INTRODUCTION TO STRUCTURED PROGRAMMING USING TURBO PASCAL® VERSION 5.0 ON THE IBM PC

KENNETH J. MORGAN

DeVry Institute of Technology–Atlanta

Merrill Publishing Company
A Bell & Howell Information Company
Columbus Toronto London Melbourne

Cover: Computer-generated art, "Depth of Feel," by Jim Dixon, Pacific Data Images, 1986.

Published by Merrill Publishing Company
A Bell & Howell Information Company
Columbus, Ohio 43216

This book was set in Melior.

Administrative Editor: Vern R. Anthony
Production Editor: Mary Harlan
Art Coordinator: Pete Robison
Cover Designer: Cathy Watterson
Text Designer: Cynthia Brunk

Library of Congress Catalog Card Number: 89-61699
International Standard Book Number: 0-675-20770-3
Printed in the United States of America
1 2 3 4 5 6 7 8 9—94 93 92 91 90

1 Structured Programming

How do you write a computer program? Part I answers this question, but it takes all of Part I to answer it completely. Learning Pascal syntax, though important, is not the primary goal of a first course in programming. A computer language can be easily learned, and the statements and rules of Turbo Pascal are thoroughly explained in Part II. What the beginning programming student must emphasize is quite simply the *how* of programming: how to develop the solution to a given problem, how to organize a program, and how to make effective use of the standard techniques that represent the "tricks" of the trade.

This first chapter of Part I introduces you to the concept of *structured programming*: its definition, history, and rationale. The author and this book are dedicated to teaching this method of programming. Then, in chapter 2, the nature of the task of programming a computer is discussed, together with a very important list of six steps that guide you through that task. They are explicitly based on the philosophy of structured programming. Finally, chapters 3 through 12 explain in some detail how to perform these six steps. Many programming examples are given in these chapters, and you are encouraged to read the sections in Part II that explain the rules of syntax for the Pascal statements used in these examples. As emphasized in the Preface, Parts I and II are designed to be studied simultaneously.

1.1 STRUCTURED PROGRAMMING: HISTORY AND RATIONALE

A computer program is simply a set of instructions that directs the computer in its calculations and movement of data. During the 1950s and 1960s, however, few people actually thought about the question of developing a general method for organizing a program. Instead, programs were written much the way you might, off the top of your head, give someone instructions on how to change a tire or, worse, build a house. When a program did not work, it was simply "patched" (fixed), error by error, until the programmer felt that he had found them all. Usually, however, errors continued to appear even after the program was released for use.

Such an approach to programming was tolerable on first and second generation computers (until about 1964). However, with the advent of the third generation during the middle and late 1960s, hardware became more powerful than the software that ran it. The "hit-and-miss" approach to programming could not keep pace. To obtain the full benefit of such third generation computers as the IBM System/360, an operating system and, for some applications, even individual programs had to contain thousands of instructions, often written by many different people. No systematic approach to program organization had been developed to handle tasks of such magnitude. The result was called a "software crisis" during a NATO conference convened to address the problem.[1] Two "horror stories" illustrate this point.

One example is the operating system that IBM developed for the System/360. Called OS/360, it cost hundreds of millions of dollars to develop, was more than a year late, and contained thousands of errors. After many releases, hundreds of errors probably still remained when the System/370 was introduced.

This was not an isolated case. Studies have shown that, in general, programmers averaged only 5 to 10 completely debugged source statements (programming lines) per day.[2] This would seem to be an incredibly low output, and indeed it was. To pick up a pencil and write 10 instructions takes less than 10 minutes. Of course, the reason for this low output was obvious to everyone: too much time was spent debugging programs of the errors introduced at the time they were written. Program language errors ("syntax errors") detected by compilers are easily and quickly corrected. Most of the debugging time was spent correcting what might be called planning errors and logical errors in the overall design and organization of the program. Prior to the advent of structured programming, this general state of affairs had been considered inevitable: it was simply part and parcel of computer programming.

