FORTRAN for Scientists and Engineers

Second Edition

Gary Bronson

Fairleigh Dickinson University

Scott/Jones Inc. Publishers
P.O. Box 696
El Granada, CA 94018
FAX (415) 726-4693
scotjones2@aol.com

Contributing Editor: Dr. Emil Neu, Stevens Institute of Technology

FORTRAN for Scientists and Engineers, Second Edition (previously Modular Fortran 77)

Gary Bronson

Scott/Jones Inc. Publishers

ISBN 1-881991-39-3

Book Production: Cecelia G. Morales
Book Manufacturing: Data Reproduction
Cover Design: Vicki Lin

Printed in the United States of America

5 4 3 2 1 V W X Y Z

ADDITIONAL TITLES OF INTEREST FROM SCOTT/JONES

The DOS-6 Coursebook
by Forest Lin

Quickstart in DOS (120 pages)
by Forest Lin

The 1-2-3 Coursebook
The 1-2-3 Primer
both by Forest Lin

The DOS Primer (covers versions 3 and 5)
by Dorothy Calvin

Modern FORTRAN 77/90: Alternate Edition
by Gary Bronson

Assembly Language for the IBM PC Family
by William Jones

C by Discovery, Second Edition (emphasizing ANSI C)
by L. S. Foster

WordPerfect 6.0a for Windows
by Rolayne Day

The Windows Textbook
by Stewart Venit

Quickstart in Windows
by Stewart Venit

Quickstart in C++
by William Jones

The Visual Basic Coursebook
by Forest Lin

Preface

The successes of both *Modular FORTRAN 77* and *Modern FORTRAN 77/90* and the resulting user feedback from these texts provided invaluable insight into the needs and diversity of the community of FORTRAN instructors.

On the positive side, there was almost universal agreement that FORTRAN is still one of the most powerful and easy to use high-level computer languages available for engineering and scientific applications. Certainly this viewpoint is significantly enhanced with the addition of pointers and targets in the new FORTRAN 90 standard, which adds to FORTRAN the additional features of dynamic memory allocation that Pascal and C provide.

The second, almost universally agreed upon point was that FORTRAN should be taught using the modular programming techniques that have been developed over the thirty years since FORTRAN was commercially produced. These, of course, are the same principles emphasized in all structured languages. But there the agreement seemed to end and widely varying opinions began.

For example, about half of the professors helping in the development of the previous texts felt that formatting should not be introduced until late in the text; while the other half insisted that formatting should be introduced early. Similarly, many reviewers indicated that they required the use of the WRITE statement, while just as many insisted that the WRITE statement be avoided in favor of the PRINT statement, to sidestep the need for explicit output unit numbers.

Yet nowhere was this diversity of opinion more apparent than regarding how modularity should be introduced. Many FORTRAN instructors seem to either want, or be willing to tolerate, an introduction of modular concepts as early as possible in the course, but under a host of conditions. Others insisted that modular concepts be introduced on "day one" of their course, and sustained throughout the semester. This lead to two distinct books—*Modular FORTRAN 77*, which provided a more flexible approach to when modular concepts had to be introduced, and *Modern FORTRAN 77/90*, which adhered to a "day one" approach. This edition is a continuation of the more flexible approach adopted by *Modular FORTRAN 77*, but with many new features. Based on this approach, the introduction of modular concepts was written so that a variety of professors could mold it to their own way of teaching and to the requirements of each class.

Features Providing Flexibility

Modularity. This book, like its *Modular FORTRAN 77* predecessor, was written to provide a flexible tool to allow each instructor to teach FORTRAN as a modular language to the extent and at the pace desired. As one reviewer of the first edition put it, "The book has successfully managed in Chapter One to introduce the concept of modularity in a painless and effective manner." After Chapter One, the book provides a flexible tool that each professor can use in a variety of

ways, depending on *how much* modularity they want to introduce, and *when* they want to introduce it. The Enrichment Section at the end of Chapter Two provides the means for introducing the passing of arguments early in the course.

