be set by pragmas such as Preelaborable_Initialization described in Section 13.8.

Many aspects such as Inline apply to a particular entity. However, there are some pragmas that do not relate to any particular entity and so for which an alternative aspect specification would be impossible. These include Assert and Assertion_Policy, Suppress and Unsuppress, and Restrictions.

Other properties of entities such as details of their representation can often be given by aspect specifications; these are described in Section 25.1.

In deciding between pragmas and aspect specifications, remember that an aspect specification such as Inline has to be given with the subprogram specification to which it relates; this is an advantage since all the properties of an entity can be given together and this avoids scattering them around as can happen with pragmas.

Moreover, since a corresponding pragma is split from the entity, not only might it occur much later in the text but it has to mention the entity to which it relates. A curious problem with pragmas is that because of overloading, we are not able to distinguish between two procedures Do_It with different profiles and so writing

```
pragma Inline(Do_It);
```

will apply to all preceding procedures Do_It which might be a nuisance.

There is, however, one consequence of the introduction of aspect specifications and this is that the rule of linear elaboration had to be revised. So, within an aspect specification for A we can refer forward to an entity B which has not yet been declared provided B has been declared by the time A has to be frozen. The concept of freezing is outlined in Section 25.1.

In the remainder of this chapter we will meet many aspects, namely

```
Pre => Boolean expression
Pre'Class =>
Post =>
Post'Class =>
Type_Invariant =>
Type_Invariant'Class =>
Static_Predicate =>
Dynamic_Predicate =>
```

In every case, the expected type of the aspect is Boolean (that is of the predefined type Boolean or derived from it). As mentioned in Section 9.4 when discussing quantified expressions, we often refer to such expressions as predicates which is simply a mathematical term for a truth value. (And nothing to do with human linguistics where we say that in the sentence ‘I love Ada’, the subject is ‘I’ and the predicate is ‘love Ada’.)

Exercise 16.1

1 Declare a type My_Boolean derived from the predefined type Boolean with a default value of True.

16.2 Preconditions and postconditions

Preconditions and postconditions are certainly the most important of the contract features introduced in Ada 2012 and perhaps are the most important new features in Ada 2012 overall.

A precondition is given by the aspect Pre. Thus to give a precondition for the procedure Pop in a package Stacks we might augment its specification thus

```
procedure Pop(S: in out Stack; X: out Integer)
  with Pre => not Is_Empty(S);
```

where Is_Empty is a function also in the package Stacks.

In a similar way we might give a postcondition as well which might be that the stack is not full. So altogether the specification of the package Stacks in Section 12.4 might become

```
package Stacks is
  type Stack is private;

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

  procedure Push(S: in out Stack; X: in Integer)
    with
      Pre => not Is_Full(S),
      Post => not Is_Empty(S);

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

  function "=" (S, T; Stack) return Boolean;

private
  ...
end Stacks;
```

Note how the individual aspects Pre and Post are separated by commas. The final semicolon is of course the semicolon at the end of the subprogram declaration as a whole. Note also that the value of the parameter S of the functions Is_Empty or Is_Full in the pre- and postcondition is the value of the parameter S of the procedures Push and Pop when they are called. This is important regarding the postcondition; in the case of Push, the value of S when Is_Empty is called is the value after the Push operation has been performed.

Pre- and postconditions are controlled by the same mechanism as assertions using the pragma Assert. It will be recalled from Section 15.5 that these can be switched on and off by the pragma Assertion_Policy. Thus if we write

```
pragma Assertion_Policy(Check);
```

then assertions are enabled whereas if the parameter of the pragma is Ignore then all assertions are ignored.

In the case of a precondition, whenever a subprogram with a precondition is called, if the policy is Check then the precondition is evaluated and if it is false then Assertion_Error is raised and the subprogram is not entered. Similarly, on return from a subprogram with a postcondition, if the policy is Check then the postcondition is evaluated and if it is false then Assertion_Error is raised.

So if the policy is Check and Pop is called when the stack is empty then Assertion_Error is raised whereas if the policy is Ignore then the predefined exception Constraint_Error would probably be raised (depending upon how the stack had been implemented).

If we wish to raise a different exception to Assertion_Error or include an exception message or both then this can be done as explained in Section 16.5.

Note that it is not permitted to give the aspects Pre or Post for a null procedure; this is because all null procedures are meant to be interchangeable.

There are also aspects Pre'Class and Post'Class for use with tagged types (and they can be given with null procedures). The subtle topic of multiple inheritance of pre- and postconditions will be discussed in a moment.

Now suppose we apply a specific precondition Before and/or a specific postcondition After to a procedure P by writing

```
procedure P(P1: in T1; P2: in out T2; P3: out T3)
   with Pre => Before,
        Post => After;
```

where Before and After are Boolean expressions.

The precondition Before and the postcondition After can involve the parameters P1 and P2 and P3 and any visible entities such as other variables, constants and functions. Note that Before can involve an **out** parameter such as P3 (inasmuch as one can always read any constraints on an **out** parameter such as the bounds if it were an array).

The attribute X'Old will be found useful in postconditions; it denotes the value of X on entry to P. It is typically applied to parameters of mode **in out** such as P2 but it can be applied to any visible entity such as a global variable. This can be useful for monitoring global variables which are updated by the call of P. But note that 'Old can only be used in postconditions and not in arbitrary text and it cannot be applied to objects of a limited type.

Perhaps surprisingly, 'Old can also be applied to parameters of mode **out**. For example, in the case of a parameter of a record type that is updated as a whole, nevertheless we might want to check that a particular component has not changed. Thus in updating some personal details, such as address and occupation, we might ensure that the person's date of birth and sex are not tampered with by writing

```
Post => P.Sex = P.Sex'Old and P.Dob = P.Dob'Old
```

In the case of an array, we can write A(J)'Old which means the original value of A(J). But A(J'Old) is different since it is the component of the final value of A but indexed by the old value of J. If J is not modified then it is better to say A(J)'Old rather than A'Old(J) which would imply remembering the whole array.

Remember that the result of a function is an object and so 'Old can be applied to it. Note carefully the difference between F(X)'Old and F(X'Old). The former applies F to X on entry to the subprogram and saves it. The latter saves X and applies

F to it when the postcondition is evaluated. These could be different because the function F might also involve global variables which have changed.

Generally, 'Old can be applied to anything but there are restrictions on its use in certain conditional structures in which it can only be applied to statically determined objects. This is illustrated by the following

```
Table: array (1 .. 10) of Integer := ... ;
procedure P(K: in out Natural)
   with Post => K > 0 and then Table(K)'Old = 1;    -- illegal
```

The programmer's intent is that the postcondition uses a short circuit form to avoid evaluating Table(K) if K is not positive on exit from the procedure. But, 'Old is evaluated and stored on entry and this could raise Constraint_Error because K might for example be zero. This is a conundrum since the compiler cannot know whether the value of Table(K) will be needed and also K can change so it cannot know which K anyway. So such structures are forbidden.

In the case of a postcondition applying to a function F, the result of the function is denoted by the attribute F'Result. Again this attribute can only be used in post-conditions.

Some trivial examples of declarations of a procedure Pinc and a function Finc to perform an increment are

```
procedure Pinc(X: in out Integer)
   with Post => X = X'Old+1;

function Finc(X: Integer) return Integer
   with Post => Finc'Result = X'Old+1;
```

With regard to Assertion_Policy, it is the value in effect at the point of the subprogram declaration that matters. So we cannot have a situation where the policy changes during the call so that preconditions are switched on but postconditions are off or vice versa.

And so the overall effect of calling P with checks enabled is roughly that, after evaluating any parameters at the point of call, it as if the body were

```
if not Before then                          -- check precondition
   raise Assertion_Error;
end if;

... evaluate and store any 'Old stuff;

... call actual body of P;

if not After then                           -- check postcondition
   raise Assertion_Error;
end if;

... copy back any by-copy parameters;

... return to point of call;
```

Occurrences of Assertion_Error are propagated and so raised at the point of call; they cannot be handled inside P. Of course, if the evaluation of Before or After themselves raise some exception then that will be propagated to the point of call.

Table 16.1 Obligations and assumptions

	Pre	Post
Call writer	obligation	assumption
Body writer	assumption	obligation

Note that conditions Pre and Post can also be applied to entries; see Section 20.2.

Before progressing to the problems of inheritance it is worth reconsidering the purpose of preconditions and postconditions.

- A precondition Before is an obligation on the caller to ensure that it is true before the subprogram is called and so is an assumption that the implementer of the body can rely upon on entry to the body.
- A postcondition After is an obligation on the implementer of the body to ensure that it is true on return from the subprogram and so is an assumption that the caller can rely upon on return from the body.

The symmetry is neatly illustrated by Table 16.1 above.

The simplest form of inheritance occurs with derived types that are not tagged. Suppose we declare the procedure Pinc as above with the postcondition shown and supply a body

```
procedure Pinc(X: in out Integer) is
begin
   X := X+1;
end Pinc;
```

and then declare a type

```
type Apples is new Integer;
```

then the procedure Pinc is inherited by the type Apples. So if we then write

```
No_Of_Apples: Apples;
...
Pinc(No_Of_Apples);
```

what actually happens is that the code of the procedure Pinc originally written for Integer is called and so the postcondition is inherited automatically.

If the user now wants to add a precondition to Pinc that the number of apples is not negative then a completely new subprogram has to be declared which overrides the old one thus

```
procedure Pinc(X: in out Apples)
   with Pre => X >= 0,
        Post => X = X'Old+1;
```

and a new body has to be supplied (which will of course in this curious case be essentially the same as the old one). So we cannot inherit an operation and change its conditions at the same time.

We now turn to tagged types and first continue to consider the specific conditions Pre and Post. As a familiar example, consider the hierarchy consisting of a type Object and then direct descendants Circle, Square and Triangle.

Suppose the type Object is

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

and we declare a function Area thus

```
function Area(O: Object) return Float
   with Pre => O.X_Coord > 0.0,
        Post => Area'Result = 0.0;
```

This imposes a requirement on the caller that the function is called only with objects with positive x-coordinate (for some obscure reason), and a requirement on the implementer of the body that the area is zero (raw objects are just points and have no area).

If we now declare a type Circle as

```
type Circle is new Object with
   record
      Radius: Float;
   end record;
```

and override the inherited function Area then the Pre and Post conditions on Area for Object are not inherited and we have to supply new ones, perhaps

```
function Area(C: Circle)
   with Pre => C.X_Coord – C.Radius > 0.0,
        Post => Area'Result > 3.1 * C.Radius**2 and
                Area'Result < 3.2 * C.Radius**2;
```

The conditions ensure that all of the circle is in the right half-plane and that the area is about right!

So the rules so far are exactly as for the untagged case. If an operation is not overridden then it inherits the conditions from its ancestor but if it is overridden then those conditions are lost and new ones have to be supplied. And if no new ones are supplied then they are by default taken to be True.

In conclusion, the conditions Pre and Post are very much part of the actual body. One consequence of this is that an abstract subprogram cannot have Pre and Post conditions because an abstract subprogram has no body.

We now turn to the class wide conditions Pre'Class and Post'Class which are subtly different. The first point is that the class wide ones apply also to all descendants even if the operations are overridden. In the case of Post'Class if an

overridden operation has no condition given then it is taken to be True (as in the case of Post). But in the case of Pre'Class, if an overridden operation has no condition given then it is only taken to be True if no other Pre'Class applies (no other is inherited). We will now look at the consequences of these rules.

It might be that we want certain conditions to hold throughout the hierarchy, perhaps that all objects concerned have a positive x-coordinate and nonnegative area. In that case we can use class wide conditions.

```
function Area(O: Object) return Float
  with Pre'Class => O.X_Coord > 0.0,
       Post'Class => Area'Result >= 0.0;
```

Now when we declare Area for Circle, the aspects Pre'Class and Post'Class from Object will be inherited by the function Area for Circle. Note that within a class wide condition a formal parameter of type T is interpreted as of T'Class. Thus O is of type Object'Class and thus applies to Circle. The inherited postcondition is simply that the area is not negative and uses the attribute 'Result.

If we do not supply conditions for the overriding Area for Circle and simply write

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

then the precondition inherited from Object still applies. In the case of the postcondition not only is the postcondition from Object inherited but there is also an implicit postcondition of True. So the applicable conditions for Area for Circle are

Pre'Class for Object

Post'Class for Object
True

Suppose on the other hand that we give explicit Pre'Class and Post'Class for Area for Circle thus

```
overriding
function Area(C: Circle) return Float
  with Pre'Class => ... ,
       Post'Class => ... ;
```

We then find that the applicable conditions for Area for Circle are

Pre'Class for Object
Pre'Class for Circle

Post'Class for Object
Post'Class for Circle

Incidentally, it makes a lot of sense to declare the type Object as abstract so that we cannot declare pointless objects. In that case Area might as well be abstract as well. Although we cannot give conditions Pre and Post for an abstract operation we can still give the class wide conditions Pre'Class and Post'Class.

If the hierarchy extends further, perhaps Equilateral_Triangle is derived from Triangle which itself is derived from Object, then we could add class wide conditions to Area for Triangle and these would also apply to Area for Equilateral_Triangle. And we might add specific conditions for Equilateral_Triangle as well. So we would then find that the following apply to Area for Equilateral_Triangle

```
Pre'Class for Object
Pre'Class for Triangle
Pre for Equilateral Triangle

Post'Class for Object
Post'Class for Triangle
Post for Equilateral_Triangle
```

The postconditions are quite straightforward, all apply and all must be True on return from the function Area. The compiler can see all these postconditions when the code for Area is compiled and so they are all checked in the body. Note that any default True makes no difference because B **and** True is the same as B.

However, the rules regarding preconditions are perhaps surprising. The specific precondition Pre for Equilateral_Triangle must be True (checked in the body) but so long as just one of the class wide preconditions Pre'Class for Object and Triangle is true then all is well. Note that class wide preconditions are checked at the point of call. Do not get confused over the use of the word apply. They all apply but only the ones seen at the point of call are actually checked.

The reason for this state of affairs concerns dispatching and especially redispatching. Consider the case of Ada airlines in Section 14.1 which has Basic, Nice and Posh passengers. Remember that Basic passengers just get a seat. Nice passengers also get a meal and Posh passengers also get a limo. The types Reservation, Nice_Reservation and Posh_Reservation form a hierarchy with Nice_Reservation being extended from Reservation and so on. The facilities are assigned when a reservation is made by calling an appropriate procedure Make thus

```
procedure Make(R: in out Reservation) is
begin
   Select_Seat(R);
end Make;

procedure Make(NR: in out Nice_Reservation) is
begin
   Make(Reservation(NR));
   Order_Meal(NR);
end Make;

procedure Make(PR: in out Posh_Reservation) is
   Make(Nice_Reservation(PR));
   Arrange_Limo(PR);
end Make;
```

Each Make calls its ancestor in order to avoid duplication of code and to ease maintenance.

A variation involving redispatching (see Section 14.5) introduces two different procedures Order_Meal, one for Nice passengers and one for Posh passengers. We then need to ensure that Posh passengers get a posh meal rather than a nice meal. We write

```
procedure Make(NR: in out Nice_Reservation) is
begin
   Make(Reservation(NR));
                              -- now redispatch to appropriate Order_Meal
   Order_Meal(Nice_Reservation'Class(NR));
end Make;
```

Now suppose we have a precondition Pre'Class on Order_Meal for Nice passengers and one on Order_Meal for Posh passengers. The call of Order_Meal sees that it is for Nice_Reservation'Class and so the code includes a test of Pre'Class on Nice_Reservation. It does not necessarily know of the existence of the type Posh_Reservation and cannot check Pre'Class on that Order_Meal. At a later date we might add Supersonic passengers and this can be done without recompiling the rest of the system so it certainly cannot do anything about checking Pre'Class on Order_Meal for Supersonic_Reservation which does not exist when the call is compiled. So when we eventually get to the body of one of the procedures Order_Meal all we know is that some Pre'Class on Order_Meal has been checked somewhere. And that is all that the writer of the code of Order_Meal can rely upon. Note that nowhere does the compiled code actually 'or' a lot of preconditions together.

In summary, class wide preconditions are checked at the point of call. Class wide postconditions and both specific preconditions and postconditions are checked in the actual body.

A small point to remember is that a class wide operation such as

```
procedure Do_It(X: in out T'Class);
```

is not a primitive operation of T and so although we can specify Pre and Post for Do_It we cannot specify Pre'Class and Post'Class for Do_It.

We noted above that the aspects Pre and Post cannot be specified for an abstract subprogram because it doesn't have a body. They cannot be given for a null procedure either, since we want all null procedures to be identical and do nothing and that includes no conditions.

We now turn to the question of multiple inheritance and progenitors and the so-called Liskov Substitution Principle (LSP). The usual consequence of LSP is that in the case of preconditions they are combined with 'or' (thus weakening) and the rule for postconditions is that they are combined with 'and' (thus strengthening). But the important thing is that a relevant concrete operation can be substituted for the corresponding operations of all its relevant ancestors.

In Ada, a type T can have one parent and several progenitors as described in Section 14.8. Thus we might have

```
type T is new P and G1 and G2 with ...
```

where P is the parent and G1 and G2 are progenitors. Remember that a progenitor cannot have components and cannot have concrete operations (apart possibly for

null procedures). So the operations of the progenitors have to be abstract or null and cannot have Pre and Post conditions. However, they can have Pre'Class and Post'Class conditions. It is possible that the same operation Op is primitive for more than one of these.

Thus the progenitors G1 and G2 might both have an operation Op thus

```
procedure Op(X: in G1) is abstract;
procedure Op(X: in G2) is abstract;
```

If they are conforming (as they are in this case) then the one concrete operation Op of the type T derived from both G1 and G2 will implement both of these. (If they don't conform then they are simply overloadings and two operations of T are required.) Hence the one Op for T can be substituted for the Op of both G1 and G2 and LSP is satisfied.

Now suppose both abstract operations have pre- and postconditions. Take postconditions first, we might have

```
procedure Op(X: in G1) is abstract
   with Post'Class => After1;

procedure Op(X: in G2) is abstract
   with Post'Class => After2;
```

Users of the Op of G1 will expect the postcondition After1 to be satisfied by any implementation of that Op. So if using the Op of T which implements the abstract Op of G1, it follows that Op of T must satisfy the postcondition After1. By a similar argument regarding G2, it must also satisfy the postcondition After2.

It thus follows that the effective postcondition on the concrete Op of T is as if we had written

```
procedure Op(X: in T)
   with Post'Class => After1 and After2;
```

But of course we don't actually have to write that since we simply write

```
overriding
procedure OP(X: in T);
```

and it automatically inherits both postconditions and the compiler inserts the appropriate code in the body. Remember that if we don't give a condition then it is True by default but 'anding' in True makes no difference.

If we do provide another postcondition thus

```
overriding
procedure OP(X: in T)
   with Post'Class => After_T;
```

then the overall class wide postcondition to be checked before returning will be After1 **and** After2 **and** After_T.

Now consider preconditions. Suppose the declarations of the two versions of Op are

```
procedure Op(X: in G1) is abstract
   with Pre'Class => Before1;

procedure Op(X: in G2) is abstract
   with Pre'Class => Before2;
```

Assuming that there is no corresponding Op for P, we must provide a concrete operation for T thus

```
overriding
procedure Op(X: in T)
   with Pre'Class => Before_T;
```

This means that at a point of call of Op the precondition to be checked is Before_T **or** Before1 **or** Before2. As long as this is satisfied it does not matter that Before1 and Before2 might have been different.

If we do not provide an explicit Pre'Class then the condition to be checked at the point of call is Before1 **or** Before2.

An interesting case arises if a progenitor (say G1) and the parent have a conforming operation. Thus suppose P itself has the operation

```
procedure Op(X: in P);
```

and moreover that the operation is not abstract. Then (ignoring preconditions for the moment) this Op for P is inherited by T and thus provides a satisfactory implementation of Op for G1 and all is well.

Now suppose that Op for P has a precondition thus

```
procedure OP(X: in P)
   with Pre'Class => Before_P;
```

and that Before_P and Before1 are not the same. If we do not provide an explicit overriding for Op, it would be possible to call the body of Op for P when the precondition it knows about, Before_P, is False (since Before1 being True would be sufficient to allow the call to proceed). This would effectively mean that no class wide preconditions could be trusted within the subprogram body and that would be totally unacceptable. So in this case there is a special rule that an explicit overriding is required for Op for T.

If Op for P is abstract then a concrete Op for T must be provided and the situation is just as in the case for the Op for G1 and G2.

If T itself is declared as abstract (and P is not abstract and Op for P is concrete) then the inherited Op for T is abstract.

(These rules are similar to those for functions returning a tagged type, see Section 14.3, when the type is extended; it has to be overridden unless the type is abstract in which case the inherited operation is abstract.)

We finish this somewhat mechanical discussion of the rules by pointing out that if silly inappropriate preconditions are given then we will get a silly program.

At the end of the day, the real point is that programmers should not write preconditions that are not sensible and sensibly related to each other. Because of the generality, the compiler cannot tell so stupid things are hard to prohibit. There is no defence against stupid programmers.

A concrete example using simple numbers might help. Suppose we have a tagged type T1 and an operation Solve which takes a parameter of type T1 and perhaps finds the solution to an equation defined by the components of T1. Solve delivers the answer in an **out** parameter A with a parameter D giving the number of significant digits required in the answer. Also we impose a precondition on the number of digits D thus

```
type T1 is tagged record ...

procedure Solve(X: in T1; A: out Float; D: in Integer)
   with Pre'Class => D < 5;
```

The intent here is that the version of Solve for the type T1 always works if the number of significant digits asked for is less than 5.

Now suppose we declare a type T2 derived from T1 and that we override the inherited Solve with a new (better) version that works if the number of significant digits asked for is less than 10 thus

```
type T2 is new T1 with ...

overriding
procedure Solve(X: in T2; A: out Float; D: in Integer)
   with Pre'Class => D < 10;
```

And so on with a type T3

```
type T3 is new T2 with ...

overriding
procedure Solve(X: in T3; A: out Float; D: in Integer)
   with Pre'Class => D < 15;
```

Thus we have a hierarchy of algorithms Solve with increasing capability. Now suppose we have a dispatching call

```
An_X: T1'Class := ... ;
Solve(An_X, Answer, Digs);
```

This will dispatch to one of the Solve procedures but we do not know which one. The only precondition that applies is that on the Solve for T1 which is D < 5. That is fine because D < 5 implies D < 10 and D < 15 and so on. Thus the preconditions work because the hierarchy weakens them.

Similarly, if we have

```
An_X: T2'Class := ... ;
Solve(An_X, Answer, Digs);
```

then it will dispatch to a Solve for one of T2, T3, ..., but not to the Solve for T1. The applicable preconditions are D < 5 and D < 10 and these are notionally 'ored' together which means D < 10 is actually required. To see this suppose we supply D = Digs = 7. Then D < 5 is False but D < 10 is True so by 'oring' False and True we get True, so the call works.

On the other hand if we write

```
An_X: T2 := ... ;
Solve(An_X, Answer, Digs);
```

then no dispatching is involved and the Solve for T2 is called. But both class wide preconditions D < 5 and D < 10 apply and so again the resulting 'ored' precondition that is required is D < 10.

Now it should be clear that if the preconditions do not form a weakening hierarchy then we will be in trouble. Thus if the preconditions were D < 15 for T1, D < 10 for T2, and D < 5 for T3, then dispatching from the root will only check D < 15. However, we could end up calling the Solve for T2 which expects the precondition D < 10 and this might not be satisfied.

Care is thus needed with preconditions that they are sensibly related.

Exercise 16.2

1 The type Square is derived from Object. Declare a function Area with some appropriate pre- and postconditions.

16.3 Type invariants

There are two other facilities of a contractual nature concerning types and subtypes. One is known as type invariants and describe properties of a type that remain true and can be relied upon; this topic is described in this section. The other is known as subtype predicates which extend the idea of constraints and are described in the next section. The distinction can be confusing at first sight and the following thoughts might be helpful.

Type invariants are not like constraints since invariants apply to all values of a type, whereas constraints are used to identify just a subset of the values of a type. Invariants are only meaningful on private types; and only apply to the external view of the type as seen by the user; within the package (that is using the full view) the invariant need not be satisfied as we shall in the examples below.

In some ways, an invariant is more like the range of values specified when declaring a new integer type, as opposed to the constraint specified when defining an integer subtype. The specified range of an integer type can be violated in the middle of an arithmetic computation, but must be satisfied by the time the value is stored back into an object of the type.

Type invariants are useful if we want to ensure that some relationship between the components of a private type always holds. Thus suppose we have a stack and wish to ensure that no value is placed on the stack equal to an existing value on the stack. We can modify the earlier example to

```
package Stacks is

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

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

procedure Push(S: in out Stack; X: in Integer)
   with
      Pre => not Is_Full(S),
      Post => not Is_Empty(S);

-- and so on
```

The function Is_Unduplicated then has to be written to check that all values of the stack are different.