The second example shows that error-ridden programs can sometimes be a matter of life and death. Astronauts John Young and Robert Crippen were already in the space shuttle *Columbia* when, twenty minutes before launch time, warning lights at Mission Control began to flash. Something was wrong with the computer system on board the shuttle. A check rather quickly showed that there was nothing wrong with the hardware. The problem seemed to be in the programs. Once that was determined, it was clear that the maiden flight of the space shuttle would have to be delayed. Why? The software for those computers consisted of nearly 500,000 lines of elaborately interwoven instructions. "Finding a programming error, a bug, in that webb," writes Steve Olson, "would be like trying to find a single misspelled word in an encyclopedia."[3]

Olson goes on to relate what Edsger Dijkstra, the father of structured programming, had to say about the above incident and the type of programming generally that goes on at NASA:

> "It was precisely the type of error that one would expect. . . . You see, most of NASA's software is full of bugs."
>
> His eyebrows arch with pleasure, a sure sign that a story follows. "I saw the first moon shot in 1969, when Armstrong, Aldrin, and Collins went to the moon, and shortly thereafter I met Joel Aron of IBM's Federal Systems Division, who I knew had been re-

sponsible for a large part of the software. So when I saw Joel at the swimming pool of our hotel in Rome, I said, 'Joel, how did you do it?' 'Do what?' he said. I said, 'Get that software to work okay.' 'Okay?' he said. 'It was full of bugs. In one of the trajectory computations, we had the moon's gravity repulsive rather than attractive, and this was discovered by accident five days before count zero.' "

Dijkstra draws back in his chair, the picture of astonished outrage. "When I had regained my composure, I said, 'Those three guys, they have been lucky.' 'Oh yes,' Joel said."[4]

The history of structured programming began in 1964 at an international colloquium held in Israel. There Corrado Bohm and Guiseppe Jacopini presented a paper (in Italian) that proved mathematically that only three "control structures" were necessary to write any program. This theorem, and control structures in general, is discussed at length in chapter 6. Suffice it to say here that the work of Bohm and Jacopini made the GOTO statement unnecessary in computer programming. The GOTO statement is discussed in chapter 4, but you should realize here that during the mid-1960s virtually no programmer could even conceive of a program written without GOTO statements. Therefore, it is no surprise that even after an English translation of their paper in the trade journal *Communications of the ACM* in 1966, the theorem of Bohm and Jacopini and its implications were almost entirely ignored in the United States.

The turning point, however, occurred in 1968 when Edsger Dijkstra of the Netherlands published a letter to the editor in the *Communications of the ACM*. This letter was given the appropriate title, "Go To Statement Considered Harmful." For over twenty years now Dijkstra has been crusading for a better way of programming—a systematic way to organize programs, called *structured programming*. It can be used with profit for any program but pays enormous dividends on very large programs of the type previously discussed.

Edward Yourdon is another important name in the history of structured programming. By giving seminars on the subject in the mid-1970s, he was the individual most responsible for popularizing the method in the United States. However, its widespread acceptance would probably not have occurred were it not for the tremendous success of the now famous *New York Times* project, which was completed in 1972. This was the first major project using structured programming and its first great success story.

This project, developed for the *New York Times* by a programming team at IBM under the direction of Harlan Mills, was a system to automate the newpaper's clipping file. Using a list of index terms, users could browse through abstracts of all the paper's articles and then retrieve the full-length articles of their choice from microfiche for display on a terminal screen.

The task took 22 months, included about 83 000 lines of code, and involved approximately 11 man-years of effort. The file processing system passed a week of acceptance testing without error and ran for 20 months until the first error was detected. In the first 13 months, only one program error resulted in system failure. The system-control programmers achieved about 10 000 lines of source code and one error per man-year.[5]

In the entire system, only 21 errors were found during five weeks of acceptance testing, and only 25 additional errors were discovered during the first year of the system's operation. Moreover, it was delivered under budget and ahead of schedule.[6] Contrast these statistics with those mentioned previously for the development of IBM's OS/360 operating system and the overall average output per programmer per day. For more details, consult the classic article on this project by F. T. Baker, listed in the suggested reading at the end of this chapter.