PRINT and WRITE Statements and FORMATTING. The sections on the WRITE statement, PRINT statement, and formatting provide the means of introducing these topics early. For more advanced classes, any or all of these topics may be introduced early in the course. Equally as effective, any or all of these topics may be skipped until the classroom environment is ready for them. I would not myself teach the PRINT and WRITE statements "back-to-back," but I have tried to structure the text so that those who prefer to cover the WRITE statement early can omit the PRINT statement, and those who would cover the PRINT statement early can omit the WRITE statement. And in *either* case, explicit formatting can either be initially included or omitted in favor of simple list-directed output.

New and Distinctive Features of this Book

In addition to the prime goal of providing flexibility to each instructor, this edition contains a number of new and distinct features. These include over 100 new engineering-oriented problems, a section on the importance of libraries, and a new chapter on matrices and Gaussian Elimination techniques. Additionally, the following features, some of which have been retained from the first edition, are contained in this text:

ANSI FORTRAN 90 Features. Unlike the first edition, in which FORTRAN 77 was used throughout the text, FORTRAN 90 features are integrated and highlighted throughout this edition. Although the new standard recommends that, except for FORTRAN 66 and FORTRAN 77, the name of the language be spelled as Fortran, we will use the notation FORTRAN 77 and FORTRAN 90 for consistency.

Lab Projects. Chapter 15 contains a set of lab exercises for the prior chapters. Although some of these projects were contained in Modular FORTRAN 77/90, most of them are new. All of them require a deeper understanding of FORTRAN and necessitate an integration of input, processing, and output concepts for their completion. They require the students to prepare a documented and formatted report in a professional manner. Both sample data and report structures are provided.

Enrichment Sections. Given the many different emphases that can be applied in teaching FORTRAN, I have added a number of Enrichment Sections to most of the chapters in this text. These allow you to provide different emphasis with different students or different FORTRAN class sections.

Applications. Engineering and scientific examples are used throughout the text to both motivate and illustrate concepts presented in the text. Additionally, the majority of the chapters have a section consisting of two specific applications relating to the material presented in the chapter. I believe the mix of applications both heighten the interest of students and reinforce software engineering concepts.

Exercises. A wide range of exercises are included at the end of almost every section, rather than just at the end of each chapter. They range from skill builders, to programming assignments, to debugging exercises. In addition there are many program modification assignments and over 100 new engineering oriented problems.

Program Testing. Every single FORTRAN program in this text has been successfully compiled and executed by myself. The majority of these are included on the diskette provided with the text..

Comparative Charts for Different Compilers. Throughout this text I have tried to allow for differences between the different computing environments in which FORTRAN can be taught. As a result, I have displayed these differences in a table whenever a significant variation seemed to occur.

Readability. The one thing I have found most important in my own teaching is *regardless of the subject being written about, it must be written so that students can read it.* As a result, I have taken every precaution for this material to be clear, unambiguous, and deliberate.

Acknowledgements

FORTRAN for Scientists and Engineers, 2nd Edition, is a direct result of the success (and the limitations) of *Modular FORTRAN* 77 and its "sister" edition *Modern FORTRAN 77/90*.

Users of these texts who provided feedback from their teaching experiences made an invaluable contribution to the quality of this text. They include Shui Lam (Cal State Long Beach); Mary Kay Frohock (University of Kansas); Kris Froehlke (Indianapolis University/Purdue University); Josef Zurada (University of Louisville); Robert Kenyon (University of Illinois at Chicago); Terry Thul (New Mexico State University); Randy Odendahl (SUNY Oswego); Chris Connant (Broome Community College); Martha Tillman (College of San Mateo); and Scott Bailey, Peter Smith, and Ginter Trybus (Cal State Northridge).

Personal telephone conversations with five users of *Modular FORTRAN* 77 also had a profound impact on my preparation of this text. I deeply appreciate the time and thoughts of Josan Duane (Ohio State University); S. Srinivasan (University of Louisville); Richard Martin (Southwest Missouri State); Neil Sorensen (Weber State University); and Kent Dunham (University of Idaho). Many of their suggestions for improving *Modular FORTRAN* 77, which were incorporated into *Modern FORTRAN 77/90*, have also been incorporated into this edition.