Note that we have mentioned Is_Unduplicated in the type invariant before its specification. This violates the usual 'linear order of elaboration'. However, as mentioned in Section 16.1, there is a general rule that all aspect specifications are only elaborated when the entity they refer to is frozen. Recall that one of the reasons for the introduction of aspect specifications was to overcome this problem with the existing mechanisms which caused information to become separated from the entities to which it relates.

The invariant on a private type T is checked when the value can be changed from the point of view of the outside user. In the case of the stack, the invariant Is_Unduplicated will be checked when we declare a new object of type Stack and each time we call Push and Pop.

Note that any subprograms internal to the package and not visible to the user can do what they like. It is only when a value of the type Stack emerges into the outside world that the invariant is checked.

The type invariant could be given on the full type in the private part rather than on the visible declaration of the private type (but not on both). Thus the user need not know that an invariant applies to the type.

Type invariants, like pre- and postconditions, are controlled by the pragma Assertion_Policy and only checked if the policy is Check. If an invariant fails to be true then Assertion_Error is raised at the appropriate point.

Like pre- and postconditions there are both specific invariants that can be applied to any type and class wide invariants that can only be applied to tagged types.

The invariant Is_Unduplicated is a curious example because it cannot be violated by Pop anyway since if there were no duplicates then removing the top item cannot make one appear.

Moreover, Push needs to ensure that the item to be added is not a duplicate of one on the stack already and so essentially much of the checking is repeated. Indeed, when writing Push we should be able to assume that no items are already duplicated and hence all we need to do is check that the new item to be added is not equal to one of the existing items (so n comparisons). However, a general function Is_Unduplicated and will typically compare all pairs and thus require a double loop (so $n(n-1)/2$ comparisons).

The reader is invited to meditate over this conundrum. One's first reaction might be that this is a bad example. However, one way to ensure reliability is to introduce redundancy. Thus if the encoding of Is_Unduplicated and Push are done independently then there is an increased probability that any error will be detected.

The aspect Type_Invariant requires an expression of a Boolean type. The mad programmer could therefore also write

```
type Stack is private
  with Type_Invariant;
```

which would thus be True by default and so useless! Actually it might not be entirely useless since it might act as a placeholder for an invariant to be defined later and meanwhile the program will compile and execute.

Type invariants are useful whenever a type is more than just the sum of its components. Note carefully that the invariant may not hold when an object is being manipulated by a subprogram having access to the full type. In the case of Push and Pop and the invariant Is_Unduplicated this will not happen but consider the following simple example.

Suppose we have a type Point which describes the position of an object in a plane. It might simply be

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

Now suppose we want to ensure that all points are within a unit circle. We could ensure that a point lies within a square by means of range constraints by writing

```
type Point is
  record
    X, Y: Float range –1.0 .. +1.0;
  end record;
```

but we need to ensure that X**2 + Y**2 is not greater than 1.0, and that cannot be done by individual constraints. So we might declare a type Disc_Pt with an invariant as follows

```
package Places is

  type Disc_Pt is private
    with Type_Invariant => Check_In(Disc_Pt);

  function Check_In(D: Disc_Pt) return Boolean
    with Inline;

  ...                          -- various operations on disc points

private

  type Disc_Pt is
    record
      X, Y: Float range –1.0 .. +1.0;
    end record;

  function Check_In(D: Disc_Pt) return Boolean is
    (D.X**2 + D.Y**2 <= 1.0);                -- completion

end Places;
```

Note that we have used an expression function in the private part as the completion for Check_In. They are very useful for small functions in situations like this and typically will be given the aspect Inline on the specification as shown.

Now suppose that we wish to make available to the user a procedure Flip that reflects a Disc_Pt in the line $x = y$, or in other words interchanges its X and Y components. The body might be

```
procedure Flip(D: in out Disc_Pt) is
   T: Float;                    -- temporary
begin
   T := D.X;  D.X := D.Y;  D.Y := T;
end Flip;
```

This works just fine but note that just before the assignment to D.Y, it is quite likely that the invariant does not hold. If the original value of D was (0.1, 0.8) then at the intermediate stage it will be (0.8, 0.8) and so well outside the unit circle.

So there is a general principle that an intermediate value not visible externally need not satisfy the invariant. There is an analogy with numeric types. The intermediate value of an expression can fall outside the range of the type but will be within range when the final value is assigned to the object. For example, suppose type Integer is 16 bits (a small machine) but the registers perform arithmetic in 32 bits, then a statement such as

```
J := K * L / M;
```

could easily produce an intermediate result K * L outside the range of Integer but the final value could be in range.

In many cases it will not be necessary for the user to know that a type invariant applies to the type; it is after all merely a detail of the implementation. So perhaps the above should be rewritten as

```
package Places is
   type Disc_Pt is private;
   ...                          -- various operations on disc points
private
   type Disc_Pt is
      record
         X, Y: Float range -1.0 .. +1.0;
      end record
      with Type_Invariant => Disc_Pt.X**2 + Disc_Pt.Y**2 <= 1.0;
end Places;
```

In this case we do not need to declare a function Check_In at all. Note the use of the type name Disc_Pt in the invariant expression. This is an example of the use of a type name to denote a current instance.

We now turn to consider the places where a type invariant on a private type T is checked. These are basically when it can be changed from the point of view of the outside user. They are

- after default initialization of an object of type T,
- after a conversion to type T,
- after assigning to a view conversion involving descendants and ancestors of type T,
- after a call of T'Read or T'Input,
- after a call of a subprogram declared in the immediate scope of T and visible outside that has a parameter (of any mode including an access parameter) with a part of type T or returns a result with a part of type T.

Note that by saying a part of type T, the checks not only apply to subprograms with parameters and results of type T but they also apply to parameters and results whose components are of the type T or are view conversions involving the type T.

Observe that parameters of mode **in** are also checked because, as is well known, there are accepted techniques for changing such parameters. However, checks on **in** parameters only apply to procedures and not to functions. This was a retrospective change after ISO standardization as explained in AI12-44. This change was necessary to avoid infinite recursion which would arise if an invariant itself called a function with a parameter of the type.

Beware, however, that the checks do not extend to deeply nested situations, such as components with components that are access values to objects that themselves involve type T or worse. Thus there are holes in the protection offered by type invariants. However, if the types are straightforward and the writer does not do foolish things like surreptitiously exporting access types referring to T then all will be well.

The checks on type invariants regarding parameters and results can be conveniently implemented in the body of the subprogram in much the same way as for postconditions. This saves duplicating the code of the tests at each point of call.

If a subprogram such as Flip which is visible outside is called from inside then the checks still apply. This is not strictly necessary of course, but fits the simple model of the checks being in the body and so simplifies the implementation.

If an untagged type is derived then any existing specific invariant is inherited for inherited operations. However, a further invariant can be given as well and both will apply to the inherited operations. This fits in with the model of view conversions used to describe how an inherited subprogram works on derivation. The parameters of the derived type are view converted to the parent type before the body is called and back again afterwards. As mentioned above, view conversions are one of the places where invariants are checked.

However, if we add new operations then the old invariant does not apply to them. In truth, the specific invariant is not really inherited at all; it just comes along for free with the inherited operations that are not overridden. So if we do add new operations then we need to state the total invariant required.

Note that this is not quite the same model as specific postconditions. We cannot add postconditions to an inherited operation but have to override it and then any specific postconditions on the parent are lost. In any event, in both cases, if we want to use inheritance then we should really use tagged types and class wide aspects.

So there is also an aspect Type_Invariant'Class for use with private tagged types. The distinction between Type_Invariant and Type_Invariant'Class has similarities to that between Post and Post'Class.

The specific aspect Type_Invariant can be applied to any type but Type_Invariant'Class can only be applied to tagged types. A tagged type can have both an aspect Type_Invariant and Type_Invariant'Class.

Type_Invariant cannot be applied to an abstract type. Type_Invariant'Class is inherited by all derived types; it can also be applied to an abstract type.

Note the subtle difference between Type_Invariant and Type_Invariant'Class. Type_Invariant'Class is inherited for all operations of the type but as noted above Type_Invariant is only incidentally inherited by the operations that are inherited.

An interesting rule is that Type_Invariant'Class cannot be applied to a full type declaration which completes a private type such as Disc_Pt in the example above. This is because the writer of an extension will need to see the applicable invariants and this would not be possible if they were in the private part.

So if we have a type T with a class wide invariant thus

```
type T is tagged private
   with Type_Invariant'Class => F(T);
procedure Op1(X: in out T);
procedure Op2(X: in out T);
```

and then write

```
type NT is new T with private
   with Type_Invariant'Class => FN(NT);
overriding
procedure Op2(X: in out NT);
not overriding
procedure Op3(X: in out NT);
```

then both invariants F and FN will apply to NT.

Note that the procedure Op1 is inherited unchanged by NT, procedure Op2 is overridden for NT and procedure Op3 is added.

Now consider various calls. The calls of Op1 will involve view conversions as mentioned earlier and these will apply the checks for FN and the inherited body will apply the checks for F. The body of Op2 will directly include checks for F and FN as will the body of Op3. So the invariant F is properly inherited and all is well.

Remember that if the invariants were specific and not class wide then although Op1 will have checks for F and FN, Op2 and Op3 will only check FN.

In the case of the type Disc_Pt we might decide to derive a type which requires that all values are not only inside the unit circle but outside an inner circle – in other words in an annulus or ring. We use the class wide invariants so that the parent package is

```
package Places is

   type Disc_Pt is tagged private
      with Type_Invariant'Class => Check_In(Disc_Pt);

   function Check_In(D: Disc_Pt) return Boolean
      with Inline;

   …                          -- various operations on disc points
private
```

```
   type Disc_Pt is tagged
      record
         X, Y: Float range –1.0 .. +1.0;
      end record;

   function Check_In(D: Disc_Pt) return Boolean is
      (D.X**2 + D.Y**2 <= 1.0);

end Places;
```

And then we might write

```
package Places.Inner is

   type Ring_Pt is new Disc_Pt with null record
      with Type_Invariant'Class => Check_Out(Ring_Pt);

   function Check_Out(R: Ring_Pt) return Boolean
      with Inline;

private

   function Check_Out(R: Ring_Pt) return Boolean is
      (R.X**2 + R.Y**2 >= 0.25);

end Places.Inner;
```

And now the type Ring_Pt has both its own type invariant but also that inherited from Disc_Pt thereby ensuring that points are within the ring or annulus.

Finally, it is worth emphasizing that it is good advice not to use inheritance with specific invariants but they are invaluable for checking internal and private properties of types.

Exercise 16.3

1 Write the function body for Is_Unduplicated assuming the general case requiring the need to make all comparisons. The implementation can be assumed to be as in Section 12.4.

2 Rewrite the function Is_Unduplicated as an expression function.

16.4 Subtype predicates

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

Accordingly, Ada 2012 introduces subtype predicates as an aspect that can be applied to type and subtype declarations. The requirements proved awkward to satisfy with a single feature so in fact there are two aspects: Static_Predicate and Dynamic_Predicate. They both take a Boolean expression and the key difference is that the static predicate is restricted to certain types of expressions so that it can be used in more contexts.

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

```
type Month is (Jan, Feb, Mar, Apr, May, ..., Nov, Dec);
```

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

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

and then we are assured that objects of subtype Winter can only be Dec, Jan or Feb (provided once more that the Assertion_Policy pragma has set the Policy to Check). Note the use of the subtype name (Winter) in the expression where it stands for the current instance of the subtype.

The aspect is checked whenever an object is default initialized, on assignments, on conversions, on parameter passing and so on. If a check fails then Assertion_Error is raised.

As another example, suppose we wish to specify that an integer is even. We might expect to be able to write

```
subtype Even is Integer
  with Static_Predicate => Even mod 2 = 0;          -- illegal
```

Sadly, this is illegal because the expression in a static predicate is restricted and cannot use some operations such as **mod**. In detail, the expression can only be

- a static membership test where the choice is selected by the current instance,
- a case expression whose dependent expressions are static and selected by the current instance,
- a call of the predefined operations =, /=, <, <=, >, >= where one operand is the current instance,
- an ordinary static expression,

and, in addition, a call of a Boolean logical operator **and**, **or**, **xor**, **not** whose operands are such static predicate expressions, and, a static predicate expression in parentheses.

So we see that the predicate in the subtype Even cannot be a static predicate because the operator **mod** is not permitted with the current instance. But **mod** could be used in an inner static expression.

As a consequence, we have to use a dynamic predicate thus

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;       -- OK
```

Note that a subtype with predicates cannot be used in some contexts such as index constraints. This is to avoid having arrays with holes and similar nasty things.

However, static predicates are allowed in a for loop meaning to try every value. So we could write

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

Beware that the loop uses values for M in the order, Jan, Feb, Dec and not Dec, Jan, Feb as the user might have wanted.

On the other hand, dynamic predicates are not allowed in a loop in this way so we cannot write

```
for X in Even loop ...
```

but have to spell it out in detail such as

```
for X in Integer loop
   if X mod 2 = 0 then                    -- or if X in Even then
      ...                          -- body of loop
   end if;
end loop;
```

Looking at the detailed rules we see that the predicate in the subtype Winter can be a static predicate because it takes the form of a membership test where the choice is selected by the current instance and whose individual items are themselves all static. Another useful example of this kind is

```
subtype Letter is Character
   with Static_Predicate => Letter in 'A' .. 'Z' | 'a' .. 'z';
```

Static case expressions are valuable because they provide the comfort of covering all values of the current instance. Suppose we have a type Animal

```
type Animal is (Bear, Cat, Dog, Horse, Wolf);
```

We could then declare a subtype of friendly animals

```
subtype Pet is Animal
   with Static_Predicate => Pet in Cat | Dog | Horse;
```

and perhaps

```
subtype Predator is Animal
   with Static_Predicate => not (Predator in Pet);
```

or equivalently

```
subtype Predator is Animal
   with Static_Predicate => Predator not in Pet;
```

Now suppose we add Rabbit to the type Animal. Assuming that we consider that rabbits are pets and not food, we should change Pet to correspond but we might forget with awkward results. Maybe we have a procedure Hunt which aims to eliminate predators

```
procedure Hunt(P: in out Predator);
```

and we will find that our poor rabbit is hunted rather than petted!

What we should have done is use a case expression controlled by the current instance thus

```
subtype Pet is Animal
   with Static_Predicate =>
      (case Pet is
         when Cat | Dog | Horse => True,
         when Bear | Wolf => False);
```

and now if we add Rabbit to Animal and forget to update Pet to correspond then the program will fail to compile.

Note that a similar form of if expression where the current instance has to be of a Boolean type would not be useful and so is excluded.

Static subtypes with static predicates can also be used in case statements. Thus elsewhere in the program we might have

```
case Animal is
   when Pet =>          ...          -- feed it
   when Predator =>     ...          -- feed on it
end case;
```

Observe that we do not have to list all the individual animals and naturally there is no others clause. If other animals are added to Pet or Predator then this case statement will not need changing. Thus not only do we get the benefit of full coverage checking, but the code is also maintenance free. Of course if we add an animal that is neither a Pet nor Predator (Sloth perhaps?) then the case statement will need updating.

Subtype predicates, like pre- and postconditions and type invariants are similarly monitored by the pragma Assertion_Policy. If a predicate fails (that is, has value False) then Assertion_Error is raised.

Subtype predicates are checked in much the same sort of places as type invariants. Thus

- on a subtype conversion,
- on parameter passing (which covers expressions in general),
- on default initialization of an object.

Note an important difference from type invariants. If a type invariant is violated then the damage has been done. But subtype predicates are checked before any damage is done. This difference essentially arises because type invariants apply to private types and can become temporarily false inside the defining package as we saw with the procedure Flip applying to the type Disc_Pt.

If an object is declared without initialization and no default applies then any subtype predicate might be false in the same way that a subtype constraint might be violated.

Beware that subtype predicates like type invariants are not foolproof. Thus in the case of a record type they apply to the record as a whole but they are not checked if an individual component is modified.

Subtype predicates can be given for all types in principle. Thus we might have

```
type Date is
   record
      D: Integer range 1 .. 31;
      M: Month;
      Y: Integer;
   end record;
```

and then

```
subtype Winter_Date is Date
   with Dynamic_Predicate => Winter_Date.M in Winter;
```

Note how this uses the subtype Winter which was itself defined by a subtype predicate. However, Winter_Date has to have a Dynamic_Predicate because the selector is not simply the current instance but a component of it.

We can now declare and manipulate a Winter_Date

```
WD: Winter_Date := (25, Dec, 2011);
...
Do_Date(WD);
```

and the subtype predicate will be checked on the call of Do_Date. However, beware that if we write

```
WD.Month := Jun;                                -- dodgy
```

then the subtype predicate is not checked because we are modifying an individual component and not the record as a whole.

Subtype predicates can be given with type declarations as well as with subtype declarations. Consider for example declaring a type whose only allowed values are the possible scores for an individual throw when playing darts. These are 1 to 20 and doubles and trebles plus 50 and 25 for an inner and outer bull's eye.

We could write these all out explicitly

```
type Score is new Integer
   with Static_Predicate =>
      Score in 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16
             | 17 | 18 | 19 | 20 | 21 | 22 | 24 | 25 | 26 | 27 | 28 | 30 | 32 | 33
             | 34 | 36 | 38 | 39 | 40 | 42 | 45 | 48 | 50 | 51 | 54 | 57 | 60;
```

But that is rather boring and obscures the nature of the predicate. We can split it down by first defining individual subtypes for singles, doubles and trebles as follows

```
subtype Single is Integer range 1 .. 20;

subtype Double is Integer
   with Static_Predicate =>
      Double in 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28
              | 30 | 32 | 34 | 36 | 38 | 40;
```

```
subtype Treble is Integer
   with Static_Predicate =>
      Treble in 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 | 42
             | 45 | 48 | 51 | 54 | 57 | 60;

subtype Score is Integer
   with Static_Predicate =>
      Score in Single or Score in Double or Score in Treble or
      Score in 25 | 50;
```

Note that it would be neater to write

```
subtype Score is Integer
   with Static_Predicate =>
      Score in Single | Double | Treble | 25 | 50;
```

Observe that it does not matter that the individual predicates overlap. That is a score such as 12 is a Single, a Double and a Treble.

If we do not mind the predicates being dynamic then we can write

```
subtype Double is Integer
   with Dynamic_Predicate =>
      Double mod 2 = 0 and Double / 2 in Single;
```

and so on. Or we could even use a quantified expression

```
subtype Double is Integer
   with Dynamic_Predicate =>
      (for some K in Single => Double = 2*K);
```

or go all the way in one lump

```
type Dyn_Score is new Integer
   with Dynamic_Predicate =>
      (for some K in 1 .. 20 => Score = K or Score = 2*K or Score = 3*K)
       or Score in 25 | 50;
```

There are some restrictions on the use of subtypes with predicates. As mentioned above, if a subtype has a static or dynamic predicate then it cannot be used as an array index subtype. This is to avoid arrays with holes. So we cannot write

```
type Winter_Hours is array (Winter) of Hours;          -- illegal
type Hits is array (Score range <>) of Integer;        -- illegal
```

Similarly, we cannot use a subtype with a predicate to declare the range of an array object or to select a slice. So if we have

```
type Month_Days is array (Month range <>) of Integer;
The_Days: Month_Days := (31, 28, 31, 30, ... );
```

then we cannot write

```
Winter_Days: Month_Days(Winter);                    -- illegal array
The_Days(Winter) := (Jan | Dec => 31, Feb => 29);   -- illegal slice
```

However, a subtype with a static predicate can be used in a for loop thus

```
for W in Winter loop ...
```

and in a named aggregate such as

```
(Winter => 10.0, others => 14.0);                   -- OK
```

but a subtype with a dynamic predicate cannot be used in these ways. Actually the restriction is slightly more complicated. If the original subtype is not static such as

```
subtype To_N is Integer range 1 .. N;
```

then even if To_N has a static predicate it still cannot be used in a for loop or named aggregate.

These rules can also be illustrated by considering the dartboard. We might like to accumulate a count of the number of times each particular score has been achieved. So we might like to declare

```
type Hit_Count is array (Score) of Integer;         -- illegal
```

but sadly this would result in an array with holes and so is forbidden. However, we could declare an array from 1 to 60 and then initialize it with 0 for those components used for hits and –1 for the unused components. Of course, we ought not to repeat literals such as 1 and 60 because of potential maintenance problems. But, we can use new attributes First_Valid and Last_Valid thus

```
type Hit_Count is array (Score'First_Valid .. Score'Last_Valid) of Integer
                                          := (Score => 0, others => –1);
```

which uses Score to indicate the used components. The attributes First_Valid and Last_Valid can be applied to any static subtype but are particularly useful with static predicates.

In detail, First_Valid returns the smallest valid value of the subtype. It takes any range and/or predicate into account whereas First only takes the range into account. Similarly Last_Valid returns the largest value. Incidentally, they are illegal on null subtypes (because null subtypes have no valid values at all).

The Hit_Count array can then be updated by the value of each hit as expected

```
A_Hit: Score := ... ;                               -- next dart
Hit_Count(A_Hit) := Hit_Count(A_Hit) + 1;
```

If we attempt to assign a value of type Integer which is not in the subtype Score to A_Hit then Assertion_Error is raised (unless a different exception is specified as explained in the next section).

After the game, we can now loop through the subtype Score and print out the number of times each hit has been achieved and perhaps accumulate the total at the same time thus

```
for K in Score loop
   New_Line;  Put(Hit);  Put(Hit_Count(K));
   Total := Total + K * Hit_Count(K);
end loop;
```

The reason for the distinction between static and dynamic predicates is that the static form can be implemented as small sets with static operations on the small sets. Hence the loop

```
for K in Score loop ...
```

can be implemented simply as a sequence of 43 iterations. However, a loop such as

```
for X in Even loop ...
```

which might look innocuous requires iterating over the whole set of integers. Thus we insist on having to write

```
for X in Integer loop
   if X in Even then ...
```

which makes the situation quite clear.

Another restriction on the use of subtypes with predicates is that the attributes First, Last and Range cannot be applied. But Pred and Succ are permitted because they apply to the underlying type. As a consequence, if a generic body (see Section 19.1) uses First, Last or Range on a formal type and the actual type has a subtype predicate then Program_Error is raised.

Subtype predicates can be applied to abstract types but not to incomplete types. Subtype predicates are inherited as expected on derivation. Thus if we have

```
type T is ...
   with Static_Predicate => PredS;
```

and then

```
type NT is new T
   with Dynamic_Predicate => PredD;
```

the result is that both predicates apply to NT rather as if we had written the predicate as PredS **and then** PredD. Note that they are applied in order. If any one is dynamic then restrictions on the use of subtypes with a dynamic predicate apply.

There is no need for special predicates for class wide types in the way that we have both Type_Invariant and Type_Invariant'Class. So in the general case where a tagged type is derived from a parent and several progenitors

```
type T is new P and G1 and G2 with ...
```

where P is the parent and G1 and G2 are progenitors, the subtype predicate applicable to T is simply those for P, G1 and G2 all anded together.

Exercise 16.4

1 The type Rainbow is declared as

```
type Rainbow is (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
```

declare a subtype Primary whose only values are Red, Yellow and Blue.

2 Declare a subtype of Integer whose only values are positive integers less than 1000 and have remainder 1 when divided by 37. Consider both static and dynamic predicates.

16.5 Messages

A number of improvements were made in this area after ISO standardization by AI12-54-2 These concern a new aspect Predicate_Failure which enables specific messages to be associated with a failure and rules regarding the order of evaluation of predicates if several apply.

The expected type of the expression defined by the aspect Predicate_Failure is String and gives the message to be associated with a failure. This can be illustrated using the type File_Type and associated operations for input–output as described in Section 23.5. We can write

```
subtype Open_File_Type is File_Type
  with
    Dynamic_Predicate => Is_Open(Open_File_Type),
    Predicate_Failure =>  "File not open";
```

If the predicate fails then Assertion_Error is raised with the message "File not open".

We can also use a raise expression and thereby ensure that a more appropriate exception is raised. If we write

```
Predicate_Failure => raise Status_Error with "File not open";
```

then Status_Error is raised rather than Assertion_Error with the given message. We could of course explicitly mention Assertion_Error by writing

```
Predicate_Failure => raise Assertion_Error with "A message";
```

Finally, we could omit any message and just write

```
Predicate_Failure => raise Status_Error;
```

in which case the message is null.

A related issue is discussed in AI12-71. If several predicates apply to a subtype which has been declared by a refined sequence then the predicates are evaluated in the order in which they occur. This is especially important if different exceptions are specified by the use of Predicate_Failure since without this rule the wrong exception might be raised. The same applies to a combination of predicates, null exclusions and old-fashioned subtypes.

This can be illustrated by an extension of the above example. Suppose we have

