Object-Oriented Programming via Fortran 90/95

ED AKIN
Rice University

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2003

Printed in the United States of America

Typefaces Times Ten 9.5/12 pt. *with* Helvetica Neue, Optima, *and* Lucida Typewriter
System LaTeX 2$_\varepsilon$ [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication data available

ISBN 0 521 52408 3 paperback

Contents

CHAPTER ONE

Program Design

1.1 Introduction

The programming process is similar in approach and creativity to writing a paper. In composition, you are writing to express ideas; in programming, you are expressing a computation. Both the programmer and the writer must adhere to the syntactic rules (grammar) of a particular language. In prose, the fundamental idea-expressing unit is the sentence; in programming, two units – *statements* and *comments* – are available.

Composition, from technical prose to fiction, should be organized broadly, usually through an outline. The outline should be expanded as the detail is elaborated and the whole reexamined and reorganized when structural or creative flaws arise. Once the outline settles, you begin the actual composition process using sentences to weave the fabric your outline expresses. Clarity in writing occurs when your sentences, both internally and globally, communicate the outline succinctly and clearly. We stress this approach here with the aim of developing a programming style that produces efficient programs humans can easily understand.

To a great degree, no matter which language you choose for your composition, the idea can be expressed with the same degree of clarity. Some subtleties can be better expressed in one language than another, but the fundamental reason for choosing your language is your audience: people do not know many languages, and if you want to address the American population, you had better choose English over Swahili. Similar situations happen in programming languages, but they are not nearly so complex or diverse. The number of languages is far fewer, and their differences minor. Fortran is the oldest language among those in use today. The C and C++ languages differ from it somewhat, but there are more similarities than not (see Bar-David [6], Barton and Nackman [7], Hanly [22], Hubbard [24], and Nielsen [30]). MATLAB, written in C and Fortran, was created much later than these two, and its structure is so similar to the others that it can easily be mastered (see Hanselman and Littlefield [23], and Pratap [33]). The C++ language is an extension of the C language that places its emphasis on object-oriented programming (OOP) methods. Fortran added object-oriented capabilities with its F90 standard, and additional enhancements for parallel machines were issued with F95(see Adams et al. [1], Gehrke [17], Hahn [21], Kerrigan [25], and Press et al. [34]). The Fortran 200X standard is planned to contain more user-friendly constructs for polymorphism and will thus enhance its object-oriented capabilities. This creation of a new language and its similarity to more established ones are this book's main points: more computer programming languages will be created during your career, but these new languages will probably not be much different than ones you already know. Why should new languages

evolve? In the case of MATLAB, the desire to express matrix-like expressions easily motivated its creation. The difference between MATLAB and Fortran 90 is infinitesimally small compared with the gap between English and Swahili.

An important difference between programming and composition is that in programming you are writing for two audiences: people and computers. As for the computer audience, what you write is "read" by interpreters and compilers specific to the language you used. They are very rigid about syntactic rules, and perform exactly the calculations you say. It is like a document you write being read by the most detailed, picky person you know; every pronoun is questioned, and if the antecedent is not perfectly clear, then they throw up their hands, rigidly declaring that the entire document cannot be understood. Your picky friend might interpret the sentence "Pick you up at eight" to mean that you will literally lift him or her off the ground at precisely 8 o'clock and will then demand to know whether the time is in the morning or afternoon and what the date is.

Humans demand even more from programs. This audience consists of two main groups whose goals can conflict. The larger of the two groups consists of *users*. Users care about how the program presents itself, its *user interface*, and how quickly the program runs that is, how efficient it is. To satisfy this audience, programmers may use statements that are overly terse because they know how to make the program more readable by the computer's compiler, enabling the compiler to produce faster but less human-intelligible programs. This approach causes the other portion of the audience – programmers – to boo and hiss. The smaller audience, of which you are also a member, must be able to read the program to enhance or change it. A characteristic of programs that further distinguishes it from prose is that you and others will seek to modify your program in the future. For example, in the 1960s, when the first version of Fortran was created, useful programs by today's standards (such as matrix inversion) were written. Back then, the user interface possibilities were quite limited, and the use of visual displays was limited. Thirty years later, you would (conceivably) want to take an old program, and provide a modern user interface. If the program is structurally sound (a good outline and organized well) and is well written, reusing the "good" portions is easy accomplished.