I was also fortunate to have four consulting editors whose teaching environments and orientations were different and complimentary to my own. Each of them contributed trenchant and thoughtful criticisms, as well as original work of their own, towards completion of this text. It has been a privilege to work with them: Judy Cain (Tompkins-Cortland Community College); John Lyon (University of Arizona); Wesley Scruggs (Brazoport College); and Howard Silver (Fairleigh Dickinson University). Additionally, I would like to thank Jerry Wolfe, of Wolfe and Hurst, Inc., for graciously explaining the technical aspects of his applications.

Perhaps most fortuitous was a reintroduction to an unusually thoughtful, delightful, and gifted engineer from Stevens Institute of Technology, who graciously agreed to become a contributing editor to this edition. My education at Stevens was both professionally and personally rewarding. Over time, however, I had forgotten how very fortunate I was in having a marvelous group of truly gifted and talented professors—not only in electrical engineering, but also in mathematics, the other engineering disciplines, humanities, and even physical education. My education at Stevens was all I could wish for—an introduction to a lifetime journey—and I would like to especially thank all of my teachers. I have the privilege of citing Dr. Emil Neu, my contributing editor, as one example of the excellence with which I was surrounded. Thank you, Professor Neu.

It has also been a privilege to work with my friend and publisher Richard Jones, who has been in spirit what every author wishes a publisher to be: a partner. Finally, the task of turning the final manuscript into a textbook again depended on many people other than myself. For this I especially want to thank the copy editor, Sheryl Strauss, and the compositor, Cecelia Morales. The dedication of these two people, their attention to detail, and their high standards, have helped in many ways to improve the quality of this book. Almost from the moment the book moved to the production stage these two individuals seemed to take personal ownership of the text, and I am very grateful to them.

No acknowledgement would be complete without also mentioning the direct encouragement and support provided by the Vice President of Academic Affairs at Fairleigh Dickinson University, Dr. Geoffrey Weinman, my Dean, Dr. Paul Lerman, and my Chairman Dr. Naadimuthu. Without their support, this text could not have been written.

Finally, I deeply appreciate the patience, understanding, and love provided by my friend, wife, and partner, Rochelle.

Gary Bronson

Dedicated to Rochelle, Matthew, Jeremy, David, and Sparky

Contents

1 Getting Started

Chapter One

1.1 Introduction to Programming

A computer is a machine and like other machines, such as an automobile or lawn mower, it must be turned on and then driven, or controlled, to do the task it was meant to do. In an automobile, for example, control is provided by the driver, who sits inside of and directs the car. In a computer, the driver is a set of instructions, called a program. More formally, a *computer program* is a sequence of instructions that is used to operate a computer to produce a specific result. *Programming* is the process of writing these instructions in a language that the computer can respond to and that other programmers can understand. The set of instructions that can be used to construct a program is called a *programming language.*

On a fundamental level, all computer programs do the same thing; they direct a computer to accept data (input), to manipulate the data (process), and to produce reports (output). This implies that all computer programming languages must provide essentially the same capabilities for performing these operations. This is indeed the case and the fundamental set of instructions provided by such high-level procedure-oriented computer languages as FORTRAN, BASIC, COBOL, and Pascal is listed in Table 1.1. The term *high-level* refers to the fact that the

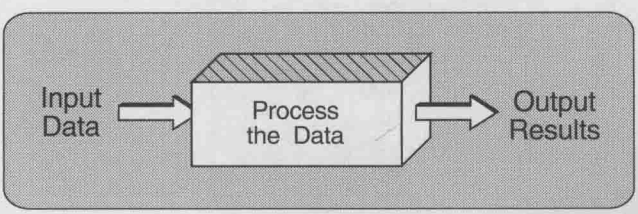

Figure 1-1 All Programs Perform the Same Operations

Table 1-1 Programming Language Instruction Summary

Operation	FORTRAN	BASIC	COBOL	Pascal
Input (get the data)	READ READ/DATA	INPUT ACCEPT	READ READLN	READ
Processing (use the data) ,	= IF/ELSE DO	LET IF/ELSE FOR	COMPUTE IF/ELSE PERFORM	:= IF/ELSE FOR WHILE REPEAT
Output (display the data)	WRITE PRINT	PRINT PRINT/USING	WRITE DISPLAY	WRITE WRITELN

statements in these languages resemble English statements. The term *procedure-oriented* refers to the fact that these languages are primarily used to describe procedures for producing specific results.