These results shocked the programming community. Software developers began to pay attention to what Dijkstra had been saying and writing. By the mid- to late-1970s, structured programming was being used for everything from home computers to multimillion-dollar defense projects. IBM certainly became a believer. Mills, IBM's chief programmer on the *New York Times* project, says, "I was surprised myself at how big an impact that project made. It was as if the world were waiting for something like that to happen. We [now] use it [structured programming] at IBM across the whole company. Hardly anyone can survive without it."[7]

Indeed, structured programming has been called a "revolution in programming"[8] and "one of the most important advances in computer software of the past two decades."[9] This is not to say, however, that old habits die easily. Many who learned to program prior to Dijkstra's work spurn the concept even today. Jim Horning, a computer scientist at Xerox's Palo Alto Research Center explains, "There are some people in this laboratory who read everything he [Dijkstra] writes and are extremely grateful for it, and there are others who would not be willing to have him come visit us. He tends to polarize people."[10]

Nevertheless, both academia and industry are turning more and more to the philosophy and techniques of structured programming. "Today it is safe to say that virtually all practitioners [of the art of programming] at least acknowledge the merits of the discipline [of structured programming], and most practice it exclusively."[11]

Students using this textbook would be well-advised to commit themselves to this approach, learn it well, and apply it consistently in every program they write.

CHECK YOURSELF

1. What was the "software crisis" in the mid-1960s?
2. Who was the leading figure in developing and promoting the concepts of structured programming?
3. What major programming project first employed the techniques of structured programming and shocked the data processing community with its astonishing success?
4. Who was instrumental in promoting the method of structured programming in the United States through a series of seminars?

1.2 DEFINITION OF STRUCTURED PROGRAMMING

Structured programming is often thought to be programming without the use of the GOTO statement. Indeed, structured programming does discourage the frequent and indiscriminate use of GOTO, but there is more to it than that. To fully appreciate the definition, however, one must understand the full extent of the problem addressed by structured programming.

In a word, this problem is *complexity*. Most programs that do anything significant in the real world are rather long. Of course, such software as word processors, compilers, or operating systems stagger the imagination. In fact, some computer scientists claim that these very large software systems are the most logically complex things humans have ever invented. However, even programs of only several hundred lines can get unwieldy, and it is difficult to keep all the details of the program in mind at one time. With really large software systems, it is impossible. Yet for these programs and systems to work, every minute detail must be perfectly correct and dovetail in every respect with every other detail.

Complexity is precisely the problem that structured programming addresses. Indeed, one author has defined structured programming as follows: "a method of designing computer system components and their relationships *to minimize complexity*."[12]

How does structured programming minimize complexity? It does so in three ways, which will serve as the full, working definition of structured programming in this book.

> **Structured programming** a method of writing a computer program that uses (1) top-down analysis for problem solving, (2) modularization for program structure and organization, and (3) structured code for the individual modules.

The full explanation of these three ideas is presented in succeeding chapters. However, it is possible here at least to briefly indicate how each of them reduces in its own way the complexity of the programming task.

Top-down analysis.
A program is written to tell a computer what to do. But what do you want it to do? What is the job you want it to perform for you? This "job" is more formally called the *problem*. However, before you can tell the computer what to do, you have to "solve" the problem yourself. In other words, you have to state every step necessary in order to accomplish the job. This activity on your part is called *problem solving* or *problem analysis*. For big problems, developing a solution can be very complicated. Where do you start? Top-down analysis is a method of problem solving. It tells you how to start and guides you through the entire process. The essential idea is to subdivide a large problem into several smaller tasks or parts. Top-down analysis, therefore, simplifies or reduces the complexity of the process of problem solving. Top-down analysis is the subject of chapter 3.

Modular programming.
Programs generally require many instructions for the computer. Modular programming is a method of organizing these instructions. Large programs are broken down into separate, smaller sections called *modules, subroutines*, or *subprograms*. Each module has a specific job to do and is relatively easy to write. Thus, modular programming simplifies the task of programming by making use of a highly structured organizational plan. There is, of course, a direct correlation between the subdivisions of the problem obtained through a top-down analysis and these modules: each subdivision will correspond to a module in the program. Modular structure also simplifies programming by greatly reducing the need for the GOTO statement, which, when used frequently, tends to obscure program organization and introduce errors. Modular programming is the subject of chapter 4.