The three-audience situation has prompted most languages to support both computer- and human-oriented "prose." The program's meaning is conveyed by statements and is what the computer interprets. Humans read this part, which in virtually all languages bears a strong relationship to mathematical equations, and also read *comments*. Comments are not read by the computer at all but are there to help explain what might be expressed in a complicated way by programming language syntax. The document or program you write today should be understandable tomorrow, not only by you, but also by others. Sentences and paragraphs should make sense after a day or so of gestation. Paragraphs and larger conceptual units should not contain assumptions or leaps that confuse the reader. Otherwise, the document you write for yourself or others serves no purpose. The same is true with programming; the program's organization should be easy to follow, and the way you write the program, using both statements and comments, should help you and others understand how the computation proceeds. The existence of comments permits the writer to express the program's outline directly in the program to help the reader comprehend the computation.

These similarities highlight the parallels between composition and programming. Differences become evident because programming is, in many ways, more demanding than prose writing. On one hand, the components and structure of programming languages are far simpler than the grammar and syntax of any verbal or written language. When reading a document, you can figure out the misspelled words and not be bothered about every little imprecision in interpreting what is written. On the other hand, simple errors, akin to

misspelled words or unclear antecedents, can completely undermine a program, rendering it senseless or causing it to go wildly wrong during execution. For example, there is no real dictionary when it comes to programming. You can define variable names containing virtually any combination of letters (upper- and lowercase), underscores, and numbers. A typographical error in a variable's name can therefore lead to unpredictable program behavior. Furthermore, computer execution speeds are becoming faster and faster, meaning that increasingly complex programs can run very quickly. For example, the program (actually groups of programs) that runs NASA's space shuttle might be comparable in size to Hugo's *Les Misérables*, but its complexity and immediate importance to the "user" far exceed that of the novel.

As a consequence, program design must be extremely structured and have the ultimate intentions of performing a specific calculation efficiently with attractive, understandable, efficient programs. Achieving these general goals means breaking the program into components, writing and testing them separately, and then merging them according to the outline. Toward this end, we stress *modular programming*. Modules can be on the scale of chapters or paragraphs and share many of the same features. They consist of a sequence of statements that by themselves express a meaningful computation. They can be merged to form larger programs by specifying what they do and how they interface to other packages of software. The analogy in prose is agreeing on the character's names and what events are to happen in each paragraph so that events happen to the right people in the right sequence once the whole is formed. Modules can be reused in two ways. As with our program from the 1960s, we would "lift" the matrix inversion routine and put a different user interface around it. We can also reuse a routine within a program several times. For example, solving the equations of space flight involves the inversion of many matrices. We would want our program to use the matrix inversion routine over and over, presenting it with a different matrix each time.

The fundamental components of good program design are

1. Problem definition, leading to a program specification;
2. Modular program design, which refines the specification;
3. Module composition, which translates specification into executable program;
4. Module and program evaluation and testing, during which you refine the program and find errors; and
5. Program documentation, which pervades all other phases.

The result of following these steps is an efficient, easy-to-use program that has a user's guide (to enable someone else run your program) and internal documentation so that other programmers can decipher the algorithm.

Today it is common in a university education to be required to learn at least one foreign language. Global interactions in business, engineering, and government make such a skill valuable to one's career. So it is in programming. One often needs to be able to read two or three programming languages – even if you compose programs in only one language. It is common for different program modules, in different languages, to be compiled separately and then brought together by a "linker" to form a single executable. When something goes wrong in such a process it is usually helpful to have a reading knowledge of the programming languages being used.

When one composes to express ideas there are, at least, two different approaches to consider: poetry and prose. Likewise, in employing programming languages to create software distinctly different approaches are available. The two most common ones are "procedural programming" and "object-oriented programming." The two approaches are conceptually

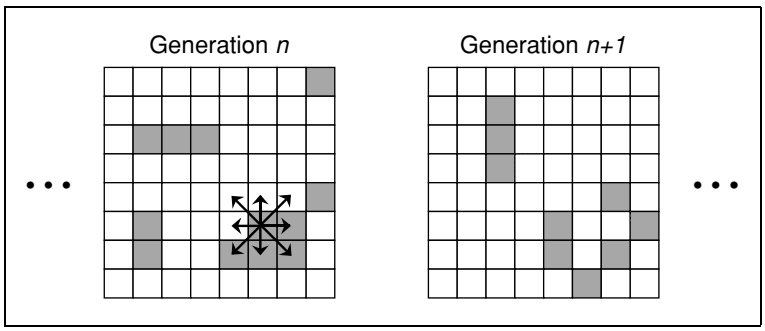

Figure 1.1: Here, the game is played on an 8×8 square array, and the filled squares indicate the presence of life. The arrows emanating from one cell radiate to its eight neighbors. The rules are applied to the nth generation to yield the next. The row of three filled cells becomes a column of three, for example. What is going to happen to this configuration in the next generation?

sketched in Figure 1.1. They differ in the way that the software development and maintenance are planned and implemented. Procedures may use objects, and objects usually use procedures called *methods*. Usually the object-oriented code takes more planning and is significantly larger, but it is generally accepted to be easier to maintain. Today when one can have literally millions of users active for years or decades, maintenance considerations are very important.