If all programming languages provide essentially the same features, why are there so many of them? Simply because there are vast differences in the types of input data, calculations needed, and required output reports. For example, scientific and engineering applications usually require high-precision numerical outputs, accurate to many decimal places. In addition, these applications typically use many algebraic or trigonometric formulae to produce their results. For example, calculating the bacteria concentration level in a polluted pond, as illustrated in Figure 1-2, requires evaluation of an exponential equation. For such applications, the FORTRAN programming language, with its algebra-like instructions, is ideal. FORTRAN, whose name is an acronym derived from FORmula TRANslation, was commercially introduced in 1957 and was originally designed for translating formulae into computer-readable form. It was the first high-level language to be

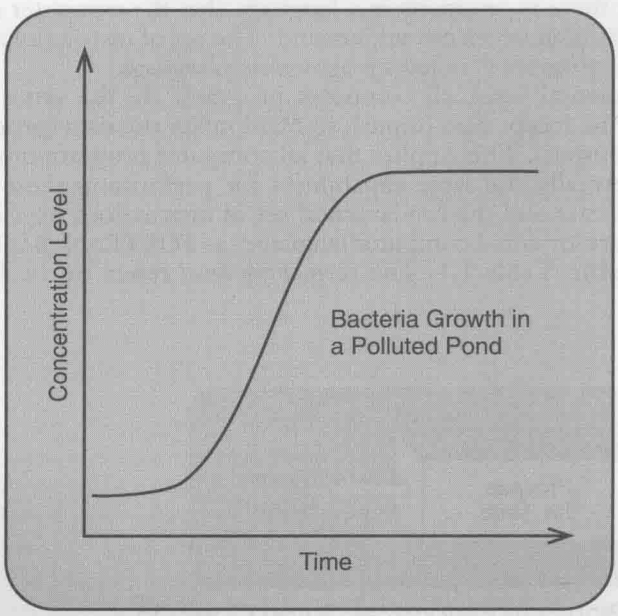

Figure 1-2 FORTRAN is Ideal for Scientific and Engineering Applications

developed. The current standard for FORTRAN, commonly referred to as FORTRAN 90, is maintained by the American National Standards Institute (ANSI).

Algorithms

Before a program is written, the programmer must have a clear understanding of what the desired result is and how the proposed program is to produce it. In this regard, it is useful to realize that a computer program describes a computational procedure.

In computer science, a computational procedure is called an algorithm. More specifically, an *algorithm* is defined as a step-by-step sequence of instructions that describes how a computation is to be performed. In essence, an algorithm answers the question, "What method will you use to solve this computational problem?" Only after we clearly understand the algorithm and know the specific steps required to produce the desired result can we write the program. Seen in this light, programming is the translation of the selected algorithm into a language that the computer can use.

To illustrate an algorithm, we shall consider a simple requirement. Assume that a program must calculate the sum of all whole numbers from 1 through 100. Figure 1-3 illustrates three methods we could use to find the required sum. Each method constitutes an algorithm.

Method 1. Columns: Arrange the numbers from 1 to 100 in a column and add them:

```
   1
   2
   3
   4
   .
   .
  98
  99
+100
----
5050
```

Method 2. Groups: Arrange the numbers in convenient groups that sum to 100. Multiply the numbers of groups by 100, then add any unused numbers to the total:

```
 0 + 100 = 100
 0 +  99 = 100
 2 +  98 = 100
 3 +  97 = 100
 .       .
 .       .
 .       .
49 +  51 = 100
50 +   0 =  50
```

50 groups

$(50 \times 100) + 50 = 5050$

One unused number

Method 3. Formula: Use the formula Sum = n(a + b)/2 where

n = number of terms to be added (100)
a = first number to be added (1)
b = last number to be added (100)

Sum = 100(1 + 100)/2

Figure 1-3 Summing the Numbers 1 Through 100

Clearly, most people would not bother to list the possible alternatives in a detailed step-by-step manner, as is done in Figure 1-3, and then select one of the algorithms to solve the problem. But then, most people do not think algorithmically; they tend to think heuristically or intuitively. For example, if you had to change a flat tire on your car, you would not think of all the steps required—you would simply change the tire or call someone else to do the job. This is an example of heuristic thinking.

