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 in 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. 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 became an ISO standard towards the end of 2012[5]. Maintenance then continued under the leadership of Jeff Cousins. A number of corrections were made and resulted in the Corrigendum 1 version of Ada 2012 published in 2016[6] and which is the subject of this book.

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 printed 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).

6 International Organization for Standardization. *Information technology – Programming languages – Ada. Technical Corrigendum 1.* ISO/IEC 8652:2012(E)/COR.1:2016.

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.