1.2 Problem Definition

The problem the program is to solve must be well specified. The programmer must broadly frame the program's intent and context by answering several questions.

- *What must the program accomplish?*
  From operating the space shuttle to inverting a small matrix, some thought must be given to *how* the program will do what is needed. In technical terms, we need to define the *algorithm* employed in small-scale programs. In particular, numeric programs need to consider well how calculations are performed. For example, finding the roots of a general polynomial demands a numeric (non-closed form) solution. The choice of algorithm is influenced by the variations in polynomial order and the accuracy demanded.
- *What inputs are required and in what forms?*
  Most programs interact with humans and other programs. This interaction needs to be clearly specified as to *what* format the data will take and *when* the data need to be requested or arrive.
- *What is the execution environment and what should be in the user interface?*
  Is the program a stand-alone program, calculating the quadratic formula for example, or do the results need to be plotted? In the former case, simple user input is probably all that is needed, but the programmer might want to write the program so that its key components could be used in other programs. In the latter, the program probably needs to be written so that it meshes well with some prewritten graphics environment.
- *What are the required and optional outputs, and what are their formats (printed, magnetic, graphical, audio)?*
  In many cases, output takes two forms: *interactive* and *archival*. Interactive output means that the programs results must be provided to the user or to other programs. The data

format must be defined so that the user can quickly see or hear the programs results. Archival results need to be stored on long-term media, such as disk, so that later interpretation of the file's contents is easy (recall the notion of being able to read tomorrow what is written today) and the reading process is easy.

The answers to these questions help programmers organize their thoughts and can lead to decisions about programming language and operating environment. At this point in the programming process, the programmer should know what the program is to do and for whom the program is written. We do not yet have a clear notion of how the program will accomplish these tasks; that comes down the road. This approach to program organization and design is known as *top–down* design. Here, broad program goals and context are defined first with additional detail filled in as needed. This approach contrasts with *bottom–up* design, where the detail is decided first and then merged into a functioning whole. For programming, top–design makes more sense, but you as well as professional programmers are frequently lured into writing code immediately, which is usually motivated by the desire to get something running and figure out later how to organize it all. That approach is prompted by expediency but usually winds up being more inefficient than a more considered, top–down approach that takes longer to get off the ground but has increased likelihood of working more quickly. The result of defining the programming problem is a *specification*: how the program is structured, what computations it performs, and how it should interact with the user.

An Extended Example: The Game of Life

To illustrate how to organize and write a simple program, let us structure a program that plays *The Game of Life*. Conway's "Game of Life" was popularized in Martin Gardner's Mathematical Games column in the October 1970 and February 1971 issues of *Scientific American*. This game is an example of what is known in computer science as *cellular automata*. An extensive description of the game can be found in *The Recursive Universe* by William Poundstone (Oxford University Press, 1987).

The rules of the game are quite simple. Imagine a rectangular array of square cells that are either empty (no living being present) or filled (a being lives there). As shown in Figure 1.1, each cell has eight neighboring cells. At each tick of the clock, a new generation of beings is produced according to how many neighbors surround a given cell.

- If a cell is empty, fill it if three of its neighboring cells are filled; otherwise, leave it empty.
- If a cell is filled, it
  dies of loneliness if it has zero or one neighbors,
  continues to live if it has two or three neighbors, or
  dies of overcrowding if it has more than three neighbors.

The programming task is to allow the user to "play the game" by letting him or her define initial configurations, start the program, which applies the rules and displays each generation, and stop the game at any time the user wants, returning to the initialization stage so that a new configuration can be tried. To understand the program task, we as programmers need to pose several questions, some of which might be

- What computer(s) are preferred, and what kind of display facilities do they have?
- Is the size of the array arbitrary or fixed?
- Am I the only programmer?

No matter how these questions are answered, we start by forming the program's basic outline. Here is one way we might outline the program in a procedural fashion.

1. Allow the user to initialize the rectangular array or quit the program.
2. Start the calculation of the next generation.
   (a) Apply game rules to the current array.
   (b) Generate a new array.
   (c) Display the array.
   (d) Determine whether the user wants to stop or not.
      i. If not, go back to 2a.
      ii. If so, go to step 1.

Note how the idea of reusing the portion of the program that applies game rules arises naturally. This idea is peculiar to programming languages, having no counterpart in prose (it's like being told at the end of a chapter to reread it!). This kind of *looping* behavior also occurs when we go back and allow the user to restart the program.