Unfortunately, computers do not respond to heuristic commands. A general statement such as "add the numbers from 1 to 100" means nothing to a computer, because the computer can only respond to algorithmic commands written in an acceptable language such as FORTRAN. To program a computer successfully, you must clearly understand this difference between algorithmic and heuristic commands. A computer is an "algorithm-responding" machine; it is not a "heuristic-responding" machine. You cannot tell a computer to change a tire or to add the numbers from 1 through 100. Instead, you must give the computer a detailed step-by-step set of instructions that, collectively, forms an algorithm. For example, the set of instructions

```
Set n equal to 100
Set a = 1
Set b equal to 100
Calculate sum = (n(a + b)) / 2
Print the sum
```

forms a detailed method, or algorithm, for determining the sum of the numbers from 1 through 100. Notice that these instructions are not a computer program. Unlike a program, which must be written in a language the computer can respond to, an algorithm can be written or described in various ways. When English-like phrases are used to describe the algorithm (processing steps), as in this example, the description is called *pseudocode*. When mathematical equations are used, the description is called a *formula*. When pictures that employ specifically defined shapes are used, the description is referred to as a *flowchart*. A flowchart provides a pictorial representation of the algorithm using the symbols shown in Figure 1-4. Figure 1-5 illustrates the use of these symbols in depicting an algorithm for determining the average of three numbers.

Because flowcharts are cumbersome to revise, the use of pseudocode to express the logic of an algorithm has gained increasing acceptance in recent years among programmers. Unlike flowcharts, where standard symbols are defined, there are no standard rules for constructing pseudocode. In describing an algorithm using pseudocode, short English phrases are used. For example, acceptable pseudocode for describing the steps needed to compute the average of three numbers is:

Input the three numbers into the computer
Calculate the average by adding the numbers and dividing the sum by three
Display the average

Only after an algorithm has been selected and the programmer understands the steps required can the algorithm be written using computer-language statements. When computer-language statements are used to describe the algorithm, the description is called a *computer program*.

From Algorithms to Programs

After an algorithm has been selected, it must be converted into a form that can be used by a computer. The conversion of an algorithm into a computer program,