```
subtype Open_File_Type is File_Type
   with
      Dynamic_Predicate => Is_Open(Open_File_Type),
      Predicate_Failure => raise Status_Error;
subtype Read_File_Type is Open_File_Type
   with
      Dynamic_Predicate => Mode(Real_File_Type) = In_File,
      Predicate_Failure => raise Mode_Error with "Can't read file: " &
                                                 Name(Read_File_Type);
```

The subtype Read_File_Type refines Open_File_Type. If the predicate for it were evaluated first and the file was not open then the call of Mode would raise Status_Error which we would not want to happen if we wrote

```
if F in Read_File_Type then ...
```

Care is needed with membership tests which were discussed in Section 9.1. The whole purpose of a membership test (and similarly the Valid attribute) is to find out whether a condition is satisfied. So if we write

```
if X in S then
   ...                        -- do this
else
   ...                        -- do that
end if;
```

we expect the membership test to be true or false. However, if the evaluation of S itself raises some exception then the purpose of the test is violated.

It is important to understand these related topics. Consider once more a very simple predicate such as

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

and suppose we declare a variable W thus

```
W: Winter := Jan;
```

If we now do

```
W := Mar;
```

then Assertion_Error will be raised because the value Mar is not within the subtype Winter (assuming that the assertion policy is Check). If, however, we would rather have Constraint_Error raised then we can modify the declaration of Winter to

```
subtype Winter is Month
   with Static_Predicate => Winter in Dec | Jan | Feb,
        Predicate_Failure => raise Constraint_Error;
```

and then obeying

```
W := Mar;
```

will raise Constraint_Error.

On the other hand suppose we declare a variable M thus

```
M: Month := Mar;
```

and then do a membership test

```
if M in Winter then
   ...                        -- do this if M is a winter month
else
   ...                        -- do this if M is not a winter month
end if;
```

then of course no exception is raised since this is a membership test and not a predicate check.

Note however, that we could write something odd such as

```
subtype Winter2 is Month
   with Dynamic_Predicate =>
                        (if Winter2 in Dec | Jan | Feb then True else raise E);
```

then the very evaluation of the predicate might raise the exception E so that

```
M in Winter2
```

will either be True or raise the exception E but will never be False. Note that in this contrived example the predicate has to be a dynamic one because a static predicate cannot include a raise expression.

So this should clarify the reasons for introducing Predicate_Failure. It enables us to give a different behaviour for when the predicate is used in a membership test as opposed to when it is used in a check and it also allows us to add a message.

Finally, it should be noted that the predicate expression might involve the evaluation of some subexpression perhaps through the call of some function. We might have a predicate describing those months that have 30 days thus

```
subtype Month30 is Month
   with Static_Predicate => Month30 in Sep | Apr | Jun | Nov;
```

which mimics the order in the nursery rhyme. However, suppose we decide to declare a function Days30 to do the check so that the subtype becomes

```
subtype Month30 is Month
   with Dynamic_Predicate => Days30(Month30);
```

and for some silly reason we code the function incorrectly so that it raises an exception (perhaps it accidentally runs into its **end** and always raises Program_Error). In this situation if we write

```
M in Month30
```

then we will indeed get Program_Error and not false.

Perhaps this whole topic can be summarized by simply saying that a membership test is not a check. Indeed a membership test is often useful in ensuring that a subsequent check will not fail as was discussed in Section 9.1.

Exercise 16.5

1 Rewrite the declaration of the subtype Even in Section 16.4 so that the exception Constraint_Error is raised with an appropriate message if the predicate is violated.

Checklist 16

Preconditions and postconditions are given by Boolean aspects; they are checked on the call of a subprogram and on its return respectively.

Remember LSP for class wide pre- and postconditions.

Type invariants are checked when the external view could change; however, they are not foolproof.

Subtype predicates are more like constraints.

These facilities are controlled by the pragma Assertion_Policy.

Violation of a predicate raises Assertion_Error unless we use the aspect Predicate_Failure to indicate a different exception.

New in Ada 2012

This chapter is all new in Ada 2012.

17 Numeric Types

We now come at last to a more detailed discussion of numeric types. There are two categories of numeric types in Ada: integer types and real types. The integer types are subdivided into signed integer types and modular types (these are unsigned). The real types are subdivided into floating point types and fixed point types; the fixed point types are further subdivided into ordinary fixed point types and decimal types.

There are two problems concerning the representation of numeric types in a computer. First, the range will inevitably be restricted and indeed many machines have hardware operations for various ranges so that we can choose our own compromise between range of values and space occupied by values. Secondly, it may not be possible to represent accurately all the possible values of a type. These difficulties cause problems with program portability because the constraints vary from machine to machine. Getting the right balance between portability and performance is not easy. The best performance is achieved by using types that correspond exactly to the hardware. Perfect portability requires using types with precisely identical range and accuracy and identical operations.

Ada recognizes this situation and provides numeric types in ways that allow a programmer to choose the correct balance for the application. High performance can thus be achieved while keeping portability problems to a minimum.

We start by discussing integer types because these are somewhat simpler since they are parameterized only by their range whereas the real types are parameterized by both range and accuracy.

17.1 Signed integer types

All implementations of Ada have the predefined type Integer. In addition there may be other predefined types such as Long_Integer, Short_Integer and so on

with a respectively longer or shorter range than Integer (could actually be the same). The range of values of these predefined types will be symmetric about zero except for an extra negative value in two's complement machines (which now seem to dominate over one's complement machines). All predefined integer types have the same predefined operations that were described in Chapter 6 as applicable to the type Integer (except that the second operand of "**" is always of type Integer).

Thus we might find that on machine A we have types Integer and Long_Integer with

```
range of Integer:
   –32_768 .. +32_767                                  (i.e. 16 bits)

range of Long_Integer:
   –2147_483_648 .. +2147_483_647                      (i.e. 32 bits)
```

whereas on machine B we might have types Short_Integer, Integer and Long_Integer with

```
range of Short_Integer:
   –2048 .. +2047                                      (i.e. 12 bits)

range of Integer:
   –8388_608 .. +8388_607                              (i.e. 24 bits)

range of Long_Integer:
   –140_737_488_355_328 .. +140_737_488_355_327        (i.e. 48 bits)
```

For most purposes the type Integer will suffice on either machine and that is why we have simply used Integer in examples in this book so far. However, suppose we have an application where we need to manipulate signed values up to a million. The type Integer is inadequate on machine A and to use Long_Integer on machine B would be extravagant. We *could* overcome this problem by using derived types and writing (for machine A)

```
type My_Integer is new Long_Integer;
```

and then using My_Integer throughout the program. To move the program to machine B we would just replace this one declaration by

```
type My_Integer is new Integer;
```

However, Ada enables the choice between the underlying types to be made automatically. If we write

```
type My_Integer is range –1E6 .. 1E6;
```

then the implementation will implicitly choose the smallest appropriate hardware type and it will be somewhat as if we had written

```
type My_Integer is new hardware_type range –1E6 .. 1E6;
```

where the anonymous *hardware_type* is chosen appropriately. So in effect My_Integer will be a subtype of an anonymous type based on one of the predefined

types and so objects of type My_Integer will be constrained to take only the values in the range –1E6 .. 1E6 and not the full range of the anonymous type. Note that the range must have static bounds since the choice of machine type is made at compile time.

To understand what is really going on we need to distinguish between the *range* of a type and the *base range* of a type. The range of a type is the requested range whereas the base range is the actual implemented range. So in the case of My_Integer, the range is –1E6 .. 1E6 whereas the base range is the full range of the underlying anonymous machine type.

The base type can be indicated by applying the attribute Base to the type. Thus My_Integer'Base is the base type (strictly the base subtype) and My_Integer'Base 'Range is the base range; we could declare variables of this subtype by writing

```
Var: My_Integer'Base;
```

Note that the attribute Base always denotes an unconstrained subtype whereas My_Integer is constrained.

It is an important rule that range checks are applied to constrained subtypes but not to unconstrained subtypes. If a range check fails then Constraint_Error is raised. Although range checks do not apply to unconstrained subtypes nevertheless overflow checks always apply and so we either get the correct mathematical result or Constraint_Error is raised. Using a constrained subtype is more portable whereas an unconstrained subtype is likely to be more efficient. But note that overflow checks in particular may well be automatically performed by the hardware with no time penalty.

Another point regarding unconstrained types is that the compiler is permitted to optimize intermediate expressions and storage for variables by using a larger range if helpful. For example, all registers might be of 32 bits and all arithmetic be performed using 32 bits on machine A even though the type Integer itself was only 16 bits. As a consequence, if an unconstrained variable of type Integer were held in such a register then it could have values outside its 16-bit base range. This is naturally only possible for an unconstrained variable.

We can illustrate these points by considering

```
type Index is range 0 .. 20_000;
I, J, K: Index;
IB, JB: Index'Base;
```

on machine A. The range of Index is then 0 .. 20_000 as given, whereas the base range is –32_768 .. +32_767. Suppose furthermore that IB is held in a 16-bit store whereas JB is optimized and held in a 32-bit register. Now consider

```
I := 17_000;
J := 16_000;
...
K := I + J;                -- range check fails, Constraint_Error
IB := I + J;               -- overflows, Constraint_Error
JB := I + J;               -- OK
```

The addition of I and J is successfully performed in a 32-bit register. The assignment to the constrained variable K fails because the result is outside the range of Index. The assignment to IB fails because the value is outside the base range and IB is implemented using the base range. The assignment to JB succeeds because JB is in a 32-bit register.

The fact that the addition can be performed using registers with a larger range than the base range is reflected by the fact that the specification of "+" can be thought of as

```
function "+" (Left, Right: Index'Base) return Index'Base;
```

so that there are no range checks on evaluating the parameters or returning the results.

Similar considerations apply to the predefined named types such as Integer. The subtype Integer is constrained whereas Integer'Base is unconstrained. The range and base range happen to be the same. The predefined operations take the form

```
function "+" (Left, Right: Integer'Base) return Integer'Base;
```

as shown in Section 23.1. Note the strange formal parameter names Left and Right; these are the usual names for parameters of the predefined operators.

Not all the implemented base ranges need correspond to a predefined type such as Integer or Long_Integer declared in Standard. There could be others. Indeed the *ARM* recommends that only Integer and Long_Integer be given such names. The type Integer will always exist and have at least a 16-bit range; if Long_Integer does exist then it will have at least a 32-bit range.

In addition to the types Integer and Long_Integer an implementation should provide types directly corresponding to the hardware in the package Interfaces. These will have names such as Integer_8, Integer_16, ..., Integer_128 according to the supported ranges.

We can convert between one integer type and another by using the normal notation for type conversion. Given

```
type My_Integer is range –1E6 .. 1E6;
type Index is range 0 .. 20_000;
M: My_Integer;
I: Index;
```

then we can write

```
M := My_Integer(I);
I := Index(M);
```

On machine A a genuine hardware conversion is necessary but on machine B both types will have the same representation as Integer and the conversion will be null (we assume that both machines have no other hardware types than those corresponding to the predefined types mentioned at the start of this section).

All the integer types can be thought of as being ultimately derived from an anonymous type known as *root_integer*. The range of this type *root_integer* is System.Min_Int .. System.Max_Int where Min_Int and Max_Int are constants declared in the package System. These are the minimum and maximum signed

integer values supported by the executing program. The type *root_integer* has all the usual integer operations. (The derivation model should not be taken too literally as we shall see when we discuss generic parameters in Section 19.2.)

The integer literals are considered to belong to another anonymous type known as *universal_integer*. The range of this type is essentially infinite. Integer numbers declared in a number declaration (see Section 6.1) such as

```
Ten: constant := 10;
```

are also of type *universal_integer*. However, there are no *universal_integer* variables and no *universal_integer* operations.

Conversion from *universal_integer* to all other integer types is implicit and does not require an explicit type conversion. In fact the type *universal_integer* behaves a bit like the class wide type for integers; that is like *root_integer*'Class. (But of course there is no tag and so the analogy cannot be taken too far.)

It should be noted that some attributes such as the function Pos in fact deliver a *universal_integer* value and since Pos can take a dynamic argument it follows that certain *universal_integer* expressions may actually be dynamic (but they immediately get converted to some other integer type – possibly *root_integer* if the context does not impose a specific type).

The initial value in a number declaration has to be static; it can be of any integer type. So

```
M: constant := 10;
MM: constant := M * M;
N: constant Integer := 10;
NN: constant := N * N;
```

are all allowed.

Sometimes an expression involving literals is ambiguous. In such a case there is a preference for the type *root_integer* if that overcomes the ambiguity. For example

```
MM: constant := M * M;
```

is apparently ambiguous since the operator "*" is defined for all the specific integer types (including *root_integer* but not *universal_integer*). It is then taken to be *root_integer* by this preference rule.

Of course, since these expressions are all static and thus evaluated at compile time the distinction is somewhat theoretical especially since static expressions are always evaluated exactly. However, it is an important rule that a static expression may not exceed the base range of the expected type (such as Integer for N above).

A fuller description of static expressions will be found in Section 27.1 but the general rule for scalar types is that if it looks static then it is static. Thus 2 + 3 is always static and not just in a context demanding a static expression.

Recall that a range such as 1 .. 10 occurring in a for statement or in an array type definition or in a quantified expression is considered to be of type Integer. This is a consequence of the preference rule and a special rule which says that a range of type *root_integer* in such contexts is taken to be Integer. Note incidentally that specifying that the range is type Integer rather than any other integer type means that the predefined type Integer does have rather special properties.

We now see why we could write

```
for I in –1 .. 10 loop
```

in Section 7.3. This is ambiguous but by preference taken to be of type *root_integer* and then the special rule considers it to be of type Integer.

The use of integer type declarations which reflect the need of the program rather than the arbitrary use of Integer and Long_Integer is good practice because it not only encourages portability but also enables us to distinguish between different kinds of objects as, for example, when we were counting apples and oranges in Section 12.3.

Consideration of separation of concerns is the key to deciding whether to use numeric constants (of a specific type) or named numbers (of type *universal_integer*). If a literal value is a pure mathematical number then it should be declared as a named number. If, however, it is a value related naturally to just one of the program types then it should be declared as a constant of that type. Thus if we are counting oranges and are checking against a limit of 100 it is better to write

```
Max_Oranges: constant Oranges := 100;
```

rather than

```
Max_Oranges: constant := 100;
```

so that the accidental mixing of types as in

```
if No_Of_Apples = Max_Oranges then
```

will be detected by the compiler.

Returning to the question of portability it should be realized that complete portability is not easily obtained. We have seen that although we can specify the ranges for our types and thus ensure that variables are constrained and can never have values outside their range, nevertheless intermediate expressions are often computed with a wider range that is not prescribed exactly. For example, assume

```
type My_Integer is range –1E6 .. +1E6;
I, J: My_Integer;
```

and consider

```
I := 10_000;
J := (I * I) / 5_000;
```

We saw that the operations actually applied to My_Integer'Base – and even then might return values outside the base range. Ignoring this last possibility for the moment, on machine A the intermediate product and final result are computed with no problem and the final result lies within the range of J. But on machine B we get Constraint_Error because the intermediate product 1E8 is outside the base range. So the program is not fully portable.

We could declare

```
function Old_Multiply(Left, Right: My_Integer) return My_Integer
                                                    renames "*";
```

```
function "*"(Left, Right: My_Integer) return My_Integer is
begin
   return Old_Multiply(Left, Right);
end "*";
```

and then this new function "*" will hide the old one; in the new one the operands and result have range checks and so our program will be portable although slow (and this example will fail on both machines!). Note that the hiding and renaming work because the constraints do not matter for the conformance rules involved (see Section 13.7). Remember that My_Integer and My_Integer'Base are both really subtypes of some anonymous type.

In conclusion, portability can be achieved at various levels. Using Integer is acceptable if we know that the values are always within 16 bits. Using Long_Integer is less wise. It is far better to use our own types such as My_Integer. If we are confident about the hardware we are using then we can use types such as Integer_32 in the package Interfaces.

Exercise 17.1

1 What types on machines A and B are used to represent

```
type P is range 1 .. 1000;
type Q is range 0 .. +32768;
type R is range –1E14 .. +1E14;
```

2 Would it make any difference if A and B were one's complement machines with the same number of bits in the representations?

3 Given

```
N: Integer := 6;
P: constant := 3;
R: My_Integer := 4;
```

what are the types of

(a) N + P (c) P + R (e) P * P
(b) N + R (d) N * N (f) R * R

4 Declare a type Longest_Integer which is the maximum supported by the implementation.

17.2 Modular types

The modular types are unsigned integer types and exhibit cyclic arithmetic. Suppose for example that we wish to perform unsigned 8-bit arithmetic (that is byte arithmetic). We can declare

```
type Unsigned_Byte is mod 256;
```

and then the range of values supported by Unsigned_Byte is 0 .. 255. The normal arithmetic operations apply but all arithmetic is performed modulo 256 and overflow cannot occur.

The modulus of a modular type has to be static. It need not be a power of two although it often will be. It might, however, be convenient to use some obscure prime number as the modulus in the implementation of hash tables. Other uses might be to represent naturally cyclic concepts such as compass bearings or the time of day expressed in appropriate units.

The logical operations **and**, **or**, **xor** and **not** are also available on modular types; the binary operations treat the values as bit patterns; the **not** operation subtracts the value from the maximum for the type. These operations are of most use when the modulus is a power of two – in which case **not** has the expected behaviour; they are well defined for other moduli but do have some curious properties. Note that no problems arise with mixing logical operations with arithmetic operations because negative values are not involved.

In the previous section we noted that the package Interfaces contains declarations of types such as Integer_16. Corresponding to each of these there is a modular type such as Unsigned_16. For these modular types (whose modulus will inevitably be a power of two) a number of shift and rotate operations are defined. They are Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left and Rotate_Right. They all have an identical profile such as

```
function Shift_Left(Value: Unsigned_16; Amount: Natural)
                                                    return Unsigned_16;
```

These subprograms have the expected behaviour. They are intrinsic which means that the Access attribute cannot be applied to them in order to produce an access to subprogram value.

Conversion between modular types and signed integer types is possible provided the value is not out of range of the destination. If it is then Constraint_Error is naturally raised.

Thus suppose we had

```
type Signed_Byte is range –128 .. +127;
S: Signed_Byte := –106;
U: Unsigned_Byte := Unsigned_Byte(S);
```

then the type conversion will raise Constraint_Error. However, we can use the attribute Mod which applies to any modular type and converts a *universal_integer* argument to the modular type with appropriate cyclic arithmetic. So we can write

```
U := Unsigned_Byte'Mod(S);
```

and this gives the value 150. Conversion in the opposite direction can be done in various ways. We might simply write

```
if U >= 128 then
   S := Signed_Byte(U – 128) – 128;          -- S is negative
else
   S := Signed_Byte(U);                      -- S is positive
end if;
```

We could of course do the conversions as bit patterns using unchecked conversion as described in Section 25.2. Note that literals and static expressions are

always checked at compile time to ensure they are within the base range of the expected type, and so

```
U := 256;
```

is illegal. On the other hand, a computation such as

```
U := 128;
U := U + U;
```

which is performed with arithmetic modulo 256 results in zero being assigned to U.

Note that since modular types are integer types they are also discrete types and so have all the common properties of the class of discrete types. Subtypes can be declared with a reduced range (this would be unusual). Modular types can be sensibly used as array index types and for loop parameters. An example of an array index dealing with a circular buffer will be found in Section 20.4.

The attribute Modulus applies to modular subtypes and naturally returns the modulus; it is of type *universal_integer*. The attributes Pred and Succ wrap around and so never raise Constraint_Error.

Exercise 17.2

1 Given

```
X: Unsigned_Byte := 16#AB#;
Y: Unsigned_Byte := 16#CD#;
```

what are the values of

(a) X **or** Y
(b) X + Y
(c) X – Y
(d) X * Y

2 Reconsider Exercise 8.2(**2**) using an appropriate modular type.

3 Given

```
type Ring5 is mod 5;
A: Ring5 := 3;
B: Ring5 := 4;
```

what are the values of

(a) **not** (A **and** B)
(b) **not** A **or not** B

4 Declare a modular type suitable for compass bearings accurate to one minute of arc, then declare a constant Degree having the value of one degree. Finally declare constants representing the directions SE, NNW and NE by E. Remember that North is zero on the compass and that East is 90°. (This is a different convention from that used for complex numbers.

17.3 Real types

Integer types are exact types. But real types are approximate and introduce problems of accuracy which have subtle effects. This book is not a specialized

treatise on errors in numerical analysis and so we do not intend to give all the details of how the features of Ada can be used to minimize errors and maximize portability but concentrate instead on outlining the basic principles.

Real types are subdivided into floating point types and fixed point types. Apart from the details of representation, the key abstract difference is that floating point values have a relative error whereas fixed point values have an absolute error. Concepts common to both floating and fixed point types are dealt with in this section and further details of the individual types are in subsequent sections.

There are types *universal_real* and *root_real* having similar properties to *universal_integer* and *root_integer*. Again the preference rule chooses *root_real* in the case of an ambiguity. Static operations on the type *root_real* are notionally carried out with infinite accuracy during compilation. The real literals (see Section 5.4) are of type *universal_real*. Real numbers declared in a number declaration such as

```
Pi: constant := 3.14159_26536;
```

are also of type *universal_real*. (Remember that the difference between an integer literal and a real literal is that a real literal always has a point in it.)

As well as the usual operations on a numeric type, some mixing of *root_real* and *root_integer* operands is also allowed. Specifically, a *root_real* can be multiplied by a *root_integer* and vice versa and division is allowed with the first operand being *root_real* and the second operand being *root_integer* – in all cases the result is *root_real*. So we can write either of

```
Two_Pi: constant := 2 * Pi;          -- legal
Two_Pi: constant := 2.0 * Pi;        -- legal
```

but not

```
Pi_Plus_Two: constant := Pi + 2;     -- illegal
```

because mixed addition is not defined. Note that we cannot do an explicit type conversion between *root_integer* and *root_real* although we can always convert the former into the latter by multiplying by 1.0.

Exercise 17.3

1 Given

```
Two: Integer := 2;
E: constant := 2.71828_18285;
Max: constant := 100;
```

what are the types of

(a) Two * E
(b) Two * Max
(c) E * Max
(d) Two * Two
(e) E * E
(f) Max * Max

2 Given

```
N: constant := 100;
```

declare a real number R having the same value as N.

17.4 Floating point types

All implementations have a predefined type Float and may also have further predefined types Long_Float, Short_Float and so on with respectively more and less precision (could be the same). These types all have the predefined operations that were described in Chapter 6 as applicable to the type Float.

The type Float will have at least 6 digits of precision (provided the hardware can cope) and the type Long_Float (if available) will have at least 11 digits of precision. Other named types in Standard are not recommended in the interests of portability. In a similar way to the integer types, named types corresponding to the hardware supported floating point types may be declared in the package Interfaces with appropriate names such as IEEE_Float_64. (Note the contrast between Integer and Float. Integer always has at least 16 bits but, for awkward hardware, Float is permitted not to have 6 digits.)

However, as mentioned in Section 2.3, it is good practice not to use the predefined types because of potential portability problems. If we write

```
type Real is digits 7;
```

then we are asking the implementation to base Real on a predefined floating point type with at least 7 decimal digits of precision.

The number of decimal digits requested, D, must be static. The actual precision and range supported will of course depend upon the hardware. The actual precision is known as the base decimal precision and the implemented range is the base range. For some awkward hardware, the base decimal precision might not hold over the whole of the base range and so that part of the range for which it does hold is called the safe range. We are guaranteed that the safe range includes –10.0**(4*D) .. +10.0**(4*D). For most applications that is all we need to know.

For specialized applications the Numerics annex contains a description of the minimum guaranteed properties in terms of what are called model numbers. See Section 26.5.

The types Float, Long_Float and Real are unconstrained and so no range checks are imposed on assignments to variables of these types. But overflow checks always apply and raise Constraint_Error if they fail (provided the attribute Machine_Overflows for the type is true).

It is possible to declare a constrained type by for example

```
type Risk is digits 7 range 0.0 .. 1.0;
```

in which case the range constraint must be static. We can impose a (possibly dynamic) range constraint on a floating point subtype or object by

```
R: Real range 0.0 .. 100.0;
```

or

```
subtype Positive_Real is Real range 0.0 .. Real'Last;
```

and so on. If a range is violated then Constraint_Error is raised.

If we do declare a type with a range such as Risk then the safe range will always include the range given and the 4*D rule does not apply. This might force the use of

a higher precision than without the range (not in the case of Risk though which has a relatively small range compared with the precision!)

The attribute Base can be applied to all real types and so Risk'Base gives the corresponding (anonymous) unconstrained subtype. Note that in contrast to Integer, the type Float is unconstrained and consequently Float and Float'Base are the same.

As well as a portable type Real, we might declare a more accurate type Long_Real perhaps for the more sensitive parts of a calculation

```
type Long_Real is digits 12;
```

and then declare variables of the two types

```
R: Real;
LR: Long_Real;
```

as required. Conversion between these follows similar rules to integer types

```
R := Real(LR);
LR := Long_Real(R);
```

and we need not concern ourselves with whether Real and Long_Real are based on the same or different predefined types.

Again in a similar manner to integer types, conversion of real literals, real numbers and *universal_real* attributes to floating types is automatic.

It is possible to apply a digits constraint to a floating point type thus