This kind of outline is a form of *pseudocode*:* a programming–language-like expression of how the program operates. Note that at this point, the programming process is language independent. Thus, *informal pseudocode* allows us to determine the program's broad structure. We have not yet resolved the issue of how, or if, the array should be displayed: Should it be refreshed as soon as a generation is calculated, or should we wait until a final state is reached or a step limit is exceeded? Furthermore, if calculating each generation takes a fair amount of time, our candidate program organization will not allow the user to stop the program until a generation's calculations have been finished. Consequently, we may, depending on the speed of the computer, want to limit the size of the array. A more detailed issue is how to represent the array internally. These issues can be determined later; programmers frequently make notes at this stage about how the program would behave with this structure. Informal pseudocode should remain in the final program in the form of comments.

Writing a program's outline is not a meaningless exercise. How the program will behave is determined at that point. An alternative would be to ask the user how many generations should be calculated and then calculate all generations and display the results as a movie, allowing the user to go backward, play in slow motion, freeze-frame, and so forth. Our outline will not allow such visual fun. Thus, programmers usually design several candidate program organizations, understand the consequences of each, and determine which best meets the specifications.

1.3 Modular Program Design

We now need to define what the routines are and how they are interwoven to archieve the program's goals. (We will deepen this discussion to include objects and messages when we introduce object-oriented formulations in Sec. 1.7.) What granularity – how large should a routine be – comes with programming experience and depends somewhat on the language

* The use of the word *code* is interesting here. It means program as both a noun and a verb: From the earliest days of programming, what the programmer produced was called code, and what he or she did was "code the algorithm." The origin of this word is somewhat mysterious. It may have arisen as an analogy to Morse code, which used a series of dots and dashes as an alternative to the alphabet. This code is tedious to read but ideal for telegraphic transmission. A program is an alternate form of an algorithm better suited to computation.

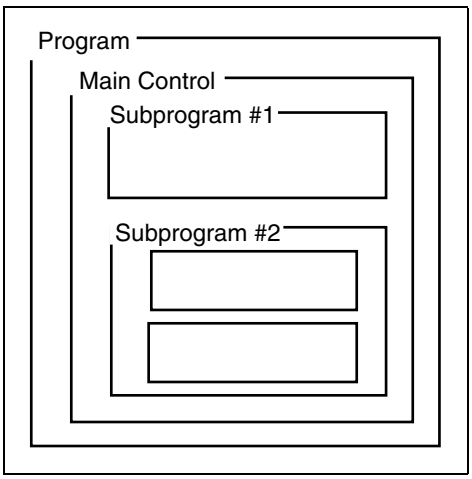

Figure 1.2: Modular program organization relies on self-contained routines in which the passage of data (or messages) from one to the other is very well defined and each routine's (or objects) role in the program becomes evident.

used to express it. A program typically begins with a main segment that controls or directs the solution of the problem by dividing it into subtasks (see Figure 1.2). Each of these may well be decomposed into other routines. This stepwise refinement continues as long as necessary and as long as it benefits program clarity and efficiency. This *modular program design* is the key feature of modern programming design practice. Furthermore, routines can be tested individually and replaced or rewritten as needed. Before actually writing each routine, a job known in computer circles as the *implementation*, the program's organization can be studied: Will the whole satisfy design specifications? Will the program execute efficiently? As the implementation proceeds, each routine's *interface* is defined: How does it interact with its master – the routine that called it – and how are data exchanged between the two? In some languages, this interface can be *prototyped*: the routine's interface – what it expects and what values it calculates – can be defined and the whole program merged and compiled to check for consistency without performing any calculations. In small programs, where you can have these routine definitions easily fitting onto one page, this prototyping can almost be performed visually. In complex programs, where there may be hundreds or thousands of routines, such prototyping really pays off. Once the interfaces begin to form, we ask whether they make sense: Do they exchange information efficiently? Does each routine have the information it needs, or should the program be reorganized so that data exchange can be accomplished more efficiently?

From another viewpoint, you should develop a programming style that "hedges your bets:" programs should be written in such a way that allows their components to be used in a variety of contexts. Again, using a modular programming style, the fundamental components of the calculation should be expressed as a series of subroutines or functions, the interweaving of which is controlled by a main program that reads the input information and produces the output. A modular program can have its components extracted and used in other programs (program reuse) or interfaced to environments. So-called monolithic programs, which tend not to use routines and express the calculation as a single, long-winded program, should not be written.