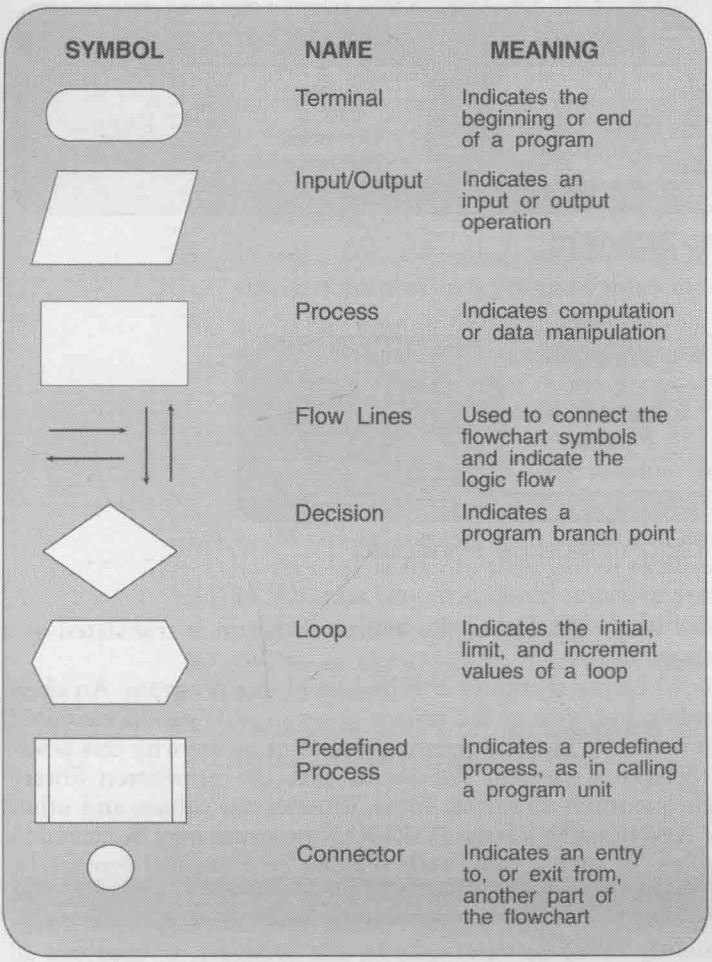

Figure 1-4 Flowchart Symbols

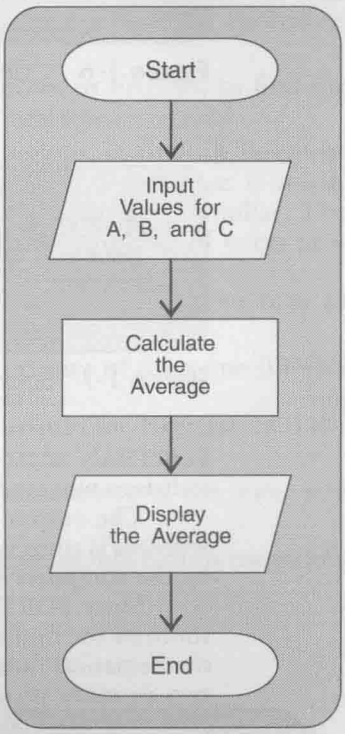

Figure 1-5 Flowchart for Calculating the Average of Three Numbers

using a language such as FORTRAN, is called *coding* the algorithm (see Figure 1-6). Much of the remainder of this text is devoted to showing you how to code algorithms into FORTRAN.

Program Translation

Once a program is written in FORTRAN it still cannot be executed on a computer without further translation. This is because the internal language of all computers consist of a series of 1s and 0s, called the computer's *machine language*. To generate a machine language program that can be executed by the computer requires that the FORTRAN program, which is referred to as a *source program*, be translated into the computer's machine language (see Figure 1-7).

The translation into machine language can be accomplished in two ways. When each statement in a high-level-language source program is translated individually and executed immediately, the programming language used is called an *interpreted language*, and the program doing the translation is called an *interpreter*.

When all of the statements in a source program are translated before any one statement is executed, the programming language used is called a *compiled language*. In this case, the program doing the translation is called a *compiler*.

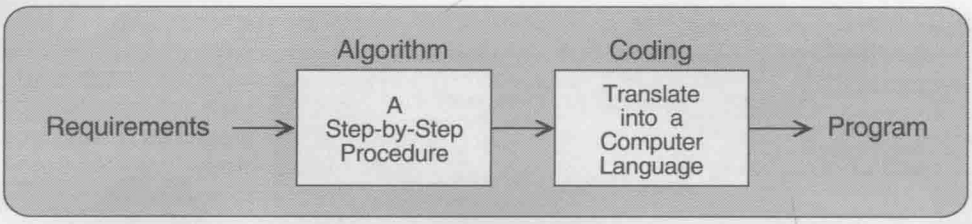

Figure 1-6 Coding an Algorithm

Figure 1-7 Source Programs Must Be Translated

FORTRAN is a compiled language. Here, the source program is translated as a unit into machine language.

The output produced by the compiler is called an object program. An *object program* is simply a translated version of the source program that can be executed by the computer system with one more processing step. Let us see why this is so.

Most FORTRAN programs contain statements that use prewritten library routines for finding such quantities as square roots, exponential values, and other mathematical functions. Additionally, a large FORTRAN program may be stored in two or more program files. In such a case, each file can be compiled separately. However, both files must ultimately be combined to form a single program before the program can be executed. In both of these cases it is the task of a linker program, which is frequently called automatically by the compiler, to combine all of the library routines and individual object files into a single program ready for execution. This final program is called an *executable program.* (See Appendix A for a complete description of entering, compiling, and running a FORTRAN program.)