```
subtype Rough_Real is Real digits 5;
```

Such a constraint has little effect – objects of the subtype will be held in the same way as those of the type Real. The only difference is that the digits attribute of the subtype will be 5 and this will affect the default output format; see Section 23.6. This is considered an obsolescent feature of Ada.

The detailed workings of floating point types are defined in terms of various machine attributes based on a canonical model. In this model, nonzero values are represented as

$$sign.mantissa.radix^{exponent}$$

where

sign is +1 or –1,
radix is the hardware radix such as 2 or 16,
mantissa is a fraction in the base *radix* with nonzero leading digit,
exponent is an integer.

For any subtype S of a type T, the attribute S'Machine_Radix gives the *radix* and the three attributes S'Machine_Mantissa, S'Machine_Emin and S'Machine_Emax collectively give the maximum number of digits in the *mantissa* and the range of values of the *exponent* for which every number in the canonical form is exactly represented in the machine. They all have type universal integer.

For example, one possible representation in a 32-bit word might be in binary with 1 sign bit, 8 bits for the exponent and 23 bits for the mantissa. Since the leading

mantissa bit is always 1, it need not be stored. The exponent can be held in a biased form with logical values ranging from –128 to +127. For such a type the attributes would be 2, 24, –128 and 127 respectively. This format was used on the old DEC PDP-11.

By contrast, the 32-bit IBM 370 format has a hexadecimal radix, with 1 sign bit, 7 exponent bits and 24 mantissa bits. The attributes are 16, 6, –64 and 63. Note carefully that the 24 bits only allow for 6 hexadecimal digits.

There are also the closely related model attributes Model_Mantissa, Model_Emin, Model_Epsilon, Model_Small and the function Model. Although they have to be provided by all implementations and are thus in the core language, their full description relates to implementations supporting the Numerics annex; see Section 26.5. Nevertheless, the following relationships always hold where D is the requested decimal precision for the subtype S

$$g + D\log_{radix}10 \leq \text{Model\_Mantissa} \leq \text{Machine\_Mantissa}$$

$$\text{Model\_Emin} \geq \text{Machine\_Emin}$$

$$\text{Model\_Epsilon} = radix^{(1-\text{Model\_Mantissa})}$$

$$\text{Model\_Small} = radix^{(\text{Model\_Emin}-1)}$$

where g is 1 unless *radix* is a power of 10 in which case g is 0.

The requested precision D determines the choice of underlying type. The model attributes relate to the abstract properties of an ideal hardware type whereas the machine attributes relate to what the hardware actually does (which in many cases will be the same, but sometimes has marginal variations). Model_Epsilon is a measure of the accuracy and is typically the difference between the number one and the next number above one. Model_Small is typically the smallest positive number. These two attributes are of type *universal_real*.

Various attributes enable the specialized programmer to manipulate the internals of floating point numbers. Thus S'Exponent returns the exponent and S'Fraction returns the fraction part. These and other attributes (Adjacent, Ceiling, Compose, Floor, Leading_Part, Machine_Rounding, Remainder, Rounding, Scaling, Truncation and Unbiased_Rounding) are briefly described in Appendix 1.

The attributes for rounding are interesting. Different applications have different requirements. The function Rounding returns the nearest integral value and for midway values rounds away from zero. For the statistically minded Unbiased_Rounding rounds to the even value if midway whereas for those who just want speed Machine_Rounding is ideal since it does whatever the hardware does.

An interesting possibility is that the hardware may distinguish between positive and negative zero. The attribute S'Signed_Zeros is then true. A negative zero would arise for example if a small negative number were divided by a large positive one such that the result underflowed to zero. The distinction can be useful in certain boundary situations; for example the function Arctan behaves differently as mentioned in Section 23.4. Note that negative zero can also be distinguished by copying its sign to a nonzero number, using S'Copy_Sign and then testing the result. But otherwise a negative zero behaves as positive zero and for example X = 0.0 is true if X is a negative zero.

The attribute S'Digits gives the number of decimal digits requested whereas the attribute S'Base'Digits gives the number of decimal digits provided.

There are also the usual attributes S'First and S'Last which are the bounds of the subtype. The base range is S'Base'First .. S'Base'Last and the safe range is S'Safe_First .. S'Safe_Last.

Digits and Base'Digits are of type *universal_integer*; Safe_First and Safe_Last are of type *universal_real*; First and Last are of the base type T.

We do not intend to say more about floating point here but hope that the reader will have understood the principles involved. In general one can simply specify the precision required and all will work. But care is sometimes needed and the advice of a professional numerical analyst should be sought when in doubt. For further details the reader should consult Section 26.5 and the *ARM*.

Exercise 17.4

1 Rewrite the function Inner of Section 10.1 using the type Real for parameters and result and a local type Long_Real with 14 digits accuracy to perform the calculation in the loop.

17.5 Fixed point types

The fixed point types come in two forms: the ordinary fixed point which is based on a binary representation, and decimal fixed point which is based on a decimal representation. Ordinary fixed point is typically used only in specialized applications or for doing approximate arithmetic on machines without floating point hardware. Decimal fixed point is typically used for accountancy and is discussed in the next section.

An ordinary fixed point type declaration specifies an absolute error and also a mandatory range. It takes the form

```
type F is delta D range L .. R;
```

In effect this is asking for values in the range L to R with an accuracy of D which must be positive. D, L and R must be real and static.

The implemented values of a fixed point type are the multiples of a positive real number *small*. The actual value of *small* chosen by the implementation can be any power of 2 less than or equal to D. The base range includes all multiples of *small* within the requested range.

As an example, if we have

```
type T is delta 0.1 range –1.0 .. +1.0;
```

then *small* could be $^{1}/_{16}$. So the implemented values of T will always include

$$-^{15}/_{16},\ -^{14}/_{16},\ \ldots,\ -^{1}/_{16},\ 0,\ +^{1}/_{16},\ \ldots,\ +^{14}/_{16},\ +^{15}/_{16}$$

Note carefully that L and R might actually be outside the base range by as much as *small*; the definition just allows this.

If we have a typical 16-bit implementation then there will be a wide choice of values of *small*. At one extreme *small* could be 2^{-15} which would give us a much greater accuracy but just the required range (the base range would include –1.0 but

not +1.0 on a two's complement machine). At the other extreme *small* could be $^1/_{16}$ in which case we would have a much greater base range but just the required accuracy.

The type T is constrained and range checks will apply; the corresponding unconstrained base subtype is given by T'Base. So the base range as usual is T'Base'Range.

Of course, a different implementation might use just 8 bits for T. Moreover, using attribute definition clauses (which will be discussed in more detail in Chapter 25) it is possible to give the compiler more precise instructions. We can say

```
for T'Size use 5;
```

which will force the implementation to use the minimum 5 bits. Of course, it might be ridiculous for a particular architecture and in fact the compiler is allowed to refuse unreasonable requests.

We can also override the rule that *small* is a power of 2 by using an attribute definition clause. Writing

```
for T'Small use 0.1;
```

will result in the implemented numbers being multiples of 0.1. On an 8-bit implementation the base range would then be from –12.8 to +12.7. Note that if we explicitly specify *small* then any spare bits give extra range and not extra accuracy. This is yet another win for the accountants at the expense of the engineers!

In Ada 2012, both Small and Size can alternatively be set using aspect specifications when the type T is declared. Thus we might write

```
type T is delta 0.1 range –1.0 .. +1.0
   with Size => 5,
        Small => 0.1;
```

The advantage of using the default standard whereby *small* is a power of 2 is that conversion between fixed point and other numeric types can be based on shifting. The use of other values of *small* will in general mean that conversion requires implicit multiplication and division.

A standard simple example is given by

```
Del: constant := 2.0**(–15);
type Frac is delta Del range –1.0 .. 1.0;
```

which will be represented as a pure fraction on a 16-bit two's complement machine. Note that it does not really matter whether the upper bound is written as 1.0 or 1.0–Del; the largest implemented number will be 1.0–Del in either case.

A good example of the use of a specified value for *small* is given by a type representing an angle and which uses the whole of a 16-bit word

```
type Angle is delta 0.1 range –π .. π
   with Small => π * 2.0**(–15);
```

Note that the value given for the aspect Small must not exceed the value for delta. A subtle point is that it is still delta that controls the default output format through the attribute Aft, see Section 23.6.

The predefined arithmetic operations +, –, *, / and **abs** can be applied to fixed point values. Addition and subtraction can only be applied to values of the same type and, of course, return that type. Addition and subtraction are always exact.

Predefined multiplication and division are explained in terms of the anonymous type *universal_fixed*; the parameters and result are of this type which is matched by any fixed point type. As a consequence, multiplication and division are allowed between different fixed point types and the result is then of the type expected by the context. However, if the result is a parameter of a further multiplication or division then the result must first be explicitly converted to a particular type; this ensures that the scale of the intermediate value is properly determined. Multiplication and division by type Integer are also allowed and these return a value of the fixed point type. If the result of a multiplication or division is not an exact multiple of *small* then either adjacent multiple of *small* is permitted.

Conversion of real literals, real numbers and *universal_real* attributes to fixed point types (including *universal_fixed*) is again automatic.

So given

```
F, G, H: Frac;
```

we can write

```
F := F + G;
F := F * G;
F := 0.5 * F;
F := F + 0.5;
```

However, we cannot write

```
F := F * G * H;              -- illegal
```

but must explicitly state the intermediate type such as

```
F := Frac(F * G) * H;
```

As expected, we cannot write

```
F := 2 + F;                  -- illegal
```

but we can write

```
F := 2 * F;
```

because multiplication is defined between fixed point types and Integer.

It is often necessary to define our own fixed point operations especially for multiplication and division. We might for example wish the type to be saturated as for the type Sat_Int in Section 15.5. Thus we might write

```
function "*" (Left, Right: Frac) return Frac is
   pragma Unsuppress(Overflow_Check);
begin
   return Standard."*" (Left, Right);
exception
```

```
    when Constraint_Error =>
       if (Left > 0.0 and Right > 0.0) or (Left < 0.0 and Right < 0.0) then
          return Frac'Last;
       else
          return Frac'First;
       end if;
end "*";
```

and similar functions for addition, subtraction, and division (taking due care over division by zero and so on). This raises an interesting issue since it does not hide the predefined operations (as it would for an integer type) because the result type is Frac whereas the predefined operation has a result type of *universal_fixed*.

This could cause awkward ambiguities if it were not for a rule which says that the predefined operation for multiplication is not used if there is a user-defined operation for either operand type unless there is an explicit conversion of the result or we write Standard."*". A similar rule applies to division.

So we can write

```
H := F * G;                          -- user-defined *
H := Standard."*" (F, G);            -- predefined *
```

A good example of the use of fixed point types might be with measurements of different but related units. Suppose we declare three types TL, TA, TV representing lengths, areas and volumes. We use centimetres as the basic unit with an accuracy of 0.1 cm together with corresponding consistent units and accuracies for areas and volumes. We might declare

```
type TL is delta 0.1 range -100.0 .. 100.0;
type TA is delta 0.01 range -10_000.0 .. 10_000.0;
type TV is delta 0.001 range -1000_000.0 .. 1000_000.0;
for TL'Small use TL'Delta;
for TA'Small use TA'Delta;
for TV'Small use TV'Delta;

function "*" (Left: TL; Right: TL) return TA;
function "*" (Left: TL; Right: TA) return TV;
function "*" (Left: TA Right: TL) return TV;

function "/" (Left: TV; Right: TL) return TA;
function "/" (Left: TV; Right: TA) return TL;
function "/" (Left: TA; Right: TL) return TL;

XL, YL: TL;
XA, YA: TA;
XV, YV: TV;
```

These types have an explicit *small* equal to their delta and are such that no scaling is required to implement the appropriate multiplication and division operations.

Note that all three types have primitive user-defined multiplication and division operations even though, in the case of multiplication, TV only appears as a result

type. Thus the predefined multiplication or division with any of these types as operands can only be considered if the result has a type conversion (or we explicitly mention Standard).

As a consequence the following are legal

```
XV := XL * XA;          -- legal, volume = length × area
XL := XV / XA;          -- legal, length = volume ÷ area
```

but the following are not because they do not match the user-defined operations

```
XV := XL * XL;          -- illegal, volume ≠ length × length
XV := XL / XA;          -- illegal, volume ≠ length ÷ area
XL := XL * XL;          -- illegal, length ≠ length × length
```

But if we insist on multiplying two lengths together then we can use an explicit conversion thus

```
XL := TL(XL * XL);      -- legal, predefined operation
```

and this uses the predefined operation.

If we need to multiply three lengths to get a volume without storing an intermediate area then we can write

```
XV := XL * XL * XL;
```

and this is unambiguous since there are no explicit conversions and so the only relevant operations are those we have declared.

As another example we return to the package Complex_Numbers of Section 12.2 and consider how we might implement the package body using a polar representation. Any reader who gave thought to the problem will have realized that writing a function to normalize an angle expressed in radians to lie in the range 0 to 2π using floating point raises problems of accuracy since 2π is not a representable number.

An alternative approach is to use a fixed point type. We can then arrange for π to be exactly represented. Another natural advantage is that fixed point types have uniform absolute error which matches the physical behaviour. The type Angle declared earlier is not quite appropriate because, as we shall see, it will be convenient to allow for angles of up to 4π.

The private part of the package could be

```
private
   π: constant := Ada.Numerics.π;
   type Angle is delta 0.1 range -4*π .. 4*π;
   for Angle'Small use π * 2.0**(-13);
   type Complex is
      record
         R: Real;
         θ: Angle range -π .. π;
      end record;
   I: constant Complex := (1.0, 0.5*π);
end;
```

where we have taken the opportunity to use the portable floating point type Real which seems more appropriate for this example.

We now need a function to normalize an angle to lie within the range of the component θ (which is neater if symmetric about zero). This is a nuisance but cannot be avoided if we use fixed point types. The function might be

```
function Normal(ϕ: Angle) return Angle is
begin
   if ϕ >= π then
      return ϕ – 2*π;
   elsif ϕ < –π then
      return ϕ + 2*π;
   else
      return ϕ;
   end if;
end Normal;
```

Another interesting point is that the values for θ that we are using do not include the upper bound of the range +π; the function Normal converts this into the equivalent –π. Unfortunately we cannot express the idea of an open bound in Ada although it would be perfectly straightforward to implement the corresponding checks.

The range for the type Angle has to be such that it will accommodate the sum of any two values of θ. However, making the lower bound of the range for Angle equal to –2*π is not adequate since there is no guarantee that the lower bound will be a represented number – it will not be in a one's complement implementation. So for safety's sake we squander a bit on doubling the range.

The various functions in the package body can now be written assuming that we have access to appropriate trigonometric functions applying to the fixed point type Angle and returning results of type Real. So we might have

```
package body Complex_Numbers is
   function Normal ...   -- as above
   ...
   function "*" (X, Y: Complex) return Complex is
   begin
      return (X.R * Y.R, Normal(X.θ + Y.θ));
   end "*";
   ...
   function Rl_Part(X: Complex) return Real is
   begin
      return X.R * Cos(X.θ);
   end Rl_Part;
   ...
end Complex_Numbers;
```

where we have left the more complicated functions for the enthusiastic reader.

An alternative would of course be to use a modular type much like the type of Exercise 17.2(**4**) but in that case we would have to look after the scaling ourselves.

One might think that the ideal solution would be for Ada to have a form of fixed point types with cyclic properties. However, hardware typically works in binary and so if a full cycle (that is 360°) is represented exactly, then even the angle of an equilateral triangle cannot be represented exactly. Clearly, Babylonian arithmetic using base 60 is what we want!

There are delta constraints much like the digits constraints for floating point types; again they are obsolescent and only affect output formats.

The attributes of a fixed point subtype S of a type T include S'Delta which is the requested delta, D, and S'Small which is the smallest positive represented number, *small*. Machine_Radix, Machine_Rounds and Machine_Overflows also apply to all fixed point types. There are also the usual attributes S'First and S'Last which give the actual upper and lower bounds of the subtype.

Delta and Small are of type *universal_real*; First and Last are of type T.

Exercise 17.5

1 Given F of type Frac explain why we could write

```
F := 0.5 * F;
```

2 Write the following further function for the package Complex_Numbers implemented as in this section

```
function "**" (X: Complex; N: Integer) return Complex;
```

Remember that if a complex number z is represented in polar form (r, θ), then

$z^n \equiv (r, \theta)^n = (r^n, n\theta)$.

3 An alternative approach to the representation of angles in fixed point would be to hold the values in degrees. Rewrite the private part and the function Normal using a canonical range of 0.0 .. 360.0 for θ. Make the most of a 16-bit word but use a power of 2 for *small*.

17.6 Decimal types

Decimal types are used in specialized commercial applications and are dealt with in depth in the Information Systems annex which is outside the scope of this book. However, the basic syntax of decimal types is in the core language and it is therefore appropriate to give a very brief overview.

A decimal type is a form of fixed point type. The declaration provides a value of delta as for an ordinary fixed point type (except that in this case it must be a power of 10) and also prescribes the number of significant decimal digits. So we can write

```
type Money is delta 0.01 digits 14;
```

which will cope with values of some currency such as euros up to one trillion (billion for older Europeans) in units of one cent. This allows 2 digits for the cents and 12 for the euros so that the maximum allowed value is

€999,999,999,999.99

The usual operations apply to decimal types as to other fixed point types; they are more exactly prescribed than for ordinary fixed point types since accountants abhor rounding errors. For example, type conversion always truncates towards zero and if the result of a multiplication or division lies between two multiples of *small* then again it truncates towards zero. Furthermore, the Information Systems annex describes a number of special packages for decimal types including conversion to external format using picture strings. See Section 26.4.

The attributes Digits, Scale and Round apply to decimal types; see Appendix 1. The attributes which apply to ordinary fixed point types also apply to decimal types. For most machines, Machine_Radix will always be 2 but some machines support packed decimal types and for them Machine_Radix will be 10.

Checklist 17

Declare explicit types for increased portability.

Beware of overflow in intermediate expressions.

Use named numbers or typed constants as appropriate.

The type Integer is constrained but Float is not.

If in doubt consult a numerical analyst.

The attribute Mod was added in Ada 2005.

The attribute Machine_Rounding was added in Ada 2005.

The rules excluding the use of predefined fixed point multiplication and division in the presence of user-defined operations did not exist in Ada 95 (and it reverts to the situation in Ada 83).

New in Ada 2012

Aspects such as Small can be given by an aspect specification in Ada 2012.

18 Parameterized Types

In this chapter we describe the parameterization of types by what are known as discriminants. Discriminants are components which have special properties. All composite types other than arrays can have discriminants. In this chapter we deal with the properties of discriminants in general and their use with record types (both tagged and untagged); their use with task and protected types is discussed in Chapter 20.

Discriminants can be of a discrete type or an access type. In the latter case the access type can be a named access type or it can be anonymous. A discriminant of an anonymous access type is called an access discriminant by analogy with an access parameter.

We start by dealing with discrete discriminants of untagged types.

18.1 Discriminated record types

In the record types we have seen so far there was no formal language dependency between the components. Any dependency was purely in the mind of the programmer as for example in the case of the private type Stack in Section 12.4 where the interpretation of the array S depended on the value of the integer Top.

In the case of a discriminated record type, some of the components are known as discriminants and the remaining components can depend upon these. The discriminants can be thought of as parameterizing the type and the syntax reveals this analogy.

As a simple example, suppose we wish to write a package providing various operations on square matrices and that in particular we wish to write a function

Trace which sums the diagonal elements of a square matrix. We could contemplate using the type Matrix of Section 8.2

```
type Matrix is array (Integer range <>, Integer range <>) of Float;
```

but the function would then have to check that the matrix passed as an actual parameter was indeed square. We would have to write something like

```
function Trace(M: Matrix) return Float is
   Sum: Float := 0.0;
begin
   if M'First(1) /= M'First(2) or M'Last(1) /= M'Last(2) then
      raise Non_Square;
   end if;
   for I in M'Range loop
      Sum := Sum + M(I, I);
   end loop;
   return Sum;
end Trace;
```

This is somewhat unsatisfactory; we would prefer to use a formulation which ensured that the matrix was always square and had a lower bound of 1. We can do this using a discriminated type. Consider

```
type Square(Order: Positive) is
   record
      Mat: Matrix(1 .. Order, 1 .. Order);
   end record;
```

This is a record type having two components: the first, Order, is a discriminant of the discrete subtype Positive and the second, Mat, is an array whose bounds depend upon the value of Order.

Variables and constants of type Square are declared in the usual way but (like array bounds) a value of the discriminant must be given either explicitly as a constraint or from an initial value. Thus the following are permitted

```
M: Square(3);
M: Square(Order => 3);
M: Square := (3, (1 .. 3 => (1 .. 3 => 0.0)));
```

The value provided for the discriminant could be any dynamic expression but once the variable is declared its constraint cannot be changed. The initial value for M could be provided by an aggregate as shown; note that the number of components in the subaggregate depends upon the first component of the aggregate. This can also be dynamic so that we could declare a Square of order N and initialize it to zero by

```
M: Square := (N, (1 .. N => (1 .. N => 0.0)));
```

We could avoid repeating N by giving the initial value separately (remember that we cannot refer to an object in its own declaration). So we could also write

```
M: Square(N);
...
M := (M.Order, (M.Mat'Range(1) => (M.Mat'Range(2) => 0.0)));
```

If we attempt to assign a value to M which does not have the correct discriminant value then Constraint_Error will be raised.

We can now rewrite the function Trace as follows

```
function Trace(M: Square) return Float is
   Sum: Float := 0.0;
begin
   for I in M.Mat'Range loop
      Sum := Sum + M.Mat(I, I);
   end loop;
   return Sum;
end Trace;
```

There is now no way in which a call of Trace can be supplied with a non-square matrix. Note that the discriminant of the formal parameter is taken from that of the actual parameter in a similar way to the bounds of an array.

Discriminants generally have much in common with array bounds. Thus we can introduce subtypes and a formal parameter can be constrained as in

```
subtype Square_3 is Square(3);
...
function Trace_3(M: Square_3) return Float;
```

but then the actual parameter would have to have a discriminant value of 3; otherwise Constraint_Error would be raised.

The result of a function could be of a discriminated type and, like arrays, the result could be a value whose discriminant is not known until the function is called. So we can write a function to return the transpose of a square matrix

```
function Transpose(M: Square) return Square is
begin
   return R: Square(M.Order) do              -- extended return for fun
      for I in 1 .. M.Order loop
         for J in 1 .. M.Order loop
            R.Mat(I, J) := M.Mat(J, I);
         end loop;
      end loop;
   end return;
end Transpose;
```

A private type can have discriminants in the partial view and it must then be implemented as a type with corresponding discriminants (usually a record but it could be a task or protected type, see Section 20.10).

As an example, reconsider the type Stack in Section 12.4. We can overcome the problem that all the stacks had the same maximum length by making Max a discriminant.