Structured coding.
If programs are broken down into modules, into what are modules subdivided? Obviously, each consists of a set of instructions to the computer. But are these instructions *organized* in any special way? That is, *are they grouped and executed in any clearly definable patterns?* In structured programming they are. They are organized within various *control structures*. A control structure represents a unique pattern of

execution for a specific set of instructions. It determines the precise order in which that set of instructions is executed.

Each control structure represents a different pattern of execution, but each of these patterns in turn represents one of three basic types of execution. The component statements within a specific control structure are executed either (1) *sequentially*, (2) *conditionally*, or (3) *repetitively*.

If, then, the order in which the instructions in a module are executed is determined exclusively by the use of control structures, the module is said to be "structured," and the code is described as *structured code*. Structured code, therefore, cannot include a GOTO statement. First, a GOTO statement affects the order in which statements in a module are executed, but it does not contain other statements; therefore, it cannot "structure" anything. Second, a GOTO statement represents no definable pattern of execution; it simply jumps to some statement other than the next one in line. Using completely structured code, therefore, reduces program complexity because the program instructions are organized into discernable patterns, and the GOTO statement—which obscures organization—is eliminated entirely. Structured coding is the subject of chapters 6 and 7.

Do not be dismayed if you did not understand every detail of the preceding discussion. It was meant as a mere introduction, and each point is discussed again and more fully in the chapters that follow. It was necessary at this point, however, to have a working definition of structured programming. Remember this at least: structured programming contains three elements. They are (1) top-down analysis, (2) modular programming, and (3) structured coding by means of control structures, whatever these terms might mean.

FOR EMPHASIS

Structured programming:

1. Top-down analysis for problem solving
2. Modularization for program structure
3. Structured code for each module

One more point should be made before passing to the next section. Not all books and articles use the term "structured programming" in the same way. Indeed, there is some controversy among theoreticians as to how it should be defined. Some authors use the term to mean modular programming (discussed in chapter 4). Others restrict it simply to the use of structured code, sometimes even limiting the code to the exclusive use of the three fundamental control structures (discussed in chapter 6). In this book it will be used in the broader sense that encompasses all three of the above ideas.

CHECK YOURSELF

1. What is the main problem of computer programming addressed by structured programming?
2. What are the three main aspects of structured programming?
3. What is the fundamental idea in top-down analysis?
4. What is the fundamental idea in the modular organization of programs?
5. What is the fundamental characteristic of structured code?

1.3 ADVANTAGES OF STRUCTURED PROGRAMMING

In general, structured programming addresses and to a great extent solves the problem of complexity in computer programming. However, in view of the previous, rather lengthy discussion of some of the specific problems that have occurred in the past, it is profitable at this point to list a more detailed summary of the advantages gained by structured programming techniques. They are all a direct result of the overall concept of subdivision or modularity both in problem analysis and program structure.

1. *Programs are more easily and more quickly written.* Big programming tasks do not remain big programming tasks. They are broken down far enough that each subtask is easy to program as a separate unit.
2. *Programs have greater reliability.* Far fewer organizational and logical errors occur in the initial stages of program development and writing.
3. *Programs require less time to debug and test.* This is true, first, because fewer errors are made in writing programs. However, modularity also makes it much faster to localize and correct those errors that do occur.

> The difficulty of debugging a program increases much faster than its size. That is, if one program is twice the size of another, then it will likely not take twice as long to debug, but perhaps four times as long. Many very large programs (such as operating systems) are put into use still containing bugs that the programmers have despaired of finding, because the difficulties seem insurmountable. Sometimes projects that have consumed years of effort must be discarded because it is impossible to discover why they will not work.[13]

The principles of structured programming eliminate much of the difficulty in debugging programs, thus avoiding such disasters.