Skill Builder Exercises

1. Determine a step-by-step procedure (list the steps) to do these tasks: *(Note: There is no one single correct answer for each of these tasks. The exercise is designed is to give you practice in converting heuristic commands into equivalent algorithms and making the shift between the thought processes involved in the two types of thinking.)*
   a. Fix a flat tire
   b. Make a telephone call
   c. Log in to a computer
   d. Roast a turkey
2. Are the procedures you developed for Exercise 1 algorithms? Discuss why or why not.
3. Determine and write an algorithm (list the steps) to interchange the contents of two cups of liquid. Assume that a third cup is available to hold

the contents of either cup temporarily. Each cup should be rinsed before any new liquid is poured into it.

4. Write a detailed set of instructions, in English, to calculate the resistance of the following resistors connected in series: n resistors, each having a resistance of 56 ohms, m resistors, each having a resistance of 33 ohms, and p resistors, each having a resistance of 15 ohms. Note that the total resistance of resistors connected in series is the sum of all individual resistances.
5. Write a set of detailed, step-by-step instructions, in English, to find the smallest number in a group of three integer numbers.
6. a. Write a set of detailed, step-by-step instructions, in English, to calculate the change remaining from a dollar after a purchase is made. Assume that the cost of the goods purchased is less than a dollar. The change received should consist of the smallest number of coins possible.
   b. Repeat Exercise 6a, but assume the change is to be given only in pennies.
7. a. Write an algorithm to locate the first occurrence of the name JONES in a list of names arranged in random order.
   b. Discuss how you could improve your algorithm for Exercise 7a if the list of names was arranged in alphabetical order.
8. Write an algorithm to determine the total occurrences of the letter *e* in any sentence.
9. Determine and write an algorithm to sort four numbers into ascending (from lowest to highest) order.

1.2 Introduction to Modularity

A well-designed program is constructed using a design philosophy similar to that used to construct a well-designed building; it doesn't just happen, but depends on careful planning and execution for the final design to accomplish its intended purpose. Just as an integral part of the design of a building is its structure, the same is true for a program.

In programming, the term *structure* has two interrelated meanings. The first meaning refers to the program's overall construction, which is the topic of this section. The second meaning refers to the form used to carry out the individual tasks within the program, which is the topic of Chapters 4 and 5. In relation to its first meaning, programs whose structure consists of interrelated segments, arranged in a logical and easily understandable order to form an integrated and complete unit, are referred to as *modular programs* (Figure 1-8). Not surprisingly, it has been found that modular programs are noticeably easier to develop, correct, and modify than programs constructed otherwise. In general programming terminology, the smaller segments used to construct a modular program are referred to as *modules*.

In a modular program each module is designed and developed to perform a clearly defined and specific function. This function can be tested and modified without disturbing other modules in the program. The final program is then constructed by connecting as many modules as necessary to produce the desired

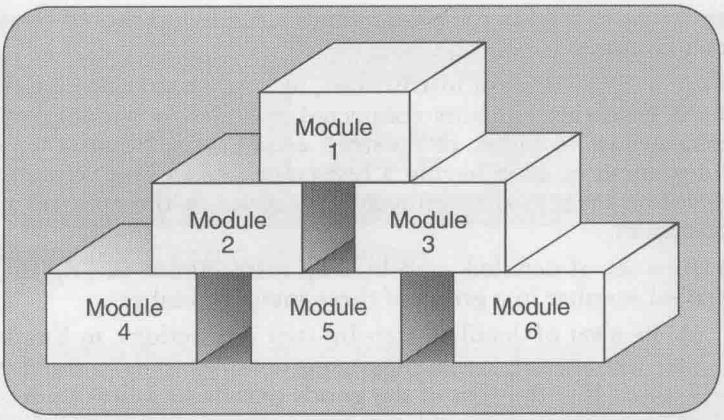

Figure 1-8 A Well-Designed Program Is Built Using Modules

result. Unfortunately, each programming language has its own specific name for modules. In FORTRAN, a module is referred to as a *program unit.*

Program Units

A program unit is essentially a small program in its own right. As such, a program unit must be capable of doing what is required of all programs: receive data, operate on the data, and produce a result (see Figure 1-9). Unlike a larger program, however, a program unit performs only limited operations. Typically, each program unit performs a single task required by the larger program of which it is a part.