```
package Stacks is
   type Stack(Max: Natural) is private;
   procedure Push(S: in out Stack; X: in Integer);
   procedure Pop(S: in out Stack; X out Integer);
   function "=" (S, T: Stack) return Boolean;
private
   type Integer_Vector is array (Integer range <>) of Integer;
   type Stack(Max: Natural) is
      record
         S: Integer_Vector(1 .. Max);
         Top: Integer := 0;
      end record;
end;
```

Each variable of type Stack now includes a discriminant component giving the maximum stack size. When we declare a stack we must supply the value thus

```
ST: Stack(Max => 100);
```

and as in the case of the type Square the value of the discriminant cannot later be changed. Note that the discriminant is visible and can be referred to as ST.Max although the remaining components are private.

The body of the package Stacks remains as before (see Section 12.4). Note that the function "=" can be used to compare stacks with different values of Max since it only compares those components of the internal array which are in use.

Discriminants bear a resemblance to subprogram parameters in several respects. For example, the subtype of a discriminant must not have an explicit constraint. This is so that the same full conformance rules can be used when a discriminant specification has to be repeated in the case of a private type with discriminants, as illustrated by the type Stack above.

We can also have deferred constants of a discriminated type and they follow similar rules to array types as discussed in Section 12.2. The deferred constant need not provide a discriminant but if it does then it must statically match that in the full constant declaration.

Suppose we wish to declare a constant Stack with a discriminant of 3. We can omit the discriminant in the visible part and merely write

```
C: constant Stack;
```

and then give the discriminant in the private part either as a constraint or through the mandatory initial value (or both)

```
C: constant Stack(3) := (3, (1, 2, 3), 3);
```

It is possible to declare a type with several discriminants. We may for instance wish to manipulate matrices which although not constrained to be square nevertheless have both lower bounds of 1. This could be done by

```
type Rectangle(Rows, Columns: Positive) is
   record
      Mat: Matrix(1 .. Rows, 1 .. Columns);
   end record;
```

and we could then declare either of

```
R: Rectangle(2, 3);
R: Rectangle(Rows => 2, Columns => 3);
```

The usual rules apply: positional values must be in order, named ones may be in any order, mixed notation can be used but positional ones must come first.

Similarly to multidimensional arrays, a subtype must supply all the constraints or none at all. We could not declare

```
subtype Row_3 is Rectangle(Rows => 3);   -- illegal
```

in order to get the equivalent of

```
type Row_3(Columns: Positive) is
   record
      Mat: Matrix(1 .. 3, 1 .. Columns);
   end record;
```

although a similar effect can be obtained with derived types (see Section 18.4).

The above examples have used discriminants as the upper bounds of arrays; they can also be used as lower bounds. In Section 18.3 we will see that they can also be used to introduce a variant part. In all these cases a discriminant must be used directly and not as part of a larger expression. So we could not declare

```
type Symmetric_Array(N: Positive) is
   record
      A: Vector(-N .. N);                    -- illegal
   end record;
```

where the discriminant N is part of the expression -N. But a discriminant can also be used in a default initial expression in which case it need not stand alone.

Exercise 18.1

1 Suppose that M is an object of the type Matrix. Write a call of the function Trace whose parameter is an aggregate of type Square in order to determine the trace of M. What would happen if the two dimensions of M were unequal?

2 Rewrite the specification of Stacks to include a constant Empty in the visible part. See also Exercise 12.4(**1**).

3 Write a function Is_Full for Stacks. See also Exercise 12.4(**2**).

4 Declare a constant Square of order N and initialize it to a unit matrix. Use the function Make_Unit of Exercise 10.1(**6**).

18.2 Default discriminants

The discriminant types we have encountered so far have been such that once a variable is declared, its discriminant cannot be changed just as the bound of an array cannot be changed. It is possible, however, to provide a default expression for a discriminant and the situation is then quite different. A variable can then be

declared with or without a discriminant constraint. If one is supplied then that value overrides the default and as before the discriminant cannot be changed. If, on the other hand, a variable is declared without a value for the discriminant, then the value of the default expression is taken but it can then be changed by a complete record assignment. A type with default discriminants is said to be mutable because unconstrained variables can change shape or mutate.

An important variation occurs in the case of tagged types. Defaults for discriminants are only permitted in the case of tagged types if they are limited. This is because we do not want tagged types to be mutable but on the other hand limited types are not mutable anyway so they are allowed to have default discriminants.

A record type with default discriminants is an example of a definite type whereas one without defaults is an indefinite type. Remember that we can declare (uninitialized) objects and arrays of a definite type but not of an indefinite type, see Section 8.5.

Suppose we wish to manipulate polynomials of the form

$$P(x) = a_0 + a_1x + a_2x^2 + \ldots a_nx^n$$

where $a_n \neq 0$ if $n \neq 0$.

Such a polynomial could be represented by

```
type Poly(N: Index) is
   record
      A: Integer_Vector(0 .. N);
   end record;
```

where

```
subtype Index is Integer range 0 .. Max;
```

but then a variable of type Poly would have to be declared with a constraint and would thereafter be a polynomial of that fixed size. This would be most inconvenient because the sizes of the polynomials may be determined as the consequences of elaborate calculations. For example, if we subtract two polynomials which have $n = 3$, then the result will only have $n = 3$ if the coefficients of x^3 are different.

However, if we declare

```
type Polynomial(N: Index := 0) is
   record
      A: Integer_Vector(0 .. N);
   end record;
```

then we can declare variables

```
P, Q: Polynomial;
```

which do not have constraints. The initial value of their discriminants would be zero because the default value of N is zero but the discriminants could later be changed by assignment. Note however that a discriminant can only be changed by a complete record assignment. So

```
P.N := 6;
```

would be illegal. This is quite natural since we cannot expect the array P.A to adjust its bounds by magic.

Variables of the type Polynomial could be declared with constraints

```
R: Polynomial(5);
```

but R would thereafter be constrained for ever to be a polynomial with $n = 5$.

Initial values can be given in declarations in the usual way

```
P: Polynomial := (3, (5, 0, 4, 2));
```

which represents $5 + 4x^2 + 2x^3$. Note that despite the initial value, P is not constrained.

In practice we might make the type Polynomial a private type so that we could enforce the rule that $a_n \neq 0$. Observe that predefined equality is satisfactory. We can give the discriminant in the partial view and then both the private type declaration and the full type declaration must give the default expression for N.

Note once more the similarity to subprogram parameters; the default expression is only evaluated when required and so need not produce the same value each time. Moreover, the same conformance rules apply when it has to be written out again in the case of a private type.

We can alternatively choose that the partial view does not show the discriminant at all. We might write

```
package Polynomials is
   type Polynomial is private;
   ...
private
   type Polynomial(N: Index := 0) is ...
   ...
end;
```

and then of course we cannot declare constrained polynomials outside the package because the discriminant is not visible. Of course it is vital that the full type has defaults so that the type is definite.

If we declare functions such as

```
function "–" (P, Q: Polynomial) return Polynomial;
```

then it will be necessary to ensure that the result is normalized so that a_n is not zero. This could be done by the following function

```
function Normal(P: Polynomial) return Polynomial is
   Size: Integer := P.N;
begin
   while Size > 0 and P.A(Size) = 0 loop
      Size := Size – 1;
   end loop;
   return (Size, P.A(0 .. Size));
end Normal;
```

This is a further illustration of a function returning a value whose discriminant is not known until it is called. Note the use of the array slice.

If default expressions are supplied then they must be supplied for all discriminants of the type. Moreover an object must be fully constrained or not at all; we cannot supply constraints for some discriminants and use the defaults for others.

The attribute Constrained can be applied to an object of a discriminated type and gives a Boolean value indicating whether the object is constrained or not. For any object of types such as Square and Stack which do not have default values for the discriminants this attribute will, of course, be True. But in the case of objects of a type such as Polynomial which does have a default value, the attribute may be True or False. So, using P, Q and R declared above

```
P'Constrained = False
Q'Constrained = False
R'Constrained = True
```

We mentioned above that an unconstrained formal parameter will take the value of the discriminant of the actual parameter. In the case of an **out** or **in out** parameter, the formal parameter will be constrained if the actual parameter is constrained (an **in** parameter is constant anyway). Suppose we declare a procedure to truncate a polynomial by removing its highest order term

```
procedure Truncate(P: in out Polynomial) is
begin
   P := (P.N–1, P.A(0 .. P.N–1));
end Truncate;
```

Then Truncate(Q); will be successful, but Truncate(R); will result in Constraint_Error being raised. (We will also get Constraint_Error if we try to remove the only term of an unconstrained polynomial.)

We have seen that a discriminant can be used as the bound of an array. It can also be used as the discriminant constraint of an inner component. We could declare a type representing rational polynomials (that is one polynomial divided by another) by

```
type Rational_Polynomial(N, D: Index := 0) is
   record
      Num: Polynomial(N);
      Den: Polynomial(D);
   end record;
```

The relationship between constraints on the rational polynomial as a whole and its component polynomials is interesting. If we declare

```
R: Rational_Polynomial(2, 3);
```

then R is constrained for ever and the components R.Num and R.Den are also permanently constrained with constraints 2 and 3 respectively. However

```
P: Rational_Polynomial := (2, 3, Num => (2, (–1, 0, 1)),
                                 Den => (3, (–1, 0, 0, 1)));
```

is not constrained. This means that we can assign complete new values to P with different values of N and D. The fact that the components Num and Den are declared as constrained does not mean that P.Num and P.Den must always have a fixed length but simply that for given N and D they are constrained to have the appropriate length. So we could not write

```
P.Num := (1, (1, 1));
```

because this would violate the constraint on P.Num. However, we can write

```
P := (1, 2, Num => (1, (1, 1)), Den => (2, (1, 1, 1)));
```

because this changes everything together. Of course we can always make a direct assignment to P.Num that does not change the current value of its own discriminant.

The original value of P represented $(x^2 - 1)/(x^3 - 1)$ and the final value represents $(x + 1)/(x^2 + x + 1)$ which happens to be the same with the common factor of $(x - 1)$ cancelled. The reader will note the strong analogy between the type Rational_Polynomial and the type Rational of Exercise 12.2(**3**). We could write an equivalent function Normal to cancel common factors of the rational polynomials and the whole package of operations would then follow.

Another possible use of a discriminant is as part of the expression giving a default initial value for one of the other record components (but not another discriminant). Although the discriminant value may not be known until an object is declared, this is not a problem since the default initial expression is only evaluated when the object is declared and no other initial value is supplied. A discriminant can also be used to introduce a variant part as described in the next section.

However, we cannot use a discriminant for any other purpose. This unfortunately meant that when we declared the type Stack in the previous section we could not continue to apply the constraint to Top by writing

```
type Stack(Max: Natural) is
   record
      S: Integer_Vector(1 .. Max);
      Top: Integer range 0 .. Max := 0;        -- illegal
   end record;
```

since the use of Max in the range constraint is not allowed.

We conclude this section by reconsidering the problem of variable length strings and ragged arrays previously discussed in Sections 8.5, 11.2 and 11.4. Yet another approach is to use discriminated records. There are a number of possibilities such as

```
subtype String_Size is Integer range 0 .. 80;

type V_String(N: String_Size := 0) is
   record
      S: String(1 .. N);
   end record;
```

The type V_String is very similar to the type Polynomial (the lower bound is different). We have chosen a maximum string size corresponding to a typical page width (or historic punched card).

We can now declare fixed or varying v-strings such as

```
V: V_String := (5, "Hello");
```

We can overcome the burden of having to specify the length by writing

```
function "+" (S: String) return V_String is
begin
   return (S'Length, S);
end "+";
```

and then

```
type V_String_Array is array (Positive range <>) of V_String;
Zoo: constant V_String_Array := (+"aardvark", +"baboon",
                                 +"camel", +"dolphin", +"elephant", ..., +"zebra");
```

Since v-strings have default discriminants they are definite and so we can declare unconstrained v-strings and also arrays of them. However, there is a limit of 80 on v-strings and, moreover, the storage space for the maximum size string is likely to be allocated irrespective of the actual string.

The reader might feel bemused by the number of different ways in which variable length strings can be handled. There are indeed many conflicting requirements and in order to promote portability a number of predefined packages for manipulating strings are provided as we shall see in Section 23.3.

Exercise 18.2

1 Declare a Polynomial representing zero (that is, $0x^0$).

2 Write a function "*" to multiply two polynomials.

3 Write a function "–" to subtract one polynomial from another. Use the function Normal.

4 Rewrite the procedure Truncate to raise Truncate_Error if we attempt to truncate a constrained polynomial.

5 What would be the effect of replacing the discriminant of the type Polynomial by (N: Integer := 0)?

6 Rewrite the declaration of the type Polynomial so that the default initial value of a polynomial of degree n represents x^n. Hint: use & in the initial value.

7 Write the specification of a package Rational_Polynomials. Make the type Rational_Polynomial private with visible discriminants. The functions should correspond to those of the package Rational_Numbers of Exercise 12.2(**3**).

8 Write a function "&" to concatenate two v-strings.

18.3 Variant parts

It is sometimes convenient to have a record type in which part of the structure is fixed for all objects of the type but the remainder can take one of several different forms. This can be done using a variant part and the choice between the alternatives is governed by the value of a discriminant.

In many ways the use of a variant is an alternative to a tagged type where the hidden tag plays the role of the discriminant. For example in Section 14.4 we had an abstract type Person and derived types Man and Woman; we can reformulate this using a variant part as follows

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

type Person(Sex: Gender) is
   record
      Birth: Date;
      case Sex is
         when Male =>
            Bearded: Boolean;
         when Female =>
            Children: Integer;
      end case;
   end record;
```

This declares a record type Person with a discriminant Sex. The component Birth of type Date (see Section 8.7) is common to all objects of the type. However, the remaining components depend upon Sex and are declared as a variant part. If the value of Sex is Male then there is a further component Bearded whereas if Sex is Female then there is a component Children.

Since no default expression is given for the discriminant, all objects of the type must be constrained either explicitly or from an initial value. We can therefore declare

```
John: Person(Male);
Barbara: Person(Female);
```

or we can introduce subtypes and so write

```
subtype Man is Person(Sex => Male);
subtype Woman is Person(Sex => Female);
John: Man;
Barbara: Woman;
```

Aggregates take the usual form but, of course, give only the components for the corresponding alternative in the variant. The value for a discriminant governing a variant must be static so that the compiler can check the consistency of the aggregate. We can therefore write

```
John := (Male, (19, Aug, 1937), False);
Barbara := (Female, (13, May, 1943), 2);
```

but not

```
S: Gender := Female;
...
Barbara := (S, (13, May, 1943), 2);
```

because S is not static but a variable.

The components of a variant can be accessed in the usual way. We could write

```
John.Bearded := True;
Barbara.Children := Barbara.Children + 1;
```

but an attempt to access a component of the wrong alternative such as John.Children would raise Constraint_Error.

Note that although the sex of an object of type Person cannot be changed, it need not be known at compile time. We could have

```
S: Gender := ...
...
Chris: Person(S);
```

where the sex of Chris is not determined until he or she is declared. The rule that a discriminant must be static applies only to aggregates.

The variables of type Person are necessarily constrained because the type has no default expression for the discriminant. It is therefore not possible to assign a value which would change the sex; an attempt to do so would raise Constraint_Error. However, as with the type Polynomial, we could declare a default initial expression for the discriminant and consequently declare unconstrained variables. Such unconstrained variables could then be assigned values with different discriminants but only by a complete record assignment.

We could therefore have

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

type Mutant(Sex: Gender := Neuter) is
   record
      Birth: Date;
      case Sex is
         when Male =>
            Bearded: Boolean;
         when Female =>
            Children: Integer;
         when Neuter =>
            null;
      end case;
   end record;
```

Note that we have to write **null**; as the alternative in the case of Neuter where we do not want any components. In a similar way to the use of a null statement in a case statement this indicates that we really meant to have no components and did not omit them by accident.

We can now declare

```
The_Thing: Mutant;
```

The sex of this unconstrained mutant is neuter by default but can be changed by a whole record assignment.

Note the difference between

```
The_Thing: Mutant := (Neuter, (1, Jan, 1984));
```

and

```
It: Mutant(Neuter) := (Neuter, (1, Jan, 1984));
```

In the first case the mutant The_Thing is not constrained but just happens to be initially neuter. In the second case the object It is permanently neuter. This example also illustrates the form of the aggregate when there are no components in the alternative; there are none so we write none – we do not write **null**.

The rules regarding the alternatives closely follow those regarding the case statement described in Section 7.2. Each **when** is followed by one or more choices separated by vertical bars and each choice is either a simple expression or a discrete range. The choice **others** can also be used but must be last and on its own. All values and ranges must be static and all possible values of the discriminant must be covered once and once only. The possible values of the discriminant are those of its static subtype (if there is one) or type. Each alternative can contain several component declarations or can be null as we have seen.

A record can only contain one variant part and it must follow other components. However, variants can be nested; a component list in a variant part could itself contain a variant part but again it must follow other components in the alternative.

Also observe that it is unfortunately not possible to use the same identifier for components in different alternatives of a variant – all components of a record must have distinct identifiers.

It is worth emphasizing the rules regarding the changing of discriminants. If an object is declared with a discriminant constraint then the constraint cannot be changed – after all it is a constraint just like a range constraint and so the discriminant must always satisfy the constraint. Because the constraint allows only a single value this naturally means that the discriminant can only take that single value and so cannot be changed.

The other basic consideration is that, for implementation reasons, all objects must have values for discriminant components. Hence if the type does not provide a default initial expression, the object declaration must and since it is expressed as a constraint the object is then consequently constrained.

There is a restriction on renaming components of a variable of a discriminated type. If the existence of the component depends upon the value of a discriminant then it cannot be renamed if the variable is unconstrained. (This only applies to variables and not to constants.) So we cannot write

```
Hairy: Boolean renames The_Thing.Bearded;          -- illegal
```

because there is no guarantee that the component Bearded of the mutant The_Thing will continue to exist after the renaming even if it does exist at the moment of renaming. However,

```
Offspring: Integer renames Barbara.Children;
```

is valid because Barbara is a person and cannot change sex.

A similar restriction applies to the use of the Access attribute. Assuming the component Children is marked as aliased then Barbara.Children'Access is permitted whereas The_Thing.Children'Access is not.

Note, amazingly, that we can write

```
Bobby: Man renames Barbara;
```

because the constraint in the renaming declaration is ignored (see Section 12.7). Barbara has not had a sex change – she is merely in disguise!

Observe that Person is indefinite but Man, Woman and Mutant are definite. So we can declare arrays of Mutant, Man and Woman, but not of Person.

Although Person is indefinite we can declare a function with return type of Person and actual persons returned will be of a specific type such as Man or Woman. We can use extended return statements and write (discounting the mutant)

```
function Frankenstein(G: Gender; ... ) return Person is
begin
   if G = Male then
      return M: Person(Sex => Male) do
         ...                                   -- make a man
      end return;
   else
      return F: Person(Sex => Female) do
         ...                                   -- make a woman
      end return;
   end if;
end Frankenstein;
```

It is very instructive to consider how the types of this section might be rewritten using tagged types. The differences stem from the fact that tagged type derivation gives rise to distinct types whereas variants are simply different subtypes of the same type. The key differences using tagged types are

- An aggregate for a tagged type would not give the sex since it is inherent in the type.
- Attempting to access the wrong component John.Children is a compile-time error with tagged types whereas it raises Constraint_Error with variants.
- The secretive Chris of unknown sex could be of a class wide type and then the sex would be dynamically determined by the mandatory initial value; this is not quite so flexible because we need a whole initial value rather than just the constraint. So we have to copy another person or we have to write an extension aggregate such as (Person **with** Children => 0).
- The Mutant has no corresponding formulation using tagged types because a mutation would correspond to changing the tag and this can never happen.
- The restriction on renaming does not apply to tagged types because the tag of an object never changes and it is the tag which determines which components are present.

In deciding whether it is appropriate to use a variant or a tagged type, a key consideration is mutability. If an object can change its shape at run time then a

variant must be used. If an individual object can never change its shape then a tagged type is possible. A tagged type is better if the root operations are fixed but the categories might be extended. On the other hand, if the number of categories is fixed but more operations might be added then a variant formulation might be better. Thus adding a new operation such as Name to the root package Geometry of Program 1 requires everything to be recompiled.

Exercise 18.3

1 Write a procedure Shave which takes an object of type Person and removes any beard if the object is male and raises Shaving_Error if the object is female.

2 Write a procedure Sterilize which takes an object of type Mutant and ensures that its sex is Neuter by changing it if necessary and possible and otherwise raises an appropriate exception.

3 Declare a discriminated type Object which describes geometrical objects which are either a circle, a triangle or just a point. Use the same properties as for the corresponding tagged types of Section 14.1. Then declare a function Area which returns the area of an Object.

4 Declare a discriminated type Reservation corresponding to the types of the subsonic reservation system of Section 14.3.

18.4 Discriminants and derived types

We now consider the interaction of discriminants with derived types. This is rather different according to whether the type is tagged or not and so to avoid confusion we consider the two cases separately starting with untagged types.

We can of course derive from a discriminated record such as Rectangle in Section 18.1 in the usual way by

```
type Another_Rectangle is new Rectangle;
```

and then the discriminants become inherited so that Another_Rectangle also has the two discriminants Rows and Columns.

We could also derive from a constrained subtype

```
type Square_4 is new Rectangle(Rows => 4, Columns => 4);
```

in which case Square_4 is a constrained subtype of some anonymous type with the two discriminants.

Of more interest are situations where the derived type has its own discriminants. In such a case the parent must be constrained and the new discriminants replace the old ones which are therefore not inherited. Moreover, each new discriminant must be used to constrain one or more discriminants of the parent. So the new type may have fewer discriminants than its parent but cannot have more. Remember that extension cannot occur with untagged types and so the implementation model is that each apparent new discriminant has to use the space of one of the discriminants of the parent. There are a number of interesting possibilities

```
type Row_3(Columns: Positive) is
        new Rectangle(Rows => 3, Columns => Columns);
type Square(N: Positive) is
        new Rectangle(Rows => N, Columns => N);
type Transpose(Rows, Columns: Positive) is
        new Rectangle(Rows => Columns, Columns => Rows);
```

where we have used named notation for clarity. In all these cases the names of the old discriminants are no longer available in the new view.

The type Row_3 has lost one degree of freedom since it no longer has a visible discriminant called Rows; we could of course convert to type Rectangle and then look at the Rows and find it was 3.

In the case of the type Square (we are almost back to the type Square of Section 18.1), the one new discriminant is used to constrain both old ones. Hence a conversion of a value of type Rectangle to type Square will check that the two discriminants of the rectangle are equal and raise Constraint_Error if they are not.

The type Transpose has the discriminants mapped in the reverse order. So if R is a Rectangle then R.Rows = Transpose(R).Columns; we can use the result of the conversion as a name and then select the discriminant. But of course the internal array is still the same and so this is not a very interesting example.

We now turn to a consideration of tagged types. Tagged record types can also have discriminants but they can only have defaults if limited. Thus tagged objects are never mutable. Again there is the basic rule that if the derived type has its own discriminants then these replace any old ones and so the parent type must be constrained. The new discriminants may be used to constrain the parent, but they need not. So in the tagged case, extension is allowed and the derived type may have more discriminants than its parent.

Thus an alternative implementation for the type Person might be based on

```
type Gender is (Male, Female);
type Person(Sex: Gender) is tagged
   record
      Birth: Date;
   end record;
```

and we can then extend with, for example

```
type Man is new Person(Male) with
   record
      Bearded: Boolean;
   end record;
```

The type Man naturally inherits the discriminant from Person in the sense that a man still has a component called Sex although of course it is constrained to be Male.

We could also extend from the unconstrained type and then the new type would inherit the old discriminant just as any other component

```
type Old_Person is new Person with
   record
      Pension: Money;
   end record;
```

and the Old_Person also has a component Sex (although it is not visible in the declaration but then neither is the Birth component). We cannot declare an object of type Old_Person without a constraint.

As an example of providing new discriminants (in which case the parent type must be constrained) we might declare a type Boxer who must be male (females don't box in this model of the world) and then add a discriminant giving his weight

```
type Weight is (Light, Middle, Heavy);

type Boxer(W: Weight) is new Person(Male) with
   record
      ...   -- information according to weight perhaps as variant
   end record;
```

In this case the Boxer does not have a sex component at all since the discriminant has been replaced.

Other examples are provided by the type Object and its descendants. A useful guide is that discriminants make sense for controlling structure but not for other aspects of a type. We might have

```
type Regular_Polygon(No_Of_Sides: Natural) is new Object with
   record
      Side: Float;
   end record;
...
type Pentagon is new Regular_Polygon(5) with null record;
```

There is a limit to the amount of specialization that can be performed through syntactic structures. Thus, one might think it would be nice to write

```
type Polygon(No_Of_Sides: Natural) is new Object with
   record
      Sides: Float_Array(1 .. No_Of_Sides);
   end record;
```

but of course this is no good because we also need the values of the angles. But it is unwise to make such details directly visible to the user since direct assignment to the components could result in inconsistent values which did not represent a closed polygon. So it is probably better to write

```
package Geometry.Polygons is
   type Polygon(No_Of_Sides: Natural) is new Object with private;
   ...
private
   type Polygon(No_Of_Sides: Natural) is new Object with
      record
         Sides: Float_Array(1 .. No_Of_Sides);
         Angles: Float_Array(1 .. No_Of_Sides);
      end record;
end Geometry.Polygons;
```

As discussed in Section 14.6 we now need a constructor function which is best declared as a child function Geometry.Polygons.Make_Polygon. This should check that the angles and sides given as parameters do form a closed polygon.

We might then declare

```
package Geometry.Polygons.Four_Sided is
   type Quadrilateral is new Polygon(4) with private;
   type Parallelogram is new Quadrilateral with private;
   type Square is new Parallelogram with private;
   ...
private
```

which reveals the hierarchy of the different kinds of quadrilaterals. The full type declarations all have null extensions of course. This structure ensures that operations that apply to the class of all parallelograms also apply to squares.

Exercise 18.4

1 Declare the type Boxer so that it can be of either sex.

2 Declare a child function to make a quadrilateral. Assume the existence of

```
function Geometry.Polygons.Make_Polygon(Sides, Angles: Float_Array)
                                                    return Polygon;
```

3 Declare a package containing conversion functions between the various types of quadrilateral.

18.5 Access types and discriminants

Access types can refer to discriminated record types in much the same way that they can refer to array types. In both cases they can be constrained or not. Consider the problem of representing a family tree. We could declare

```
type Person;
type Person_Name is access all Person;

type Person is limited
   record
      Sex: Gender;
      Birth: Date;
      Spouse: Person_Name;
      Father: Person_Name;
      First_Child: Person_Name;
      Next_Sibling: Person_Name;
   end record;
```

This model assumes a monogamous and legitimate system. The children are linked together through the component Next_Sibling and a person's mother is identified as the spouse of the father. (The reader could consider how the model might be altered to accommodate the complexity of more flexible social systems.)

Although the above type Person is adequate for the model, it is more interesting to use a discriminated type so that different components can exist for the different sexes and more particularly so that appropriate constraints can be applied. Consider

```
type Person(Sex: Gender);
type Person_Name is access all Person;

type Person(Sex: Gender) is limited
   record
      Birth: Date;
      Father: Person_Name(Male);
      Next_Sibling: Person_Name;
      case Sex is
         when Male =>
            Wife: Person_Name(Female);
         when Female =>
            Husband: Person_Name(Male);
            First_Child: Person_Name;
      end case;
   end record;
```

The incomplete declaration of Person need not give the discriminants but if it does (as here) then it must also give any default initial expressions and the discriminants must conform to those in the subsequent complete declaration.

An important point is that we have made the type Person a limited type. This is because it would be quite inappropriate to copy a person although it is of course quite reasonable to copy the name of a person. We have also made the access type Person_Name general (using **all**) so that we can declare persons as ordinary variables if required as well as using the storage pool.

The component Father is now constrained always to access a person whose sex is male (could be null). Similarly the components Wife and Husband are constrained; note that these had to have distinct identifiers and so could not both be Spouse. However, the components First_Child and Next_Sibling are not constrained and so could access a person of either sex. We have also taken the opportunity to save on storage by making the children belong to the mother only.

When an object of type Person is created by an allocator, a value must be provided for the discriminant either through an explicit initial value as in

```
Janet: Person_Name;
...
Janet := new Person'(Female, (22, Feb, 1967), John, others => null);
```

or by supplying a discriminant constraint thus

```
Janet := new Person(Female);
```

Note that as in the case of arrays (see Section 11.3) a quote is needed in the case of the full initial value but not when we just give the constraint. Note also the use of **others** in the aggregate; this is allowed because the last three components all have the same base type. We could also write **others** => <>.

Naturally enough, the constraint cannot be omitted since the type Person does not have a default discriminant. For convenience we could introduce subtypes and so declare

```
subtype Man is Person(Male);
subtype Woman is Person(Female);
...
Janet := new Woman;
```

An allocated object cannot later have its discriminants changed except in one circumstance explained in Section 18.6. This general rule applies even if the discriminants have default initial expressions; objects created by an allocator are in this respect different from objects created by a normal declaration where having defaults for the discriminants always allows unconstrained objects to be declared and later to have their discriminants changed.

On the other hand, we see that despite the absence of a default initial expression for the discriminant, we can nevertheless declare unconstrained objects of type Person_Name; such objects, of course, take the default initial value **null** and so no problem arises. Thus although an allocated object cannot generally have its discriminants changed, nevertheless an unconstrained access variable could refer from time to time to objects with different discriminants.

The reason for not normally allowing an allocated object to have its discriminants changed is that it could be accessed from several constrained objects such as the components Father and it would be difficult to ensure that such constraints were not violated. Another similar rule is that we cannot have general access subtypes with constraints (as opposed to pool specific types) referring to types that have defaulted discriminants. So we cannot write

```
type Mutant_Name is access all Mutant;
subtype Things_Name is Mutant_Name(Sex => Neuter);      -- illegal
```

where the type Mutant is as in Section 18.3. For examples of the problems that this and other related rules solve see the *Ada 2005 Rationale*.

But there are no problems with Person_Name since the type Person does not have a default for its discriminant. So we can write

```
subtype Mans_Name is Person_Name(Male);
subtype Womans_Name is Person_Name(Female);
```

We can now write a procedure to marry two people.