4. *Programs are easier to maintain.* As programs and systems are used, the need often arises to modify them, either by making changes or adding new features. Informal surveys indicate that writing, debugging, and testing a program consume less than half of the overall programming effort invested in the project. More than half of this overall effort goes into program maintenance.[14] With a structured program, this task becomes much easier, especially for someone other than the original programmer. There are several reasons for this: First, the program is more *readable*; that is, the logic of a well-organized, modular program is much easier to follow. Therefore, one can more quickly discover how and where to make the needed changes. Since modules are separate, independent units, an old module is simply replaced by a new module. No longer does one change require a multitude of other changes throughout the entire program. Second, adding new features or capabilities to a program becomes as easy as adding new modules to a superstructure already available.

The cumulative result of all these advantages is greater overall programmer productivity. In the early days of computer programming, it was necessary to use as many clever tricks as possible to minimize memory usage and optimize CPU time. Such coding is generally obscure and is now called *clever code*. However, program efficiency is usually inversely proportional to program readability. To get a program that used "clever code" to work right in the first place was quite difficult and time-consuming; to read or modify someone else's program was next to impossible. But in those early days, a programmer's time was cheaper than CPU time. However, with hardware advances, memory space is no longer a problem, and programmer time has become much more expensive than CPU time. The emphasis today, therefore, must be on maximizing programmer productivity and software system reliability rather than saving a few nanoseconds of CPU time. This is the purpose of structured programming.

CHECK YOURSELF

1. List the four major advantages gained by structured programming.
2. (a) What is meant by *program reliability?* (b) What is meant by *program readability?*
3. What does the term *clever code* mean?
4. Which is more costly today—CPU time or programmer time?

1.4 STRUCTURED PROGRAMMING AND PROGRAMMING LANGUAGES

Three statements can be made about structured programming and computer languages:

1. A programmer can apply the methods of structured programming in any language.
2. Some languages have been designed in such a way that one virtually has to write a modular program.
3. The ease with which one can write structured code depends to some extent on the language used.

Consider the second statement. The languages ALGOL, PL/I, Ada, and Pascal have been designed not only to facilitate but also to encourage the use of modularization in program organization. Such languages are sometimes said to be *block-structured*. Chapter 4, which deals with modular structure, discusses this term in more detail. FORTRAN, on the other hand, is unstructured. Dijkstra has made the statement that the sooner we forget that FORTRAN ever existed the better.[15] BASIC also falls into this category, and it is probably still the most popular language today, at least on microcomputers. However, even though FORTRAN and BASIC are not block-structured, it is still possible to write structured programs in these languages. Fortunately, our task in this textbook is much simpler: Pascal was designed to teach structured programming. But this brings us to the third statement.

Any program can be written by using only three fundamental control structures. This is what Bohm and Jacopini proved. On the one hand, however, some languages provide the convenience of several additional control structures. This is the case with Pascal. Although only one conditional loop control structure is necessary, Pascal provides three different types of loop structures. This makes the task of avoiding GOTO statements that much easier. On the other hand, other languages make it difficult to write structured code. For example, Applesoft BASIC does not have the ELSE option in its IF statement. Neither does it have a conditional loop structure. Therefore, the use of the GOTO statement can be minimized but not eliminated. Pascal, by comparison with other languages, tends to be rich in control structures. Not only is it possible to write code without a GOTO statement, it is actually easy to do so.

CHECK YOURSELF

1. True or false: The techniques of structured programming can be used in any computer language.
2. What is a block-structured language?
3. (a) Is Pascal a block-structured language? (b) Name two languages that are not block-structured. (c) Name two additional languages that are block-structured.
4. True or false: The more control structures directly implemented in a language, the easier it is to write structured code.

1.5 ADDITIONAL TOPICS RELATED TO STRUCTURED PROGRAMMING

There are additional important aspects to the philosophy of structured programming. They are not discussed in this book, however, because they are more closely related to systems analysis than to writing individual programs. These topics include the concept of the *"chief programmer" team* and *structured walkthroughs*. Additional information on these topics can be found in Capron and Williams (see the following list of suggested reading). *Managing the Structured Techniques* and *Structured Walkthroughs*, both by Edward Yourdon, contain extensive treatments of these topics. The first of these two books is written especially for data-processing managers.