We emphasize that this modular design process proceeds without actually writing program statements. We use a programming-like language known as *formal pseudocode* to express in prose what routines call others and how. This prose might reexpress a graphic

representation of program organization such as that shown in Figure 1.2. In addition, expressing the program's design in pseudocode eases the transition to program composition, the actual programming process. The components of formal pseudocode at this point are few:

- *comments* that we allow to include the original outline and to describe computational details;
- *functions* that express each routine, whether it be computational or concerned with the user interface;
- *conditionals* that express changing the flow of a program; and
- *loops* that express iteration.

Comments. A comment begins with a comment character, which in our pseudocode we take to be the exclamation point `!`, and ends when the line ends. Comments can consume an entire line or the right portion of some line.

```
! This is a comment: you can read it, but the computer won't
statements
statement ! From the comment character to end of this line is a comment
statements
```

The statements cited in the lines above share the status of the sentence that characterizes most written languages. They are made up of components specific to the syntax of the programming language in use. For example, most programming books begin with a program that does nothing but print "Hello world" on the screen (or other output device). The pseudocode for this might have the following form:

```
! if necessary, include the device library
initiate my program, say main
     send the character string ''Hello world'' to the output device library
terminate my program
```

Figure 1.3 illustrates this in three common languages beginning with F90. At this point it is possible to say we are multilingual in computer languages. Here, too, we may note that, unlike the other two languages shown, in Fortran, when we begin a specific type of software construct, we almost always explicitly declare where we are ending its scope. Here the construct pair was `program` and `end program`, but the same style holds true for `if` and `end if` pairs, for example. All languages have rules and syntax to terminate the scope of some construct, but when several types of different constructs occur in the same program segment, it may be unclear in which order they are terminating.

Functions. To express a program's organization through its component routines we use the notation of mathematical *functions*. Each program routine accepts inputs expressed as arguments of a function, performs its calculations, and returns the computational results as functional values.

```
output_1 = routine (input_1,...,input_m)
```

or

```
call routine (input_1,..., input_m, output_1,..., output_n)
```

```
[1]    !  This is a comment line in Fortran 90
[2]
[3]    program  main               ! a program called main
[4]                                ! begin the main program
[5]      print *,"Hello, world"    ! * means default format
[6]    end program main            ! end the main program

[1]    // This is a comment line in C++
[2]    #include <iostream.h>   // standard input output library
[3]
[4]    main ()                                 // a program called main
[5]                                            // begin the main program
[6]      cout << "Hello, world" << endl ;       // endl means new line
[7]      return 0;                             // needed by some compilers
[8]                                            // end the main program

[1]    %  This is a comment line in MATLAB
[2]
[3]    function main ()                %  a program called main
[4]                                    %  begin the main program
[5]      disp ('Hello, world');        %  display the string
[6]                                    %  end the main program
```

Figure 1.3: 'Hello World' program and comments in three languages.

In Fortran, a routine evaluating a single-output object, as in the first style, is called a *function* and, otherwise, it is called a *subroutine*. Other languages usually use the term function in both cases. Each routines's various inputs and results are represented by *variables*, which, in sharp contrast to mathematical variables, have text-like names that indicate what they contain. These names contain no spaces but may contain several words. There are two conventions for variable names containing two or more words: either words are joined by the underbar character "_" (like `next_generation`) or each word begins with an uppercase letter (like `NextGeneration`). The results of a routines's computation are always indicated by a sequence of variables on the left side of the equals sign `=`. The use of an equals sign does not mean mathematical equality; it is a symbol in our pseudocode that means "assign a routines's results to the variables (in order) listed on the left."

Conditionals. To create something other than a sequential execution of routines, conditionals form a test on the values of one or more variables and continue execution at one point or another depending on whether the test was true or false. That is usually done with the `if` statement. It either performs the instruction(s) that immediately follow (after the `then` keyword) if some condition is valid (like `x>0`) or those that follow the `else` statement if the condition is not true.

```
if test then
  statement group A ! executed if true
else
  statement group B ! executed if false
end if
```

The test here can be very complicated but is always based on values of variables. Parentheses should be used to clarify exactly what the test is. For example,

```
((x > 0) and (y = 2))
```

One special statement frequently found in `if` statements is `stop`: This command means to stop or abort the program – usually with a fatal error message.

Conditionals allow the program to execute nonsequentially (the *only* mode allowed by statements). Furthermore, program execution order can be data-dependent. In this way, how the program behaves – what output it produces and how it computes the output – depends on what data, or messages, it is given. *This means that exact statement execution order is determined by the data, messages, or both, and the programmer – not just the programmer.* It is this aspect of programming languages that distinguishes them from written or spoken languages. An analogy might be chapters in a novel being read in the order specified by the reader's birthday; what that order might be is determined by the novelist through logical constructs. The tricky part is that, in programming languages, each execution order must make sense and not lead to inconsistencies or, at worst, errors: the novel must make sense in all the ways the novelist allows. This data- and message-dependent execution order can be applied at all programming levels from routine execution to statements. Returning to our analogy with the novel, we recall that chapter (routine) and sentence (statement) order depend on the reader's birthday. Such complexity in prose has little utility but does in programming. How else can a program be written that informs the user on what day of the week and under what phase of the moon he or she was born given the birth date?