```
procedure Marry(Bride: not null Womans_Name;
                Groom: not null Mans_Name) is
begin
   if Bride.Husband /= null or Groom.Wife /= null then
      raise Bigamy;
   end if;
   Bride.Husband := Groom;
   Groom.Wife := Bride;
end Marry;
```

The constraints on the parameter subtypes ensure that an attempt to marry people of the same sex will raise Constraint_Error at the point of call. Similarly, the explicit null exclusions on the parameters ensure that an attempt to marry a nonexistent person will also result in Constraint_Error at the point of call. We should perhaps have included the null exclusions in the definitions of the subtypes Mans_Name and Womans_Name.

Remember that although **in** parameters are constants we can change the components of the accessed objects – we are not changing the values of Bride and Groom to access different objects. So this section should really have started with a sequence such as

```
John, Barbara: Person_Name;
...
John := new Person'(Male, (19, Aug, 1937), Gilbert, others => null);
Barbara := new Person'(Female, (13, May, 1943), Tom, others => null);
...
Marry(Barbara, John);
```

A function could return an access value as for example

```
function Spouse(P: Person_Name) return Person_Name is
begin
   case P.Sex is
      when Male =>
         return P.Wife;
      when Female =>
         return P.Husband;
   end case;
end Spouse;
```

The result of such a function call is treated as a constant object (see the end of Section 10.1) and can be directly used as part of a name so we can write

```
Spouse(P).Birth
```

to give the birthday of the spouse of P. We could even write

```
Spouse(P).Birth := Newdate;
```

but this is only possible because the function delivers an access value. It could not be done if the function actually delivered a value of type Person rather than Person_Name. Moreover, we cannot write Spouse(P) := Q; in an attempt to replace our spouse by someone else. Note also that

```
Spouse(P).all := Q.all;                    -- illegal
```

is also illegal because the type Person is limited and cannot be copied. We can copy the components individually since they are not limited but writing .**all** gives a dereference and is not the same as writing out the individual assignments. Of course we cannot change the sex of our spouse anyway.

The following function gives birth to a new child. We need the mother, the sex of the child and the date as parameters.

```
function New_Child(Mother: not null Womans_Name;
                   Boy_Or_Girl: Gender;
                   Birthday: Date) return Person_Name is
   Child: Person_Name;
begin
   if Mother.Husband = null then
      raise Out_Of_Wedlock;
   end if;
   Child := new Person(Boy_Or_Girl);
   Child.Birth := Birthday;
   Child.Father := Mother.Spouse;
   declare
      Last: Person_Name := Mother.First_Child;
   begin
      if Last = null then
         Mother.First_Child := Child;
      else
         while Last.Next_Sibling /= null loop
            Last := Last.Next_Sibling;
         end loop;
         Last.Next_Sibling := Child;
      end if;
   end;
   return Child;
end New_Child;
```

Observe that a discriminant constraint need not be static – the value of Boy_Or_Girl is not known until the function is called. As a consequence we cannot give the complete initial value with the allocator because we do not know which components to provide. Hence we allocate the child with just the value of the discriminant and then separately assign the date of birth and the father. The remaining components take the default value **null**. We finally link the new child onto the end of the chain of siblings which starts from the component First_Child of Mother. Note that a special case arises if the new child is the first born.

We can now write

```
Helen: Person_Name := New_Child(Barbara, Female, (28, Sep, 1969));
```

It is interesting to consider how the family saga could be rewritten using type extension and inheritance rather than discriminants. Clearly we can use an abstract root type Person and derived types Man and Woman. We also need three distinct access types. So

```
type Person is tagged;
type Man is tagged;
type Woman is tagged;
```

```
type Person_Name is access all Person'Class;
type Mans_Name is access all Man;
type Womans_Name is access all Woman;

type Person is abstract tagged limited
   record
      Birth: Date;
      Father: Mans_Name;
      Next_Sibling: Person_Name;
   end record;

type Man is limited new Person with
   record
      Wife: Womans_Name;
   end record;

type Woman is limited new Person with
   record
      Husband: Mans_Name;
      First_Child: Person_Name;
   end record;
```

In the declaration of Man and Woman, we can omit **limited** since limitedness is always derived from a parent type but it should be stated for clarity (it has to be stated if the parent is an interface – see Section 14.8).

In practice we would undoubtedly declare the various types (in either formulation) as private although keeping the relationships between the types visible. In the tagged case the visible part might become

```
package People is

   type Person is abstract tagged limited private;
   type Man is limited new Person with private;
   type Woman is limited new Person with private;

   type Person_Name is access all Person'Class;
   type Mans_Name is access all Man;
   type Womans_Name is access all Woman;

private
```

We could then declare all the various subprograms Marry, Spouse and so on inside this package or maybe in a child package which of course would have full visibility of the details of the types. We need to take care in starting the system because we cannot assign to the various components externally. The function New_Child cannot be used initially because it needs married parents. However, a sequence such as

```
Adam := new Man;
Eve := new Woman;
Marry(Eve, Adam);
Cain := New_Child(Eve, Male, Long_Ago);
```

seems to work although it leaves some components of Adam and Eve not set properly.

We might consider declaring the component Father in the type Person with a null exclusion on the grounds that everyone has a father and so save some run-time checks. But there is then the problem of declaring Adam and Eve since they must be given a non-null Father otherwise Constraint_Error will be raised.

This is the usual chicken and egg problem. A common programming technique is to make such a first object refer to itself. But this is almost impossible if we insist on using a null exclusion. For example we could not write

```
First_Man: Person_Name := new Person'
                                (Male, Some_Date, First_Man, others => <>);
```

because we cannot refer to an object in its own declaration. One way around it is to introduce a third variant which does not have a component Father and then the first person can be of this kind but that seems very artificial. Another way is to make every person by default have their father as themselves. This can be done by a dirty trick (that is, an interesting technique) as explained in Section 18.7.

We leave to the reader the task of writing the various subprograms using the tagged formulation. One advantage is that more checking is done at compile time because Man and Woman are distinct types rather than just subtypes of Person.

Note that we did not use anonymous access types in the above. This is because there would have been a forward reference between the type Person and the component Father. We could have used anonymous access types in the discriminant model but for ease of comparison they have both been treated the same way. Moreover, for either model, it seems very appropriate for this application to talk explicitly about names as well as persons. Observe also that we made the access types general (with **all**); this is so that we can convert between them – remember that conversion is not allowed between pool specific access types.

This brings us finally to the rules regarding type conversion between different general access types referring to the same discriminated record type. Conversion is permitted provided the accessed subtypes statically match (the usual rule) but might raise Constraint_Error if the target subtype is constrained.

Exercise 18.5

1 Write a function to return a person's heir (use the variant formulation). Follow the historical rules of primogeniture applicable to monarchies – the heir is the eldest son if there is one and otherwise is the eldest daughter. Return **null** if there is no heir.

2 Write a procedure to enable a woman to get a divorce. Divorce is only permitted if there are no children.

3 Modify the procedure Marry in order to prevent incest. A person may not marry their sibling, parent or child.

4 Rewrite Marry using the tagged type formulation; this is trivial.

5 Rewrite Spouse using the tagged type formulation. Hint: consider dispatching.

6 Rewrite New_Child using the tagged type formulation.

18.6 Private types and discriminants

In this section we bring together a number of matters relating to the control of resources through private types and discriminants.

In earlier sections we have seen how private types provide general control by ensuring that all operations on a type are through defined subprograms. Making a type also limited prevents assignment.

The introduction of discriminants adds other possibilities. We have seen that a partial view might show the discriminants as in the type Stack of Section 16.1

```
package Stacks is
   type Stack(Max: Natural) is private;
   ...
private
   type Stack(Max: Natural) is ...
```

and also that the partial view might hide them as with

```
package Polynomials is
   type Polynomial is private;
   ...
private
   type Polynomial(N: Index := 0) is ...
```

of Section 18.2 even though the full type has a discriminant; we noted that the full type has to be definite and so the discriminant has to have a default in this case.

Another possibility is that the partial view might have discriminants and then the full type could be implemented in terms of an existing type with discriminants thus

```
package Squares is
   type Square(N: Positive) is private;
   ...
private
   type Square(N: Positive) is new Rectangle(N, N);
```

using the examples of Section 18.4.

The final possibility is that the partial view might have unknown discriminants written

```
type T(<>) is private;
```

This view is considered indefinite and prevents the user from declaring uninitialized objects of the type; the partial view might also be tagged. If the writer of the package does not provide any means of initializing objects (through a function or deferred constant) then the user cannot declare objects at all.

Making the partial view limited (again it might also be tagged) by writing

```
type T(<>) is limited private;
```

prevents the user from copying objects as usual.

These forms enable the provider of the package to have complete control over all objects of the type; typically the user would be given access to objects of the type through an access type which is then often called a handle. We will consider some examples of controlling the privacy of abstractions in this way in Section 21.7.

If the partial view has unknown discriminants then the full view may or may not have discriminants; none were promised and so none need be provided. The full type can also be another type with unknown discriminants (such as being derived from one).

A class wide type is also treated as having unknown discriminants (the tag is a hidden discriminant). Remember that a class wide type is another example of an indefinite type so that uninitialized objects are not allowed.

An interesting situation occurs when the full type is mutable but the partial view does not reveal the discriminants. Consider

```
package Beings is
   type Mutant is private;
   type Mutant_Name is access Mutant;
   F, M: constant Mutant;
private
   type Mutant(Sex: Gender := Neuter) is
      record
         ...                    -- as in Section 18.3
      end record;

   F: constant Mutant := (Female, ... );
   M: constant Mutant := (Male, ... );
end Beings;
```

The user can now write

```
Chris: Mutant_Name := new Mutant'(F);      -- it's a girl, a copy of F
...
Chris.all := M;                            -- OK?  Yes! it's a boy
```

Note that we have now changed the sex of the allocated object referred to by Chris. This has to be allowed because the external view does not show that the constants M and F are different internally. This is the one situation referred to in Section 18.5 where an allocated object can have its discriminant changed.

We conclude this section with a curious example of discriminants by reconsidering the type Key of Section 12.6. We could change this to

```
type Key(Code: Natural := 0) is limited private;
```

with full type

```
type Key(Code: Natural := 0) is null record;
```

With this formulation the user can read the code number of the key, but cannot change it. There is, however, a small flaw whose detection and cure is left as an exercise. Note also that we have declared Code as of subtype Natural rather than of type Key_Code; this is because Key_Code is not visible to the user. Of course we

could make Key_Code visible but this would make Max visible as well and we might not want the user to know how many keys there are.

However, as observed in Exercise 14.7(**2**), it is far better to make the type tagged and controlled so that it can be initialized and finalized properly. The inquisitive user can always be given read access to the value of the code through a function.

Exercise 18.6

1 What is the flaw in the suggested new formulation for the type Key? Hint: remember that the user declares keys explicitly. Show how it can be overcome.

18.7 Access discriminants

A discriminant may also be of an access type. This enables a record (or task or protected object) to be parameterized with some other structure with which it is associated. The access type can be a named access type or it can be anonymous in which case the discriminant is known as an access discriminant. In both cases a default value might be provided.

Discriminants of a named access type are not particularly interesting; they behave much as other components of a record except that they can be visible even though the rest of the record might be private; but they have no controlling function in the way that discrete discriminants can control array bounds and variants.

Access discriminants have similar accessibility properties to access parameters – for example, they can refer to local variables. This gives extra flexibility and so most discriminants of access types are in fact access discriminants.

A typical structure might be

```
type Data is ...

type R(D: access Data) is
   record
      ...
   end record;
```

and then a declaration of an object of type R must include an access value to an associated object of type Data. Thus

```
The_Data: aliased Data := ... ;
The_Record: R(The_Data'Access, ... );
```

An important point is that the two objects are bound together permanently. On the one hand, if the discriminant does not have a default value then it cannot be changed anyway. On the other hand, if it does have a default value then the type has to be explicitly marked as **limited**. This means that since a discriminant can only be changed by a whole record assignment and assignment is forbidden for limited types, once more it cannot be changed. Thus either way the two objects are bound together permanently. See also Section 22.4 for coextensions.

Within the type R, the discriminant D could be used as a constraint for an inner component or as the default initial value of an inner component.

An interesting special case is where the object having the discriminant is actually a component of the type the discriminant refers to! This can be used to enable a component of a record to obtain the identity of the record in which it is embedded. Consider

```
type Inner(Ptr: access Outer) is limited ...

type Outer is limited
   record
      ...
      Component: Inner(Outer'Access);
      ...
   end record;
```

The Component of type Inner has an access discriminant Ptr which refers back to the instance of the record Outer. This is because the attribute Access applied to the name of a record type inside its declaration refers to the current instance. (When we deal with tasks we will see that a similar situation arises when the name of a task type is used inside its own body; see Section 22.1.) If we now declare an object of the type Outer

```
Obj: Outer;
```

then the structure created is as shown in Figure 18.1. We call it a self-referential structure for obvious reasons. All instances of the type Outer will refer to themselves.

An important example of the use of this feature will be found in Section 22.2 where a component of a record is a task with a discriminant.

Incidentally, we can also write

```
type Cell is tagged limited
   record
      Next: access Cell := Cell'Unchecked_Access;
      ...
   end record;
```

In this case whenever an object of the type Cell is created without an explicit initial value (whether by declaration or by an allocator) the component Next refers to the new object itself. Such a mechanism might be useful in automatically providing a dummy header cell in some linked list applications as for example when we were trying to make Father have a null exclusion in the family saga of Section 18.5. Sadly, the accessibility rules require us to use Unchecked_Access in this case.

Access discriminants work together with access parameters and thereby minimize accessibility problems. Suppose we had some general procedure to manipulate the type Data

```
procedure P(A: access Data);
```

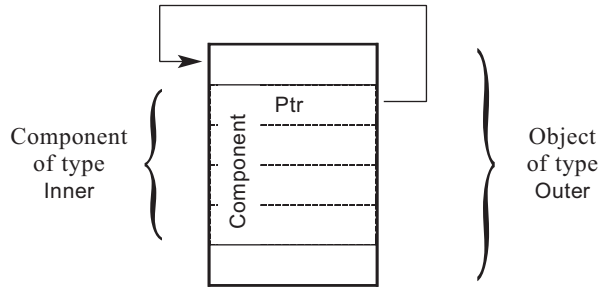


Figure 18.1 A self-referential structure.

then we can make calls such as in

```
declare
   My_Data: aliased Data := ...
   My_Record: R(My_Data'Access);
begin
   ...
   P(My_Record.D);                    -- legal
   ...
```

and the accessibility of the data is passed via the access parameter of P. On the other hand, using a named access type as the parameter of P thus

```
type Data_Ptr is access all Data;
procedure P(A: Data_Ptr);
```

fails because the type conversion required in the corresponding call

```
P(Data_Ptr(My_Record.D));          -- illegal
```

breaks the static accessibility rules of Section 11.5.

Examples of using access parameters with access discriminants will be found in Sections 21.5 and 22.4.

Exercise 18.7

1 Analyse the following and explain why the various attempts to assign the reference to the local data to the more global variable or component are thwarted.

```
procedure Main is
   type Data is ...
   type Data_Ptr is access all Data;
   type R(D: access Data) is limited
      record
         ...
      end record;
```

```
   Global_Ptr: Data_Ptr;
   Global_Data: aliased Data;
   Global_Record: R(Global_Data'Access);
begin
   declare
      Local_Data: aliased Data;
      Local_Record: R(Local_Data'Access);
   begin
      Global_Ptr := Local_Record.D;                -- incorrect
      Global_Ptr := Data_Ptr(Local_Record.D);      -- incorrect
      Global_Record := Local_Record;               -- incorrect
   end;
end Main;
```

Checklist 18

If a discriminant does not have a default expression then all objects must be constrained.

The discriminant of an unconstrained object can only be changed by a complete record assignment.

Discriminants can only be used as array bounds or to govern variants or as nested discriminants or in default initial expressions for components.

A discriminant in an aggregate and governing a variant must be static.

Any variant must appear last in a component list.

If an accessed object has a discriminant then it is always constrained.

Discriminants of tagged types can only have defaults if limited.

On derivation, discriminants are all inherited or completely replaced with a new set.

A record type with access discriminants must be explicitly limited or not have defaults for the discriminants.

Access discriminants have dynamic accessibility.

Conversion of a general access type to a subtype with a discriminant constraint was not permitted in Ada 95.

The rules preventing misuse of discriminants were somewhat different in Ada 95. General access types to a type with default discriminants were permitted in Ada 95 (and caused awkward problems).

Aliased objects are not automatically constrained in Ada 2005. Allocated objects are also not constrained in Ada 2005 if there is a partial view without discriminants.

Access discriminants were only permitted with limited types in Ada 95.

New in Ada 2012

Tagged record types can have default discriminants if the record type is limited.

19 Generics

In this chapter we describe the generic mechanism which allows a special form of parameterization at compile time which can be applied to subprograms and packages. The generic parameters can be types of various categories (access types, limited types, tagged types and so on), subprograms and packages as well as values and objects.

Genericity is important for reuse. It provides static polymorphism as opposed to the dynamic polymorphism provided by type extension and class wide types. Being static it is intrinsically more reliable but usually less flexible. However, in the case of subprogram parameters it does not have the convention restrictions imposed by access to subprogram types and thus is particularly useful where the parameter is an operation.

Package parameters enable the composition of generic packages while ensuring that their instantiations are compatible. They can also be used to group together the parameters of other generic units to provide signatures.

19.1 Declarations and instantiations

We often get the situation that the logic of a piece of program is independent of the types involved and it therefore seems unnecessary to repeat it for all the different types to which we might wish it to apply. A simple example is provided by the procedure Swap of Exercise 10.3(**1**)

```
procedure Swap(X, Y: in out Float) is
   T: Float;
begin
   T := X;  X := Y;  Y := T;
end;
```

It is clear that the logic is independent of the type Float. If we also wanted to swap integers or dates we could write other similar procedures but this would be tedious. The generic mechanism allows us to avoid this. We declare

```
generic
   type Item is private;
procedure Exchange(X, Y: in out Item);

procedure Exchange(X, Y: in out Item) is
   T: Item;
begin
   T := X;  X := Y;  Y := T;
end;
```

The subprogram Exchange is a generic subprogram and acts as a kind of template. The subprogram specification is preceded by the generic formal part consisting of the reserved word **generic** followed by a (possibly empty) list of generic formal parameters. The subprogram body is written exactly as normal but note that, in the case of a generic subprogram, we always have to give both the specification and the body separately.

A generic procedure is not a real procedure and so cannot be called; but from a generic procedure we can create an actual procedure by a mechanism known as generic instantiation. For example, we may write

```
procedure Swap is new Exchange(Float);
```

This is a declaration and states that Swap is obtained from the template given by Exchange. Actual generic parameters are provided in a parameter list in the usual way. The actual parameter in this case is the type Float corresponding to the formal parameter Item. We could also use the named notation

```
procedure Swap is new Exchange(Item => Float);
```

So we have now created the procedure Swap acting on type Float and can henceforth call it in the usual way. We can make further instantiations

```
procedure Swap is new Exchange(Integer);
procedure Swap is new Exchange(Date);
```

and so on. We are here creating further overloadings of Swap which can be distinguished by their parameter types just as if we had laboriously written them out in detail.

Superficially, it may look as if the generic mechanism is merely one of text substitution and indeed in simple cases the behaviour would be the same. However, an important difference relates to the meaning of identifiers in the generic body but which are neither parameters nor local to the body. Such nonlocal identifiers have meanings appropriate to where the generic body is declared and not to where it is instantiated. If text substitution were used then nonlocal identifiers would of course take their meaning at the point of instantiation and this could give very surprising results.

We may also have generic packages. A simple example is provided by the package Stack in Section 12.1. The trouble with that package is that it only works on type Integer although of course the same logic applies irrespective of the type of the values manipulated. We can also make Max a parameter so that we do not have an arbitrary limit of 100. We write

```
generic
   Max: Positive;
   type Item is private;
package Stack is
   procedure Push(X: Item);
   function Pop return Item;
end Stack;

package body Stack is
   S: array (1 .. Max) of Item;
   Top: Integer range 0 .. Max;
   ...   -- etc. as before but with Integer
   ...   -- replaced by Item
end Stack;
```

We can now create and use a stack of a particular size and type by instantiating the generic package as in the following

```
declare
   package My_Stack is new Stack(100, Float);
   use My_Stack;
begin
   ...
   Push(X);
   ...
   Y := Pop;
   ...
end;
```

The package My_Stack which results from the instantiation behaves just as a normal directly written out package. The use clause allows us to refer to Push and Pop directly. If we did a further instantiation

```
package Another_Stack is new Stack(50, Integer);
use Another_Stack;
```

then Push and Pop are further overloadings but can be distinguished by the parameter and result type. Of course, if Another_Stack were also declared with the actual generic parameter being Float, then we would have to use the dotted notation to distinguish the instances of Push and Pop despite the use clauses.

Both generic units and generic instantiations may be library units. Thus, having compiled the generic package Stack, an instantiation could itself be separately compiled just on its own thus

```
with Stack;
package Boolean_Stack is new Stack(200, Boolean);
```

If we added an exception Error to the package as in Section 15.2 so that the generic package declaration was

```
generic
   Max: Positive;
   type Item is private;
package Stack is
   Error: exception;
   procedure Push(X: Item);
   function Pop return Item;
end Stack;
```

then each instantiation would give rise to a distinct exception and because exceptions cannot be overloaded we would naturally have to use the dotted notation to distinguish them.

We could, of course, make the exception Error common to all instantiations by making it global to the generic package. It and the generic package could perhaps be declared inside a further package