NOTES

1. Steve Olson, "Sage of Software," *Science*, January/February 1984, p. 79.
2. Don Cassell and Martin Jackson, *Introduction to Computers and Information Processing* (Reston, Va.: Reston Publishing Co., 1980), p. 243.
3. Olson, "Sage of Software," p. 75.
4. Olson, "Sage of Software," pp. 75–76.
5. Donald D. Spencer, *Introduction to Information Processing*, 3d ed. (Columbus, Ohio: Merrill Publishing Co., 1981), p. 362.
6. H. L. Capron and Brian K. Williams, *Computers and Data Processing* (Menlo Park, Cal.: Benjamin/Cummings Publishing Co., 1982), p. 265.
7. Quoted by Olson, "Sage of Software," p. 84.
8. Daniel D. McCracken, "Revolution in Programming," *Datamation*, December 1973, p. 50.
9. Olson, "Sage of Software," p. 76.
10. Quoted by Olson, "Sage of Software," p. 76.
11. Robert T. Grauer, *Structured COBOL Programming* (Englewood Cliffs, N.J.: Prentice-Hall, 1985), p. 151.
12. Capron and Williams, *Computers and Data Processing*, p. 266.
13. Robert L. Kruse, *Data Structures and Program Design*, 2d ed. (Englewood Cliffs, N.J.: Prentice-Hall, 1987), p. 3.
14. Kruse, *Data Structures and Program Design*, p. 3.
15. Quoted by Olson, "Sage of Software," p. 76.

SUGGESTED READING

Baker, F. T. "Chief Programmer Team Management of Production Programming." *IBM Systems Journal* 11 (January 1972): 56–73.

Capron, H. L., and Williams, Brian K. *Computers and Data Processing.* Menlo Park, Cal.: Benjamin/Cummings Publishing Co., 1982. Pp. 263–66.

Dijkstra, Edsger W. "The Humble Programmer." *Communications of the ACM* 15 (October 1972): 859–66.

McCracken, Daniel D. "Revolution in Programming." *Datamation,* December 1973, pp. 50–52.

McGowan, Clement L., and Kelly, John R. *Top-down Structured Programming Techniques.* New York: Mason/Charter Publishers, 1975. Pp. 1–4.

Olson, Steve. "Sage of Software." *Science,* January/February 1984, pp. 75–84.

Spencer, Donald D. *Introduction to Information Processing.* 3d ed. Columbus, Ohio: Merrill Publishing Co., 1981. Pp. 355–63.

Yourdon, Edward. *Managing the Structured Techniques.* 2d ed. New York: Yourdon Press, 1979.

Yourdon, Edward. *Structured Walkthroughs.* 3d ed. Englewood Cliffs, N.J.: Prentice-Hall, 1985.

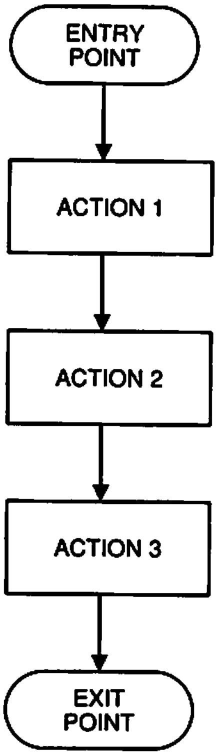

FIGURE 6.2
General SEQUENCE control structure.

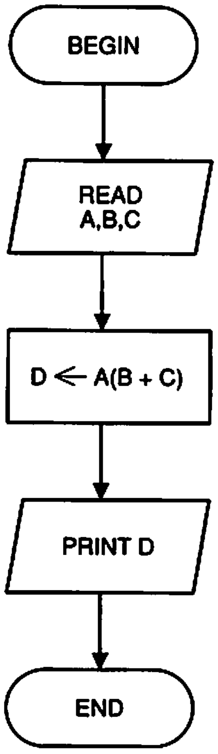

FIGURE 6.3
Flowchart of example SEQUENCE control structure.