Loops. Looping constructs in our formal pseudocode take the form of *do loops*, where the keyword `do` is paired with the key phrase `end do` to mean that the expressions and routine invocations contained therein are calculated in order (from top to bottom), then calculated again starting with the first, then again, then again, . . . , forever. The loop ceases only when we explicitly exit it with the `exit` command. The pseudocode loop shown below on the left has the execution history shown on the right.

```
do                                  y = routine_1(x)
  y = routine_1(x)                  z = routine_2(y)
  z = routine_2(y)                  x = routine_3(z) [let's say x=-1]
  x = routine_3(z)                  y = routine_1(x)
  if x > 0 then                     z = routine_2(y)
    exit                            x = routine_3(z) [let's say x=1]
  end if                            [program ends]
end do
```

Infinite loops occur when the Boolean expression always evaluates to true; these are usually not what the programmer intended and represent one type of program error – a "bug."* The constructs enclosed by the loop can be anything: statements, logical constructs, and other loops! Because of this variety, programs can exhibit extremely complex behaviors. How a program behaves depends entirely on the programmer and how his or her definition of the program flows based on user-supplied data and messages. The pseudocode loops are defined in Table 1.1.

* This term was originated by Grace Hopper, one of the first programmers. In the early days of computers, they were partially built with mechanical devices known as relays. A relay is a mechanical switch that controls which way electric current flows: the realization of the logical construct in programming languages. One day, a previously working program stopped being so. Investigation revealed that an insect had crawled into the computer and had become lodged in a relay's contacts. She then coined the term "bug" to refer not only to such hardware failures but to software ones as well since the user becomes upset no matter which occurs.

Table 1.1: Pseudocode Loop Constructs

Loop	Pseudocode

Indexed loop	`do index = b,i,e`
	statements
	`end do`

Pretest loop	`while (test)`
	statements
	`end while`

Posttest loop	`do`
	statements
	`if (test) exit`
	`end do`

1.4 Program Composition

Composing a program is the process of expressing or translating the program design into computer language(s) selected for the task. Whereas the program design can often be expressed as a broad outline, each routine's algorithm must be expressed in complete detail. This writing process elaborates the formal pseudocode and contains more explicit statements that more greatly resemble generic program statements.

Generic programming language elements fall into five basic categories: the four we had before – comments, loops, conditionals, and functions – and statements. We will expand the variety of comments, conditionals, loops, and functions/subroutines that define routines and their interfaces. The new element is the statement, the workhorse of programming. It is the statement that actually performs a concrete computation. In addition to expanding the repertoire of programming constructs for formal pseudocode, we also introduce what these constructs are in MATLAB, Fortran, and C++. As we shall see, formal pseudocode parallels these languages; the translation from pseudocode to executable program is generally easy.

1.4.1 Comments

Comments need no further elaboration for pseudocode. However, programmers are encouraged to make heavy use of comments.

1.4.2 Statements

Calculation is expressed by *statements*, which share the structure (and the status) of the sentence that characterizes virtually all written language. Statements are always executed one after the other as written. A statement in most languages has a simple, well-defined structure common to them all such as

```
variable = expression
```

Statements are intended to bear a great resemblance to mathematical equations. This analogy with mathematics can appear confusing to the first-time programmer. For example, the statement `a = a+1`, which means "increment the variable `a` by one" makes perfect sense as a programming statement but no sense as an algebraic equality since it seems to say that $0 = 1$. Once you become more fluent in programming languages, what is mathematics and what is programming become easily apparent. Statements are said to be *terminated* when a certain

character is encountered by the interpreter or the compiler. In Fortran, the termination character is a carriage return or a semicolon (;). In C++, all statements must be terminated with a semicolon or a comma; carriage returns do not terminate statements. MATLAB statements may end with a semicolon ';' to suppress display of the calculated expression's value. Most statements in MATLAB programs end thusly.

Sometimes, statements become quite long, becoming unreadable. Two solutions to improve clarity can be used: decompose the expression into simpler expressions or use *continuation markers* to allow the statement to span more than one line of text. The first solution requires you to use intermediate variables, which only results in program clutter. Multiline statements can be broken at convenient arithmetic operators, and this approach is generally preferred. In C++, there is no continuation character; statements can span multiple text lines and end only when the semicolon is encountered. In MATLAB, the continuation character sequence comprises three periods '...' placed at the end of each text line (before the carriage return or comment character). In Fortran, a statement is continued to the next line when an ampersand `&` is the last character on the line.