A complete program is constructed by combining as many program units as necessary to produce the desired result. The advantage to this modular construction is that the overall design of the program can be developed before any single program unit is written. Once the requirements for each program unit are finalized, each unit can be programmed and integrated within the overall program as it is completed.

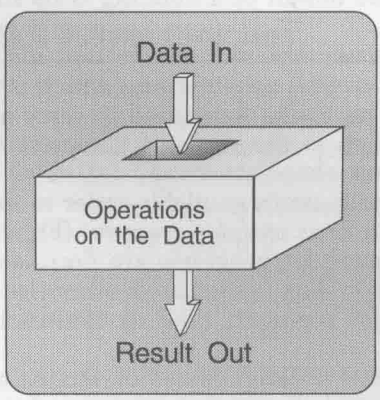

Figure 1-9 A Program Unit Receives Data, Operates on the Data, and Produces a Result

FORTRAN provides three common types of program units: the MAIN, SUBROUTINE, and FUNCTION types.[1] Each of these program unit types performs a specific type of task. We shall learn and use all of these unit types as we progress.

It is useful to think of a program unit, regardless of its type, as a small machine that transforms the data it receives into a finished product. For example, Figure 1-10 illustrates a program unit that accepts three numbers and calculates their average to produce an output.

The MAIN Program Unit

A distinct advantage to using program units in FORTRAN is that we can plan in advance the overall structure of the program, including making provisions for testing and verifying the operation of individual units. We first determine the individual tasks required of each unit, and establish how the units will be combined. Only after the overall structure of the program has been designed is each program unit written to perform its required task.

To provide for the orderly placement and execution of individual program units, every FORTRAN program must have one, and only one, MAIN program unit (Figure 1-11). This MAIN unit is frequently referred to as the *driver unit*, due to its function of telling all other program units the sequence in which they are to be executed.

Figure 1-12 illustrates a complete MAIN program unit. The first line in the program unit, PROGRAM TEST, is called a *header line*. The word PROGRAM in the header line identifies the beginning of a MAIN program unit. The word TEST is a user-selected name for this MAIN unit. The rules for choosing your own program unit names are presented at the end of this section.

The end of a MAIN unit is always designated by the word END, written on a line by itself. Both words, PROGRAM and END, are examples of FORTRAN keywords. A *keyword* is a word that takes on a special meaning when it is used in a

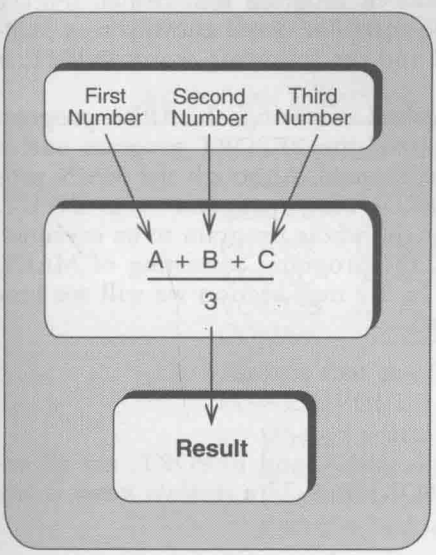

Figure 1-10 A Program Unit That Averages Three Numbers

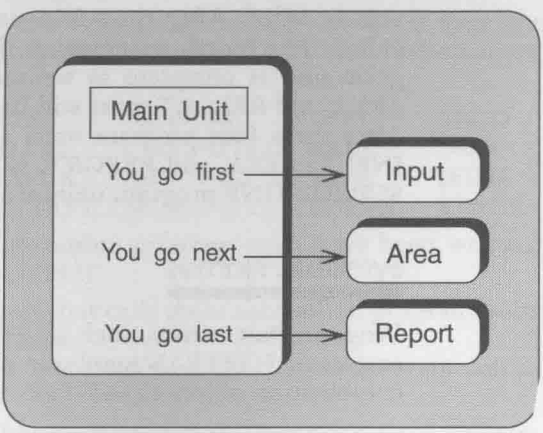

Figure 1-11 The MAIN Program Unit Directs All Other Units

[1] A fourth type, BLOCK DATA , described in Section 9.3, is rarely used.