```
package All_Stacks is
   Error: exception;
   generic
      Max: Positive;
      type Item is private;
   package Stack is
      procedure Push(X: Item);
      function Pop return Item;
   end Stack;
end All_Stacks;

package body All_Stacks is
   package body Stack is
      ...
   end Stack;
end All_Stacks;
```

This illustrates the binding of identifiers global to generic units. The meaning of Error is determined at the point of the generic declaration irrespective of the meaning at the point of instantiation.

The above examples have illustrated formal parameters which were types and also integers. In fact generic formal parameters can be values and objects much as the parameters applicable to subprograms; they can also be types, subprograms and packages. As we shall see in the next sections, we can express the formal types, subprograms and packages so that we can assume in the generic body that the actual parameters have the properties we require.

Object parameters can be of mode **in** or **in out** but not **out**. As in the case of parameters of subprograms, **in** is taken by default as illustrated by Max in the example above. Explicit constraints are not permitted but null exclusions are.

An **in** generic parameter acts as a constant whose value is provided by the corresponding actual parameter. A default expression is allowed as in the case of

parameters of subprograms; such a default expression is evaluated at instantiation if no actual parameter is supplied in the same way that a default expression for a subprogram parameter is evaluated when the subprogram is called if no actual parameter is supplied. Observe that an **in** generic parameter can be of a limited type; the actual parameter might then be an aggregate or constructor function call as explained in Section 12.5.

An **in out** parameter, however, acts as a variable renaming the corresponding actual parameter. The actual parameter must therefore be the name of a variable and its identification occurs at the point of instantiation using the same rules as for renaming described in Section 13.7. One such rule is that constraints on the actual parameter apply to the formal parameter and any constraints implied by the formal subtype mark are, perhaps surprisingly, completely ignored. Another rule is that if the value of any identifier which is part of the name subsequently changes then the identity of the object referred to by the generic formal parameter does not change. Because of this there is a restriction, like that on renaming, that the actual parameter cannot be a component of an unconstrained discriminated record if the very existence of the component depends on the value of the discriminant. Thus if The_Thing is a Mutant as in Section 18.3, The_Thing.Bearded could not be an actual generic parameter because The_Thing could have its Sex changed. However, The_Thing.Birth would be valid.

It will now be realized that although the notation **in** and **in out** is identical to subprogram parameters the meaning is somewhat different. Thus there is no question of copying in and out and indeed no such thing as **out** parameters.

Anonymous access types (both access to object and access to subprogram types) are also permitted for the type of the formal parameter so we can have parameters such as

```
generic
   A: access T := null;
   AN: in out not null access constant T;
   F: access function (X: Float) return Float;
   FN: not null access function (X: Float) return Float;
```

As usual, explicit null exclusions are permitted whereas explicit constraints are not. If the subtype of the formal object excludes null (as in AN and FN) then the actual must also exclude null but the reverse does not hold. Unlike access parameters of subprograms the mode can be **in** or **in out** as for any generic object parameter. Default expressions are permitted as illustrated for A and **constant** is permitted as illustrated for AN. The obvious matching rules apply such as subtype conformance in the case of F and FN.

Inside the generic body, the formal generic parameters can generally be used quite freely – but there is one important restriction. This arises because generic parameters (and their attributes) are not generally considered to be static. There are various places where an expression has to be static such as in the alternatives in a case statement or variant, and in the range in an integer type definition, or the number of digits in a floating point type definition and so on. In all these situations a generic formal parameter cannot be used because the expression would not then be static.

However, the type of the expression in a case statement and similarly the type of the discriminant in a variant may be a generic formal type provided there is an others clause – this ensures that all values are covered.

But a generic parameter is treated as static in both the visible and private part of a generic package if the actual parameter is static.

Our final example in this section illustrates the nesting of generics. The following generic procedure performs a cyclic interchange of three values and for amusement is written in terms of the generic procedure Exchange

```
generic
   type Thing is private;
procedure CAB(A, B, C: in out Thing);

procedure CAB(A, B, C: in out Thing) is
   procedure Swap is new Exchange(Item => Thing);
begin
   Swap(A, B);
   Swap(A, C);
end CAB;
```

Although nesting is allowed, it must not be recursive.

It is important to appreciate that generic subprograms are not subprograms and cannot be overloaded with subprograms or with each other. One consequence is that generic subprogram renaming does not require the parameter list to be repeated. So we can simply write

```
generic procedure Taxi renames CAB;
```

although perhaps confusing for this particular example!

But instantiations of generic subprograms are real subprograms and in the appropriate circumstances can be primitive operations. Thus if we wrote

```
package Dates is
   type Date is record ... end record;
   procedure Swap is new Exchange(Date);
```

then Swap would be a primitive operation of Date. We might even have

```
overriding
procedure Swap is new Exchange(Thing);
```

which means that Swap must be an overriding operation for the type Thing.

Another point is that generic subprograms can have preconditions and post-conditions. But instantiations of generic subprograms cannot have preconditions and postconditions. This is discussed in AI12-45 and so was added after ISO standardization.

Finally, note that the same rules for mixing named and positional notation apply to generic instantiation as to subprogram calls. Hence if a parameter is omitted, subsequent parameters must be given using named notation. Of course, a generic unit need have no parameters in which case the instantiation takes the same form as for a subprogram call – the parentheses are omitted.

Exercise 19.1

1 Write a generic package based on the package Stacks in Section 18.1 so that stacks of any type may be declared. Then declare a stack S of length 30 and type Boolean. Use named notation.

2 Write a generic package containing both Swap and CAB.

19.2 Type parameters

In the previous section we introduced types as generic parameters. The examples showed the formal parameter taking the form

```
type T is private;
```

In this case, inside the generic subprogram or package, we may assume that assignment and equality are defined for T and that T is definite so that we may declare uninitialized objects of type T. We can assume nothing else unless we specifically provide other parameters as we shall see in a moment. Hence T behaves in the generic unit much as a private type outside the package defining it; this analogy explains the notation for the formal parameter. The corresponding actual parameter must, of course, provide assignment and equality and so it can be any definite type except one that is limited. Note carefully that it cannot be an indefinite type such as String.

A formal generic type parameter can take many other forms. It can be

```
type T is limited private;
```

and in this case assignment and predefined equality are not available. The corresponding actual parameter can be any definite type, limited or nonlimited, tagged or untagged.

Either of the above forms could have unknown discriminants such as

```
type T(<>) is private;
```

in which case the type is considered indefinite within the generic unit and so uninitialized objects cannot be declared. The actual type could then be any nonlimited indefinite type such as String as well as any nonlimited definite type. It could also be a class wide type since these are considered to be indefinite.

Another possibility is that the formal type could have known discriminants such as

```
type T(X: U; Y: V; ... ) is private;
```

and the actual type must then have discriminants with statically matching subtypes. The formal type must not have default expressions for the discriminants but the actual type can. The formal type is therefore indefinite.

We can also require that the actual type be tagged by one of

```
type T is tagged private;
type T is abstract tagged private;
```

In the first case the actual type must be tagged but not abstract. The second case allows the actual type to be abstract as well and this includes the possibility of the actual type being an interface.

All the above forms can be put together in the obvious way, so an extreme example might be

```
type T(I: Index) is abstract tagged limited private;
```

in which case the actual type must have one discriminant statically matching the subtype Index, it must be tagged, it might be abstract, it might be limited.

Another possibility concerns incomplete types. Thus we might have the forms

```
type T;
type T is tagged;
```

and the actual type can then be an incomplete type. If the formal specifies **tagged** then the actual type must be tagged; if the formal does not specify **tagged** then the actual type might or might not be tagged. Discriminants can also be given and must match. Note moreover that an incomplete formal type can also be matched by an appropriate complete type.

The matching rules are expressed in terms of categories as outlined in Section 3.4. A category of types is just a set of types with common properties. The formal parameter defines the category and the actual can then be any type in that category.

Derivation classes are important categories and the forms

```
type T is new S;
type T is new S with private;
```

require that the actual type must be derived from S. The first form applies if S is not tagged and the second if S is tagged. We can also insert **abstract** after **is** in the second case. Within the generic unit all properties of the derivation class may be assumed. Unknown discriminants and limited can be added to both forms. (But known discriminants are not permitted with either of these forms.)

Most of the matching rules follow automatically from the derivation model but there are a number of additional rules. For example if S is an access type without a null exclusion then the actual type must not have a null exclusion.

A very important form of tagged type is an interface and these can also be used as generic parameters. Thus we might have

```
type F is interface;
```

The actual type could then be any interface. This is perhaps unlikely.

If we wanted to ensure that a formal interface had certain operations then we might first declare an interface A with the required operations

```
type A is interface;
procedure Op1(X: A; ... ) is abstract;
procedure N1(X: A; ... ) is null;
```

and then the generic formal parameter would be

```
type F is interface and A;
```

and then the actual interface must be descended from A and so have operations which match Op1 and N1.

A formal interface might specify several ancestors

```
type FAB is interface and A and B;
```

where A and B are themselves interfaces. And A and B or just some of them might themselves be further formal parameters as in

```
generic
   type A is interface;
   type FAB is interface and A and B;
```

This means that FAB must have both A and B as ancestors; it could of course have other ancestors as well. Note that this illustrates that one generic formal parameter can depend upon a previous formal parameter that is a type.

Formal tagged types might also be descended from interfaces. Thus we might have

```
generic
   type NT is new T and A and B with private;
```

in which case the actual type must be descended from the tagged type T and also from the interfaces A and B. The parent type T itself might be an interface or a normal tagged type. Again some or all of T, A, and B might be earlier formal parameters. Also we can explicitly state **limited** in which case all of the ancestor types must also be limited.

As we shall see in Chapter 22, interfaces can also be restricted to synchronized types such as task types and protected types and special forms of generic formal parameter apply in these cases.

The formal parameter could also be one of

```
type T is (<>);
type T is range <>;
type T is mod <>;
type T is digits <>;
type T is delta <>;
type T is delta <> digits <>;
```

In the first case the actual parameter must be a discrete type – an enumeration type or integer type. In the other cases the actual parameter must be a signed integer type, modular type, floating point type, ordinary fixed point type or decimal type respectively. Within the generic unit the appropriate predefined operations and attributes are available.

As a simple example consider

```
generic
   type T is (<>);
function Next(X: T) return T;
```

```
function Next(X: T) return T is
begin
   if X = T'Last then
      return T'First;
   else
      return T'Succ(X);
   end if;
end Next;
```

The formal parameter T requires that the actual parameter must be a discrete type. Since all discrete types have attributes First, Last and Succ we can use these attributes in the body in the knowledge that the actual parameter will supply them. However, if a generic body uses First, Last or Range on a formal type and the actual type has a subtype predicate then Program_Error is raised. On the other hand the attributes First_Valid and Last_Valid can always be used; if the actual subtype does not have a subtype predicate then these attributes are equivalent to First and Last.

We could now write

```
function Tomorrow is new Next(Day);
```

so that Tomorrow(Sun) = Mon.

An actual generic parameter can also be a subtype but an explicit constraint or null exclusion is not allowed. The formal generic parameter then denotes the subtype. Thus we can have

```
function Next_Work_Day is new Next(Weekday);
```

so that Next_Work_Day(Fri) = Mon. Note how the behaviour depends on the fact that the Last attribute applies to the subtype and not to the base type so that Day'Last is Sun and Weekday'Last is Fri.

The actual parameter could also be an integer type so we could have

```
subtype Digit is Integer range 0 .. 9;
function Next_Digit is new Next(Digit);
```

and then Next_Digit(9) = 0.

Now consider the package Complex_Numbers of Section 12.2; this could be made generic so that the particular floating point type upon which the type Complex is based can be a parameter. It would then take the form

```
generic
   type Floating is digits <>;
package Generic_Complex_Numbers is
   type Complex is private;
   ...   -- as before with Float replaced by Floating
   I: constant Complex := (0.0, 1.0);
end;
```

Note that we can use the literals 0.0 and 1.0 because they are of the type *universal_real* which can be converted to whatever type is passed as actual parameter. The package could then be instantiated by for instance

```
package My_Complex_Numbers is
      new Generic_Complex_Numbers(My_Float);
```

A formal generic parameter can also be an array type. The actual parameter must then also be an array type with the same number of dimensions and statically matching index subtypes and component subtypes. The subcomponents could be marked as aliased but this must apply to either both the formal and actual type or neither of them. Similarly, either both must be unconstrained arrays or both must be constrained arrays. If constrained then the index ranges must statically match. Note that unlike ordinary array type declarations, generic formal arrays cannot have explicit constraints or null exclusions in the index subtypes and component subtypes in all cases.

As noted above it is possible for one generic formal parameter to depend upon a previous formal parameter which is a type. This will often be the case with arrays. As an example consider the function Sum in Section 10.1. This added together the components of an array of component type Float and index type Integer. We can generalize this to add together the components of any floating point array with any index type

```
generic
   type Index is (<>);
   type Floating is digits <>;
   type Vec is array (Index range <>) of Floating;
function Sum(A: Vec) return Floating;

function Sum(A: Vec) return Floating is
   Result: Floating := 0.0;
begin
   for I in A'Range loop
      Result := Result + A(I);
   end loop;
   return Result;
end Sum;
```

Note that although Index is a formal parameter it does not explicitly appear in the generic body; nevertheless it is implicitly used since the loop parameter I is of type Index.

We could instantiate this by

```
function Sum_Vector is new Sum(Integer, Float, Vector);
```

and this will give the function Sum of Section 10.1.

The matching of actual and formal arrays takes place after any formal types have been replaced in the formal array by the corresponding actual types. As an example of matching index subtypes note that if we had

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

then we would have to use Positive (or an equivalent subtype) as the actual parameter for the Index.

The final possibility for formal type parameters is the case of an access type. The formal can be

```
type A is access T;
type A is access constant T;
type A is access all T;
```

where T may but need not be a previous formal parameter. The actual parameter corresponding to A must then be an access type with accessed type T. In the first case the actual type can be any such access type other than an access to constant type. In the second case it must be an access to constant type and in the third it must be a general access to variable type. Constraints on the accessed type must statically match. The formal type can also have a null exclusion thus

```
type A is not null access T;
```

in which case the actual type must also have a null exclusion and vice versa.

A formal access to subprogram type takes one of the forms

```
type P is access procedure ...
type F is access function ...
```

and the profiles of actual and formal subprograms must have mode conformance. This is the same conformance as applies to the renaming of subprogram specifications and so follows the model that generic parameter matching is like renaming. Null exclusions are again allowed.

Observe that there is no concept of a formal record type; a similar effect can be achieved by the use of a formal derived type.

Incidentally, we have noted that a numeric type such as

```
type My_Integer is range –1E6 .. 1E6;
```

is implemented as a hardware type that might correspond to one of the predefined types such as Integer or Long_Integer. However, it is not actually derived from one of these although ultimately all are conceptually derived from *root_integer*. So if we had

```
generic
   type T is new Integer;
```

then this could never be matched by My_Integer. Another thought is that **type** T **is range** <>; can be considered as equivalent to **type** T **is new** *root_integer*; which we cannot write.

Another typical example is provided by considering operations on sets. We saw in Section 8.6 how a Boolean array could be used to represent a set. Exercises 10.1(**4**), 10.2(**3**) and 10.2(**4**) also showed how we could write suitable functions to operate upon sets of the type Colour. The generic mechanism allows us to write a package for manipulating sets of an arbitrary type.

Consider

```
generic
   type Element is (<>);
package Set_Of is
   type Set is private;
   type List is array (Positive range <>) of Element;

   Empty, Full: constant Set;

   function Make_Set(L: List) return Set;
   function Make_Set(E: Element) return Set;
   function Decompose(S: Set) return List;

   function "+" (S, T: Set) return Set;        -- union
   function "*" (S, T: Set) return Set;        -- intersection
   function "-" (S, T: Set) return Set;        -- symmetric difference

   function "<" (E: Element; S: Set) return Boolean;    -- inclusion
   function "<=" (S, T: Set) return Boolean;            -- contains

   function Size(S: Set) return Natural;       -- no of elements
private
   type Set is array (Element) of Boolean;

   Empty: constant Set := (Set'Range => False);
   Full: constant Set := (Set'Range => True);
end;
```

The single generic parameter is the element type which must be discrete. The type Set is private so that the Boolean operations cannot be applied directly (inadvertently or malevolently). Aggregates of the type List are used to represent literal sets. The constants Empty and Full denote the empty and full set respectively. The functions Make_Set enable the creation of a set from a list of the element values or a single element value. Decompose turns a set back into a list of elements.

The operators +, * and - represent union, intersection and symmetric difference; they are chosen as more natural than the underlying **or**, **and** and **xor** which can be used to implement them. The operator < tests to see whether an element value is in a set. The operator <= tests to see whether one set is a subset of another. Finally, the function Size returns the number of element values present in a particular set.

In the private part the type Set is declared as a Boolean array indexed by the element type (which is why the element type had to be discrete). The constants Empty and Full are declared as arrays whose elements are all False and all True respectively. The body of the package is left as an exercise.

We can instantiate the package to work on the type Primary of Section 8.6 by

```
package Primary_Sets is new Set_Of(Primary);
use Primary_Sets;
```

For comparison we could then write

```
subtype Colour is Set;
White: Colour renames Empty;
Black: Colour renames Full;
```

and so on.

We can use this example to illustrate the difference between the rules for the template (the generic text as written) and an instance (the effective text after instantiation).

The first point is that the generic package is not a genuine package and in particular does not export anything. So no meaning can be attached to Set_Of.List outside the generic package and nor can Set_Of appear in a use clause. Of course, inside the generic package we could indeed write Set_Of.List if we wished to be pedantic or had hidden List by an inner redeclaration.

If we now instantiate the generic package thus

```
package Character_Set is new Set_Of(Character);
```

then Character_Set is a genuine package and so we can refer to Character_Set.List outside the package and Character_Set can appear in a use clause. In this case there is no question of writing Character_Set.List *inside* the package because the inside text is quite ethereal.

Another very important point concerns the properties of an identifier such as List. Inside the generic template we can only use the properties common to all possible actual parameters as expressed by the formal parameter notation. Outside we can additionally use the properties of the particular instantiation. So, inside we cannot write

```
S: List := "String";
```

because we do not know that the actual type is going to be a character type – it could be an integer type. However, outside we can indeed write

```
S: Character_Set.List := "String";
```

because we know full well that the actual type is, in this instance, a character type.

We can also use this example to explore the creation and composition of types. Our attempt to give the type Set the name Colour through a subtype is not ideal because the old name could still be used. We would really like to pass the name Colour in some way to the generic package as the name to be used for the type. We cannot do this and retain the private nature of the type. But we can use the derived type mechanism to create a proper type Colour from the type Set

```
type Colour is new Set;
```

Recalling the rules for inheriting primitive subprograms from Section 12.3, we note that the new type Colour automatically inherits all the functions in the specification of Set_Of (strictly the instantiation Primary_Sets) because they all have the type Set as a parameter or result type.

However, this is a bit untidy; the constants Empty and Full will not have been inherited and the type List will still be as before.

One improvement therefore is to replace the constants Empty and Full by equivalent parameterless functions so that they will also be inherited. A better approach to the type List is to make it and its index type into further generic parameters.

The visible part of the package will then just consist of the type Set and its subprograms

```
generic
   type Element is (<>);
   type Index is (<>);
   type List is array (Index range <>) of Element;
package Nice_Set_Of is
   type Set is private;
   function Empty return Set;
   function Full return Set;
   ...
private
```

We can now write

```
type Primary_List is array (Positive range <>) of Primary;

package Primary_Sets is new Nice_Set_Of(Element => Primary,
                                        Index => Positive,
                                        List => Primary_List);
type Colour is new Primary_Sets.Set;
```

The type Colour now has all the functions we want and the array type has a name of our choosing. We might still want to rename Empty and Full thus

```
function White return Colour renames Empty;
```

or we can still declare White as a constant by

```
White: constant Colour := Empty;
```

As a general rule it is better to use derived types rather than subtypes because of the greater type checking provided during compilation; sometimes, however, derived types introduce a need for lots of explicit type conversions which clutter the program, in which case the formal distinction is probably a mistake and one might as well use subtypes.

As a final example in this section, consider the following generic form of a package to manipulate stacks (see Section 12.5)

```
generic
   type Item (<>) is private;
package Stacks is
   type Stack is limited private;
   procedure Push(S: in out Stack; X: in Item);
   procedure Pop(S: in out Stack; X: out Item);
   function "=" (S,T: Stack) return Boolean;
private ...
```

The key point to note is that we have chosen to use the form of generic parameter with unknown discriminants. The actual type can then be indefinite such as a class wide type or the unconstrained array type String. This enables us to manipulate a

stack of strings of different sizes such as the names of the animals. However, since the actual type can be indefinite this means that the generic body cannot declare objects of the type Item (without an initial value) or arrays of the type Item. The implementation therefore has to use allocated objects to hold the values of the type Item. The access values could themselves be held in an array in the traditional manner as in Section 12.4

```
type Access_Item_Array is array (Integer range <>) of access Item;

type Stack is
   record
      S: Access_Item_Array(1 .. Max);
      Top: Integer range 0 .. Max := 0;
   end record;
```

where Max could well be another generic parameter. Alternatively the individual access values could be linked together as in Section 12.5. Either way the type has to be limited private and care is needed in deallocating unneeded objects. See also the discussion in Section 21.4.

Note that the attribute Definite can be applied to an indefinite formal type such as Item and gives a Boolean value indicating whether the actual type is definite or not. We could therefore use alternative algorithms according to the nature of the actual parameter.

We conclude by summarizing the general principle regarding the matching of actual to formal generic types which should now be clear. The formal type represents a category of types which have certain common properties and these properties can be assumed in the generic unit. The corresponding actual type must then supply these properties. The matching rules are designed so that this is assured by reference to the parameters only and without considering the details of the generic body. As a consequence the user of the generic unit need not see the body for debugging purposes. This notion of matching guaranteed by the parameters is termed the contract model.

Some properties cannot be passed via the parameters and have to be checked upon instantiation. The general principle is to assume the best in the specification and then check it at instantiation but to assume the worst in the body so that it cannot go wrong. The situations where this happens are mostly obscure and in some cases the offending stuff can be moved to the private part. For example if the formal type is limited then the actual need not be. The body will assume the worst (that it is limited), but in the private part it will be viewed as limited or not according to the actual parameter. Similarly, remember that the use of generic formal parameters is restricted by the rule that they are not static in the body although they will be static in the specification if the actual parameter is static.

Another point is that accessibility checks in a generic body are always dynamic; this is because a unit might be instantiated at any level. Apart from these and a few other minor cases which need not concern the normal user, the general principle that any unit can be made generic holds true.

Exercise 19.2

1 Instantiate Next to give a function behaving like **not**.

2 Rewrite the specification of the package Rational_Numbers so that it is a generic package taking the integer type as a parameter. See Exercise 12.2(**3**).

3 Rewrite the function Outer of Exercise 10.1(**3**) so that it is a generic function with appropriate parameters. Instantiate it to give the original function.

4 Write the body of the package Set_Of.

5 Rewrite the private part of Set_Of so that an object of the type Set is by default given the initial value Empty when declared.

19.3 Subprogram parameters

As mentioned earlier a generic parameter can also be a subprogram. There are a number of characteristic applications of this facility and we introduce the topic by considering the classical problem of sorting.

Suppose we wish to sort an array into ascending order. There are a number of general algorithms that can be used which do not depend on the type of the values being sorted. All we need is some comparison operation such as "<" which is defined for the type.

We might start by considering the specification

```
generic
   type Index is (<>);
   type Item is (<>);
   type Collection is array (Index range <>) of Item;
procedure Sort(C: in out Collection);
```

Although the body is largely irrelevant it might help to illustrate the problem to consider the following crude possibility

```
procedure Sort(C: in out Collection) is
   Min: Index;
   Temp: Item;
begin
   for I in C'First .. Index'Pred(C'Last) loop
      Min := I;
      for J in Index'Succ(I) .. C'Last loop
         if C(J) < C(Min) then Min := J; end if;       -- use of <
      end loop;
      Temp := C(I);  C(I) := C(Min);  C(Min) := Temp;
   end loop;
end Sort;
```

This trivial algorithm repeatedly scans the part of the array not sorted, finds the least component (which because of the previous scans will be not less than any component of the already sorted part) and then swaps it so that it is then the last

element of the now sorted part. Note that because of the generality we have imposed upon ourselves, we cannot write

```
for I in C'First .. C'Last–1 loop
```

because we cannot rely upon the array index being an integer type. We only know that it is a discrete type and therefore have to use the attributes Index'Pred and Index'Succ which we know to be available since they are common to all discrete types.

However, the main point to note is the call of "<" in the body of Sort. This calls the predefined function corresponding to the type Item. We know that there is such a function because we have specified Item to be discrete and all discrete types have such a function. Unfortunately the net result is that our generic sort can only sort arrays of discrete types. It cannot sort arrays of floating types. Of course we could write a version for floating types by replacing the generic parameter for Item by