Variables. A *variable* is a named sequence of memory locations to which values can be assigned. As such, every variable has an address in memory, which most languages conceal from the programmer so as to present the programmer with a *storage model* independent of the architecture of the computer running the program. Program variables correspond roughly to mathematical variables that can be integer, real, or, complex-valued. Program variables can be more general than this, being able in some languages to have values equal to a user-defined data type or object which, in turn, contains sequences of other variables. Variables in all languages have *names*: a sequence of alphanumeric characters that cannot begin with a number. Thus, `a`, `A`, `a2`, and `a9b` are feasible variable names (i.e., the interpreter or compiler will not complain about these), whereas `3d` is not. Since programs are meant to be read by humans as well as interpreters and compilers, such names may not lead to program clarity even if they are carefully defined and documented. The compiler and interpreter do not care whether humans can read a program easily or not, but you should: *Use variable names that express what the variables represent*. For example, use `force` as a name rather than `f`; use `i`, `j`, and `k` for indices rather than `ii` or `i1`.

In most languages, variables have *type*: the kind of quantity stored in them. Frequently occurring data types are integer and floating point, for example. Integer variables would be chosen if the variable were only used as an array index; floating point if the variable might have a fractional part.

In addition to having a name, type, and address, each variable has a value of the proper type. The value should be assigned before the variable is used elsewhere. Compilers should indicate an error if a variable is used before it has been assigned a value. Some languages allow variables to have aliases, which are usually referred to as "pointers" or "references." Most higher-level languages also allow programmers to create "user-defined" data types.

Assignment Operator. The symbol = in a statement means *assignment* of the expression into the variable provided on the left. This symbol does not mean algebraic equality; it means that once `expression` is computed, its value is stored in the `variable`. Thus, statements that make programming sense, like `a=a+1`, make no mathematical sense because '=' means different things in the two contexts. Fortran 90 and other languages allow the user to extend the meaning of the assignment symbol (=) to other operations. Such advanced features are referred to as "operator overloading."

Expressions. Just as in mathematics, expressions in programming languages can have a complicated structure. Most encountered in engineering programs amount to a mathematical expression involving variables, numbers, and functions of variables, numbers, or both. For example, the following are all valid statements:

```
A = B
x = sin (2*z)
force = G * mass1 * mass2 / (r*r)
```

Thus, mathematical expressions obey the usual mathematical conventions but with one added complexity: vertical position cannot be used help express what the calculation is; program expressions have only one dimension. For example, the notation $\frac{a}{b}c$ clearly expresses to you how to perform the calculation. However, the one-dimensional equivalent obtained by smashing this expression onto one line becomes ambiguous: Does a/bc mean divide a by b then multiply by c or divide a by the product of b and c? This ambiguity is relieved in program expressions in two ways. The first, the human-oriented way, demands the use of parentheses – grouping constructs – to clarify what is being meant, as in $(a/b)c$. The language-oriented way makes use of *precedence rules*: What an expression means is inferred from a set of rules that specify what operations take effect first. In our example, because division is stronger than multiplication, a/bc means $(a/b)c$. Most people find that frequent reliance on precedence rules leads to programs that take a long time to decipher; the compiler/interpreter is "happy" either way.

Expressions make use of the common arithmetic and relational operators. They may also involve function evaluations; the `sin` function was called in the second expression given in the previous example. Programming expressions can be as complicated as the arithmetic or Boolean algebra ones they emulate.

1.4.3 Flow Control

If a program consisted of a series of statements, statements would be executed one after the other in the order they were written. Such is the structure of all prose, where the equivalent of a statement is the sentence. Programming languages differ markedly from prose in that statements can be meaningfully executed over and over with details of each execution differing each time (the value of some variable might be changed) or some statements skipped with statement ordering dependent on which statements were executed previously or upon external events (the user clicked the mouse). With this extra variability, programming languages can be more difficult for the human to trace program execution than the effort it takes to read a novel. In written languages, sentences can be incredibly complex, much more so than program statements; in programming, the sequencing of statements – program flow – can be more complex.

The basic flow control constructs present in virtually all programming languages are *loops* – repetitive execution of a series of statements – and *conditionals* – diversions around statements.

Loops. Historically, the loop has been a major tool in designing the flow control of a procedure, and one would often code a loop segment without giving it a second thought. Today, massively parallel computers are being widely used, and one must learn to avoid coding explicit loops in order to take advantage of the power of such machines. Later we will review which intrinsic tools are included in F90 for use on parallel (and serial) computers to offer improved efficiency over explicit loops.

The loop allows the programmer to repeat a series of statements, and parameter – the *loop variable* – takes on a different value for each repetition. The loop variable can be an integer or a floating-point number. Loops can be used to control iterative algorithms such as the Newton–Raphson algorithm for finding solutions to nonlinear equations, to accumulate results for a sequential calculation, or merely to repeat a program phrase such as awaiting for the next typed input. Loops are controlled by a `logical expression`, which when evaluated to `true` allows the loop to execute another iteration and when false terminates the loop and commences program execution with the statement immediately following those statements enclosed within the loop.

There are three basic kinds of looping constructs, the choice of which is determined by the kind of iterative behavior most appropriate to the computation. The *indexed loop* occurs most frequently in programs. Here, one loop variable varies across a range of values. In pseudocode, the index's value begins at `b` and increments each time through the loop by `i`; the loop ends when the index exceeds `e`. For example,

```
do j = b, e, i
```

or through the default increment of unity:

```
do j = b, e
```

As an example of an indexed loop, let us explore summing the series of numbers stored in the array `A`. If we know the number of elements in the array when we write the program, the sum can be calculated explicitly without using a loop as follows:

$$\texttt{sum} = \texttt{A}_1 + \texttt{A}_2 + \texttt{A}_3 + \texttt{A}_4$$

However, we have already said that our statements must be on a single line, and so we need a way to represent the subscript attached to each number. We develop the convention that a subscript is placed inside parentheses like

```
sum = A(1) + A(2) + A(3) + A(4)
```

Such programs are very inflexible, and this *hard-wired* programming style is discouraged. For example, suppose in another problem the array contains 1,000 elements. With an indexed loop, a more flexible, easier to read program can be obtained. Here, the index assumes a succession of values, its value tested against the termination value *before* the enclosed statements are executed with the loop terminating once this test fails to be true. The following generic indexed loop also sums array elements but in a much more flexible, concise way.

```
sum = 0
for i = 1, n
  sum = sum + A(i)
end for
```

Here, the variable `n` does *not* need to be known when the program is written; this value can wait until the program executes and can be established by the user or after data are read.

In F90 the extensive support for matrix expressions allows *implicit loops*. For example, consider the calculation of $\sum_{i=1}^{N} x_i y_i$. The language provides at least three ways of performing this calculation. If it is assumed the vectors `x` and `y` are column vectors,

```
1.  sum_xy = 0
    N = size (x)
    do i = 1,N
       sum_xy = sum_xy + x(i)*y(i)
    end do
```

```
2.  sum_xy = sum (x*y)

3.  sum_xy = dot_product (x,y)
```

The first method is based on the basic loop construct and yields the slowest-running program of the three versions. In fact, avoiding the do statement by using implicit loops will almost always lead to faster running programs. The second and third statements employ intrinsic functions, operators designed for arrays, or both. In many circumstances, calculation efficiency and clarity of expression must be balanced. In practice, it is usually necessary to set aside memory to hold subscripted arrays, such as x and y above, before they can be referenced or used.

Conditionals. Conditionals test the veracity of logical expressions and execute blocks of statements accordingly (see Table 1.2). The most basic operation occurs if we want to execute a series of statements when a logical expression, say test, evaluates to true. We call that a simple if conditional; the beginning and end of the statements to be executed when test evaluates to true are enclosed by special delimiters, which differ according to language. When only one statement is needed, C++ and Fortran allow that one statement to end the line that begins with the if conditional. If you want one group of statements to be executed when test is true and another set to be executed when false, you use the if-else construct. When you want to test a series of logical expressions that are not necessarily complementary, the nested-if construct allows for essentially arbitrarily complex structure to be defined. In such cases, the logical tests can interlock, thereby creating programs that are quite difficult to read. Here is where program comments become essential. For example, suppose you want to sum only the positive numbers less than or equal to 10 in a given sequence. Let us assume the entire sequence is stored in array A. In informal pseudocode, we might write

```
loop across A
  if A(i) > 0 and A(i) < = 10 add to sum
end of loop
```

More formally, this program fragment as a complete pseudocode would be

Table 1.2: Syntax of Pseudocode Conditionals

Conditional	Pseudocode
if	`if (test) statement`

if	`if (test) then`
	`  statements`
	`end if`

if-else	`if (test) then`
	`  statements A`
	`else`
	`  statements B`
	`end if`

nested if	`if (test1) then`
	`  statements A`
	`  if (test2) then`
	`  statements B`
	`  end if % end of test2`
	`end if`