```
type Item is digits <>;
```

but then it would not work for discrete types. What we really need to do is specify the comparison function to be used in a general manner. We can do this by adding a fourth parameter which is a formal subprogram so that the specification becomes

```
generic
   type Index is (<>);
   type Item is private;
   type Collection is array (Index range <>) of Item;
   with function "<" (X, Y: Item) return Boolean;
procedure Sort(C: in out Collection);
```

The formal subprogram parameter is like a subprogram declaration preceded by **with**. (The leading **with** is necessary to avoid a syntactic ambiguity and has no other subtle purpose.)

We have also made the type Item private since the only common property now required (other than supplied through the parameters) is that the type Item can be assigned. The body remains as before.

We can now sort an array of any (nonlimited) type provided that we have an appropriate comparison to supply as parameter. So in order to sort an array of the type Vector, we first instantiate thus

```
procedure Sort_Vector is
      new Sort(Integer, Float, Vector, "<");
```

and we can then apply the procedure to the array concerned

```
An_Array: Vector( ... );
...
Sort_Vector(An_Array);
```

Note that the call of "<" inside Sort is actually a call of the function passed as actual parameter; in this case it is indeed the predefined function "<" anyway.

Passing the comparison rule gives our generic sort procedure amazing flexibility. We can, for example, sort in the reverse direction by

```
procedure Reverse_Sort_Vector is
      new Sort(Integer, Float, Vector, ">");
...
Reverse_Sort_Vector(An_Array);
```

This may come as a slight surprise but it is a natural consequence of the call of the formal "<" in

```
if C(J) < C(Min) then ...
```

being, after instantiation, a call of the actual ">". No confusion should arise because the internal call is hidden but the use of the named notation for instantiation would look curious

```
procedure Reverse_Sort_Vector is
      new Sort( ..., "<" => ">");
```

We could also sort our second Farmyard of Section 8.5 assuming it to be a variable so that the animals are in alphabetical order

```
subtype String_3 is String(1 .. 3);

procedure Sort_String_3_Array is
      new Sort(Positive, String_3, String_3_Array, "<");
...
Sort_String_3_Array(Farmyard);
```

The "<" operator passed as parameter is the predefined operation applicable to one-dimensional arrays described in Section 8.6.

The correspondence between formal and actual subprograms is such that the formal subprogram just renames the actual subprogram. Thus the matching rules regarding parameters, results and so on are as described in Section 13.7. In particular the constraints on the parameters are those of the actual subprogram and any implied by the formal subprogram are ignored. A parameterless formal function can also be matched by an enumeration literal of the result type just as for renaming.

Generic subprogram parameters (like generic object parameters) can have default values. These are given in the generic formal part and take three forms. In the above example we could write

```
with function "<" (X, Y: Item) return Boolean is <>;
```

This means that we can omit the corresponding actual parameter if there is visible at the point of *instantiation* a unique subprogram with the same designator and matching specification. With this alteration to Sort we could have omitted the last parameter in the instantiation giving Sort_Vector.

Another form of default value is where we give an explicit name for the default parameter. The usual rules for defaults apply; the default name is only evaluated if

required by the instantiation but the binding of identifiers in the expression which is the name occurs at the point of *declaration* of the generic unit. In our example

```
with function "<" (X, Y: Item) return Boolean is Less_Than;
```

could never be valid because the specification of Less_Than must match that of "<" and yet the parameter Item is not known until instantiation. Valid possibilities are where the formal subprogram has no parameters depending on formal types or the default subprogram is an operation or attribute of a formal type or is itself another formal parameter. Thus we might have

```
with function Next(X: T) return T is T'Succ;
```

The final form of default parameter can only be used with formal procedures and not functions and is simply a null procedure. Thus we might have

```
with procedure P( ... ) is null;
```

As a final example of the use of our generic Sort (which we will assume now has a default parameter <> for "<"), we show how any type can be sorted provided we supply an appropriate rule.

Thus consider sorting an array of the type Date from Section 8.7. We write

```
type Date_Array is array (Positive range <>) of Date;

function "<" (X, Y: Date) return Boolean is
begin
   if X.Year /= Y.Year then
      return X.Year < Y.Year;
   elsif X.Month /= Y.Month then
      return X.Month < Y.Month;
   else
      return X.Day < Y.Day;
   end if;
end "<";

procedure Sort_Date_Array is new Sort(Positive, Date, Date_Array);
```

where the function "<" is passed through the default mechanism.

We could give the comparison rule a more appropriate name such as

```
function Earlier(X, Y: Date) return Boolean;
```

but we would then have to pass it as an explicit parameter.

Formal subprograms can be used to supply further properties of type parameters in a quite general way. Consider the generic function Sum of the previous section. We can generalize this even further by passing the adding operator itself as a generic parameter

```
generic
   type Index is (<>);
   type Item is private;
   type Vec is array (Index range <>) of Item;
```

```
   with function "+" (X, Y: Item) return Item;
function Apply(A: Vec) return Item;

function Apply(A: Vec) return Item is
   Result: Item := A(A'First);
begin
   for I in Index'Succ(A'First) .. A'Last loop
      Result := Result + A(I);
   end loop;
   return Result;
end Apply;
```

The operator "+" has been added as a parameter and Item is now just private and no longer floating. This means that we can apply the generic function to any binary operation on any type. However, we no longer have a zero value and so have to initialize Result with the first component of the array A and then iterate through the remainder. In doing this, remember that we cannot write

```
for I in A'First+1 .. A'Last loop
```

because the type Index may not be an integer type.

Our original function Sum of Section 10.1 is now given by

```
function Sum is new Apply(Integer, Float, Vector, "+");
```

We could equally have

```
function Prod is new Apply(Integer, Float, Vector, "*");
```

Another possible use of formal subprograms is in mathematical applications such as integration. As we saw in Section 11.8, values of access to subprogram types can be passed as parameters to other subprograms and this is usually perfectly satisfactory. However, we might need to make the unit generic so that any floating type can be used. We could then pass the function to be integrated as a further generic parameter.

We could have a generic function

```
generic
   type Floating is digits <>;
   with function F(X: Floating) return Floating;
function Integrate(A, B: Floating) return Floating;
```

and then in order to integrate a particular function we must instantiate Integrate with that function as actual generic parameter. Thus suppose we needed to evaluate

$$\int_0^P e^t \sin t \, dt$$

using the type Long_Real. We would write

```
function G(T: Long_Real) return Long_Real is
begin
   return Exp(T) * Sin(T);
end;

function Integrate_G is new Integrate(Long_Real, G);
```

and then the result is given by the expression

```
Integrate_G(0.0, P)
```

In practice the function Integrate would have other parameters indicating the accuracy required and so on.

Examples such as this are often found confusing at first sight. The key point to remember is that there are two distinct levels of parameterization. First we fix the function to be integrated at instantiation and then we fix the bounds when we call the integration function thus declared. The sorting examples were similar; first we fixed the parameters defining the type of array to be sorted and the rule to be used at instantiation, and then we fixed the actual array to be sorted when we called the procedure.

Whether to use a generic or an access to subprogram parameter is to some extent a matter of taste. The generic mechanism enables the type to be parameterized as well and also permits the actual subprogram to have convention Intrinsic whereas this cannot be done with an access to subprogram parameter since the Access attribute cannot be applied to intrinsic operations. As we shall see in Chapter 24, the container library uses access to subprogram parameters for iteration but generics for sorting.

There is one other form of generic subprogram parameter which we will briefly outline here and explain its use in Section 21.7. This takes the form

```
with procedure Do_This( ... ) is abstract;
```

The formal subprogram Do_This must have controlling parameters or result of just one tagged type which may be a formal type or some other type global to the generic. The following would be illegal

```
with procedure Do_That(X1: TT1; X2: TT2) is abstract;      -- illegal
with function Fn(X: Float) return Float is abstract;       -- illegal
```

The first is illegal because it has two controlling types TT1 and TT2 and the second is illegal because it does not have any.

The actual parameter can be abstract or concrete. Remember that the overriding rules ensure that the specific operation for any concrete type will always have a concrete operation. Note also that since the operation is abstract it can only be called through dispatching.

Formal abstract subprograms can have defaults in much the same way that formal concrete subprograms can have defaults. We write

```
with procedure P(X: in out T) is abstract <>;
with function F return T is abstract Unit;
```

The first means of course that the default has to have identifier P and the second means that the default is some function Unit. It is not possible to give null as the default for an abstract parameter for various reasons. Defaults will probably be rarely used for abstract parameters.

We conclude with an important remark concerning the identification of primitive operations. A primitive operation such as "<" which applies to all discrete types (as in the first example of sorting) and occurring inside the generic unit always refers to the predefined operation even if it has been redefined for the actual type concerned. The general principle is that since the generic unit applies to the category of discrete types as a whole then the primitive operations ought to refer to those guaranteed for all members of the category. Of course if the operation is passed as a distinct parameter (explicitly or by default) then the redefined operation will apply.

On the other hand, if the formal parameter is tagged, so that the actual type is an extension of the formal, then primitive operations refer to overriding ones. The reason for the different approach is that the whole essence of tagged types is to override operations and so within the generic it is natural to refer to the operation of the actual type. For example if we wrote

```
generic
   type T is new Object with private;
package P ...
```

so that the actual type must be derived from Object then inside P we expect the function Area to refer to that of the actual type supplied such as Circle and not to that of Object. Indeed in the formulation of Exercise 14.3(**1**) it would be a disaster if it referred to the Area of Object since it is abstract.

Exercise 19.3

1 Instantiate Sort to apply to

```
type Poly_Array is array (Integer range <>) of Polynomial;
```

See Section 18.2. Define a sensible ordering for polynomials.

2 Instantiate Sort to apply to an array of the type Mutant of Section 18.3. Put neuter things first, then females, then males and within each class the younger first. Could we sort an array of the type Person from the same section?

3 Sort the array People of Section 8.7.

4 What happens if we attempt to sort an array of less than two components?

5 Describe how to make a generic sort procedure based on the procedure Sort of Section 11.2. It should have an identical specification to the procedure Sort of this section.

6 Write a generic function to search an array and return the index of the first component satisfying some criterion. Make one parameter the identity of an exception to be raised if there is no such component. Note that although an exception cannot be passed as a parameter, nevertheless a value of the type

Exception_Id can be passed; see Section 15.4. Make the exception Constraint_Error by default.

7 Write a generic function Equals to define the equality of one-dimensional arrays of a private type. See Exercise 12.4(**4**). Instantiate it to give the function "=" applying to the type Stack_Array.

8 Reconsider Exercise 11.8(**2**) using generics.

19.4 Package parameters

The last kind of formal generic parameter is the formal package. This greatly simplifies the composition of generic packages by allowing one package to be used as a parameter to another so that a hierarchy of consistently related packages can be created.

There are two basic forms of formal package parameters. The simplest is

```
with package P is new Q(<>);
```

which indicates that the actual parameter corresponding to P must be a package which has been obtained by instantiating Q which must itself be a generic package. We can also explicitly indicate the actual parameters required by the instantiation of Q thus

```
with package R is new Q(P1, P2, P3);
```

and then the actual package corresponding to R must be an instantiation of Q with matching parameters – **in** parameters must be static with the same value, subtypes must statically match and any others must denote the same entity.

It is also possible to specify just some of the parameters by using the <> notation as in aggregates. The named notation must be used for such parameters. We could write either of

```
with package S is new Q(P1, F2 => <>, F3 => <>);
with package S is new Q(P1, others => <>);
```

and in this case the actual package corresponding to S can be any package which is an instantiation of Q where the first actual parameter is P1 but the other two parameters are left unspecified. Incidentally the form Q(<>); can be seen as an abbreviation for Q(**others** => <>);

As a simple example suppose we wish to develop a package for the manipulation of complex vectors; we want to make it generic with respect to the underlying floating type. Naturally enough we will build on our simple package for complex numbers in its generic form outlined in Section 19.2

```
generic
   type Floating is digits <>;
package Generic_Complex_Numbers is
   type Complex is private;
   ...
```

```
   function "+" (X, Y: Complex) return Complex;
   ...
end;
```

Our new generic package will need to use the various arithmetic operations exported from some instantiation of Generic_Complex_Numbers. These could all be imported as individual subprogram parameters but this would give a very long formal parameter list. Instead we can import the instantiated package as a whole. All we have to write is

```
generic
   type Index is (<>);
   with package Complex_Numbers is
         new Generic_Complex_Numbers (<>);
package Generic_Complex_Vectors is
   use Complex_Numbers;
   type Vector is array (Index range <>) of Complex;
   ...   -- other types and operations on vectors
end;
```

and then we can instantiate the two packages by a sequence such as

```
package Long_Complex is
      new Generic_Complex_Numbers(Long_Float);
package Long_Complex_Vectors is
      new Generic_Complex_Vectors(Integer, Long_Complex);
```

Note the use clause in the specification of Generic_Complex_Vectors. Without this the component subtype of the type Vector would have to be written as Complex_Numbers.Complex. Note also that within Generic_Complex_Vectors not only do we have visibility of all the operations exported by the instantiation of Generic_Complex_Numbers but we can also refer to the formal parameter Floating. This is a special rule for formal parameters with the default form <>; we shall see its use in a moment.

We now consider a more elaborate example where the new package builds on the properties of two other packages. One is the toy package Generic_Complex_Numbers and the other is the predefined library package for computing elementary functions. This is described in detail in Section 23.4 but for this chapter all we need is the following outline sketch

```
generic
   type Float_Type is digits <>;
package Ada.Numerics.Generic_Elementary_Functions is
   ...
   function Sqrt(X: Float_Type'Base) return Float_Type'Base;
   ...   -- similarly other functions such as
   ...   -- Log, Exp, Sin, Cos, Sinh, Cosh
   ...
end;
```

(The parameters and results have subtype Float_Type'Base in order to avoid unnecessary constraint checks in intermediate expressions.)

Although the following example is of a rather mathematical nature it is hoped that the general principles will be appreciated. It follows on from the above elementary functions package and concerns the provision of similar functions but working on complex arguments. Suppose we want to provide the ability to compute Sqrt, Log, Exp, Sin and Cos with functions such as

```
function Sqrt(X: Complex) return Complex;
```

Many readers will have forgotten that this can be done or perhaps never knew. It is not necessary to dwell on the details of how such calculations are performed or their use; the main point is to concentrate on the principles involved. These computations use various operations on the real numbers out of which the complex numbers are formed. Our goal is to write a generic package which works however the complex numbers are implemented (cartesian or polar) and also allows any floating point type as the basis for the underlying real numbers.

Here are the formulae which we will need to compute

Taking $z \equiv x + iy \equiv r(\cos\ \theta + i \sin\ \theta)$ as the argument:

$\text{sqrt}\ z = r^{1/2}\ (\cos\ \theta/2 + i \sin\ \theta/2)$
$\log z = \log r + i\ \theta$
$\exp z = e^x(\cos y + i \sin y)$
$\sin z = \sin x \cosh y + i \cos x \sinh y$
$\cos z = \cos x \cosh y - i \sin x \sinh y$

We thus see that we will need the functions Sqrt, Cos, Sin, Log, Exp, Cosh and Sinh applying to the underlying floating type. We also need to be able to decompose and reconstruct a complex number using both cartesian and polar forms; we will assume that the package Generic_Complex_Numbers has been extended to do this (we will consider the question of child packages in the next section).

We can now write

```
with Ada.Numerics.Generic_Elementary_Functions;
use Ada.Numerics;
with Generic_Complex_Numbers;
generic
   with package Elementary_Functions is
         new Generic_Elementary_Functions(<>);
   with package Complex_Numbers is
         new Generic_Complex_Numbers
                                    (Elementary_Functions.Float_Type);
package Generic_Complex_Functions is
   use Complex_Numbers;

   function Sqrt(X: Complex) return Complex;
   ...

end Generic_Complex_Functions;
```

where the actual packages must be instantiations of Generic_Elementary_Functions and Generic_Complex_Numbers. Note the forms of the formal packages. Any instantiation of Generic_Elementary_Functions is allowed but the instantiation of Generic_Complex_Numbers must have Elementary_Functions.Float_Type as its actual parameter. This ensures that both packages are instantiated with the same floating type.

Note carefully that we are using the formal parameter exported from the first instantiation as the required parameter for the second instantiation. As noted above, a formal parameter is only accessible in this way when the default form <> is used. In order to reduce verbosity it is permitted to have a use clause in the generic formal list, so we could have written

```
with package Elementary_Functions is
      new Generic_Elementary_Functions(<>);
use Elementary_Functions;
with package Complex_Numbers is
      new Generic_Complex_Numbers(Float_Type);
```

although this is perhaps not so clear.

Finally the instantiations are

```
type My_Float is digits 9;

package My_Elementary_Functions is
      new Generic_Elementary_Functions(My_Float);

package My_Complex_Numbers is
      new Generic_Complex_Numbers(My_Float);

package My_Complex_Functions is
      new Generic_Complex_Functions
                  (My_Elementary_Functions, My_Complex_Numbers);
```

and *Hey presto!* it all works. Note that, irritatingly, the complex type is just Complex and not My_Complex. However we could use a subtype or derived type as we did for the type Colour and the package Set_Of in Section 19.2.

The reader might wonder why we did not simply instantiate the various packages inside the body of Generic_Complex_Functions and thereby avoid the package parameters. This works but could result in wasteful and unnecessary multiple instantiations since we may well need them at the user level anyway.

It is hoped that the general principles have been understood and that the mathematics has not clouded the issues. The principles are important but not easily illustrated with short examples. It should also be noted that although the package Generic_Elementary_Functions described above is exactly as in the predefined library, the complex number packages are just an illustration of how generics can be used. The Numerics annex does indeed provide packages for complex numbers and complex functions but they use a rather different approach as outlined in Section 26.5. A better example (also in the Numerics annex) is the package Generic_Complex_Arrays which takes instantiations of the two packages Generic_Real_Arrays and Generic_Complex_Types.

Another application of formal packages is where we wish to bundle together a number of related types and operations and treat them as a whole. We make them parameters of an otherwise null generic package. In essence, the successful instantiation of the generic package is an assertion that the entities have the required relationship and the group can then be referred to using the instantiated package (sometimes known as a signature).

As a very trivial example we will have noticed that the group of parameters

```
type Index is (<>);
type Item is private;
type Vec is array (Index range <>) of Item;
```

has occurred from time to time; see Sort and Apply in the previous section. For a successful instantiation of a generic unit having these as parameters it is necessary that the actual types have the required relationship. We could write

```
generic
   type Index is (<>);
   type Item is private;
   type Vec is array (Index range <>) of Item;
package General_Vector is end;
```

(note the curious juxtaposition of **is end** – we can even write **is private end**) and then rewrite Sort and Apply as

```
generic
   with package P is new General_Vector(<>);
   with function "<" (X, Y: P.Item) return Boolean is <>;
procedure Sort (V: in out P.Vec);
```

```
generic
   with package P is new General_Vector(<>);
   use P;
   with function "+" (X, Y: Item) return Item;
function Apply(A: Vec) return Item;
```

Note that the use clause in the formal list for Apply simplifies the text with some risk of obscurity. We can now perform instantiations as follows. First we might write

```
package Float_Vector is new General_Vector(Integer, Float, Vector);
```

which ensures that Integer, Float and Vector have the required relationship. Then we can write

```
procedure Sort_Vector is new Sort(Float_Vector, "<");

function Sum is new Apply(Float_Vector, "+");
```

which should be compared with the corresponding instantiations in the previous section. Clearly this example is rather trivial and the gain obtained by simplifying the parameter lists for Sort and Apply is barely worth the effort of the extra layer of abstraction.

Another example might be the signature of a package for manipulating sets. We can write (note that the formal types are incomplete)

```
generic
   type Element;              -- incomplete
   type Set;                  -- incomplete
   with function Empty return Set is <>;
   with function Unit(E: Element) return Set is <>;
   with function Union(S, T: Set) return Set is <>;
   with function Intersection(S, T: Set) return Set is <>;
   ...
package Set_Signature is end;
```

We might then have some other generic package which takes an instantiation of this set signature. However, it is likely that we would need to specify the type of the elements but possibly not the set type and certainly not all the operations. So typically we would have

```
generic
   type My_Element is private;
   with package Sets is
                new Set_Signature(Element => My_Element, others => <>);
```

and this illustrates a situation where one parameter is given but the others are arbitrary.

For a further discussion on this topic see Section 4.3 of the *Rationale*.

Exercise 19.4

1 Write a body for the package Generic_Complex_Functions using the formulae defined above. Ignore exceptions.

2 Reconsider Exercise 19.3(**1**) using the package General_Vector.

3 A mathematical group is defined by an operation over a set of elements thus

```
generic
   type Element is private;
   Identity: in Element;
   with function Op(X, Y: Element) return Element;
   with function Inverse(X: Element) return Element;
package Group is end;
```

Declare a generic function Power which takes an element E and a signed integer N as parameters and delivers the *N*th power of the element using the group operation. Then define the addition group over the type Integer and finally instantiate the power function.

4 Any finite group can be represented as a discrete type. Adapt the definition of Group of the previous exercise accordingly and then define a (generic) function Is_Group that checks whether a signature conforms to the semantic requirements of a group.

19.5 Generic library units

We now consider the interaction between generics and hierarchical libraries. Both are important tools in the construction of subsystems and it is essential that genericity be usable with the child concept.

It is an important rule that if a parent unit is generic then all its children must also be generic. The reverse does not hold; a nongeneric parent can have both generic children and nongeneric children.

If the parent unit is not generic then a generic child may be instantiated in the usual way at any point where it is visible. On the other hand, if the parent unit is itself generic, then the rules regarding the instantiation of a child are somewhat different according to whether the instantiation is inside or outside the generic hierarchy. If inside then the instantiation is as normal (see Program 4 for an example) but if outside then the parent must be instantiated first.

In effect the instantiation of the parent creates an actual unit with a generic child inside it; we can then instantiate this child provided we have a with clause for the original generic child. Note that these instantiations do not have to be as library units although they might be.

This is best illustrated with a symbolic example consisting of a package Parent and a child Parent.Child. The parent will typically have formal generic parameters but often the child will have none since it will be parameterized by those of its parent which will be visible to it anyway. So we might have

```
generic
   type T is private;
package Parent is
   ...
end Parent;

generic
package Parent.Child is
   ...
end Parent.Child;
```

We can then instantiate this hierarchy inside some other package P by writing

```
with Parent.Child;
package P is
   package Parent_Instance is new Parent(T => Some_Type);
   package Child_Instance is new Parent_Instance.Child;
   ...
end P;
```

Note that the names of the new units are quite unrelated. Moreover, the instantiation for the child refers to Parent_Instance.Child and not Parent.Child. Nevertheless we need a with clause for Parent.Child (we do not need a with clause for the parent because that for the child implies one anyway).

If we instantiate at library level then again they might have unrelated names thus

```
with Parent;
package Parent_Instance is new Parent(T => Some_Type);

with Parent.Child;
with Parent_Instance;
package Child_Instance is new Parent_Instance.Child;
```

Note how in the second case we need a with clause for both the instance of the parent and the original child.

We could also form the instances into a hierarchy with a similar structure to the original

```
with Parent;
package Parent_Instance is new Parent(T => Some_Type);

with Parent.Child;
package Parent_Instance.Child_Instance is new Parent_Instance.Child;
```

In the second case we only have one with clause; remember that a child never needs one for its parent. Note moreover, that in this hierarchical case, the child names have to be different from those in the original generic hierarchy otherwise there would be a clash of names.

It is a general principle that we have to instantiate a generic hierarchy (or as much of it as we want) unit by unit. The main reason for requiring all children of a generic unit to be generic is to provide a handle to do this; as a consequence it is often the case that the child units have no formal generic parameters.

Of course, we need not instantiate the whole of a hierarchy for a particular application; indeed, one of the reasons for the unit by unit approach is to eliminate problems concerning the impact of the addition of new children to the generic hierarchy on existing instantiations.

Exercise 19.5

1 Sketch the structure of a hierarchy of three complex number packages Generic_Complex_Numbers, Generic_Complex_Numbers.Cartesian and Generic_Complex_Numbers.Polar; see Exercise 13.3(**1**). Then rewrite the specification of Generic_Complex_Functions of the previous section using this hierarchical form.

Checklist 19

The generic mechanism is not text replacement; nonlocal name binding would be different.

Object **in out** parameters are bound by renaming.

Subprogram generic parameters are bound by renaming.

Generic subprograms may not overload – only the instantiations can.

Generic subprograms always have a separate specification and body.

Formal parameters (and defaults) may depend upon preceding parameters.
Generic formal parameters and their attributes are not static.
Ada 95 did not permit parameters to be objects of limited types of mode **in**.
The parameter forms for objects of anonymous access types were added in Ada 2005.
The forms for interfaces were added in Ada 2005.
The default **null** for formal subprogram parameters was added in Ada 2005.
The form **is abstract** for formal subprogram parameters was added in Ada 2005.
The ability to specify just some parameters of a formal package was added in Ada 2005.

New in Ada 2012

Formal parameters for incomplete types are added in Ada 2012.