Three observations are in order. First, virtually all Pascal control sections consist of a single SEQUENCE control structure, which must be implemented by the compound statement. In this particularly simple case, there are no other control structures within the compound statement. Normally, of course, the compound statement representing the control section of a Pascal program would consist of some combination of control structures. Second, the compound statement in Pascal is enclosed or bracketed by the reserved words BEGIN and END. These terms serve as markers and are called *delimiters*. Third, compound statements can be used anywhere in a Pascal program where the syntax requires a statement. In the following section an example is given of a compound statement used within an IF statement.

The SELECTION Control Structure

The SELECTION control structure represents *conditional execution* and is implemented in Pascal by means of the *IF statement*. The execution path described by this control structure is a *fork* that divides into two branches. The general flowchart is given in Figure 6.4.

1. A control structure has *only one entry point and only one exit point*. Implicit in this rule is the fact that a control structure and the coding that represents it must be a *block* that can be inserted or removed while the rest of the module or program remains intact. As you have learned in chapter 4, modules as a whole are also blocks in a similar sense, and the same term is used in both cases to describe this property: *plug-in compatibility*.
2. Control structures can be *nested* or *imbedded* within control structures. That is, the component statements in a structured control statement can themselves be other structured control statements as well as simple statements. Therefore, arbitrarily complex modules can be created. The general topic of nesting is discussed in chapter 7. Although, in the following examples, the "action" is usually a simple programming statement or two, this should not obscure the programming power represented by Pascal's six control structures.

CHECK YOURSELF

1. There are two ways the use of GOTO can destroy the "one entry point–one exit point" requirement for the proper use of the control structures. Explain each.
2. What are the two parts of a program to which the term "plug-in compatibility" can be applied?
3. What is meant by "nesting" control structures?

6.7 NONREPETITIVE CONTROL STRUCTURES

Each control structure available in Pascal is now described in detail. Flowcharts and Pascal examples are included. However, because loops are somewhat complicated, they are treated separately in the next section. Three items are included in the presentation of each control structure: a general or generic flowchart, a flowchart for an actual example, and the Pascal code for that example. Remember that this chapter is concerned primarily with the general nature of control structures, not with Pascal syntax. Read chapter 18 in Part II simultaneously with this chapter. It contains a full discussion of the syntax of the structured control statements used to implement the control structures in the Pascal code.

The SEQUENCE Control Structure

The SEQUENCE control structure is the easiest to understand. It represents *sequential execution* and is implemented in Pascal by means of the *compound statement*. The execution path described by this control structure is a *straight line*. The general flowchart is given in Figure 6.2. Figure 6.3 contains a specific example, which is coded in Pascal as follows:

```
PROGRAM EXAMPLE_SEQUENCE (INPUT, OUTPUT);
VAR A,B,C,D: REAL;
BEGIN
  READLN (A,B,C);
  D = A * (B + C);
  WRITELN (D)
END.
```

the *minimum* number of control structures necessary for programming? The answer was established in 1964: only three control structures are really needed. In other words, any algorithm can be programmed by using only three control structures. However, they cannot be chosen at random from the preceding list. Not surprisingly, each of the three basic types of execution must be represented: sequential, conditional, and repetitive. Moreover, the control structure representing repetitive execution must be "conditional" rather than "unconditional" (these terms are explained later in this chapter).

In 1964 Corrado Bohm and Guiseppe Jacopini presented a paper at an international colloquium held in Israel. In it they proved mathematically that any program could be written using the following three control structures: SEQUENCE, SELECTION, and DO-WHILE. The English translation of this paper was published in 1966[2]. These three control structures, therefore, are often considered the *fundamental* control structures. Some authors, in fact, define structured coding to mean coding that uses only these three control structures. That, however, is not the definition of structured coding given in this book; a broader definition is given in the next chapter.

There are two consequences of this proof. First, if a high-level language implements at least these three control structures, it is not necessary for it to include a GOTO statement at all. Of course, other control structures can be implemented for convenience. Why, then, are there still GOTO statements? Because old habits die hard among old programmers. Languages include a GOTO statement in deference to programmers who still think in terms of GOTO. However, more and more companies, programming departments, and schools are insisting that programmers and students of programming avoid the use of the GOTO statement and write structured programs.

Second, structured programming and structured coding demand that programming *practice* reflect what is *theoretically* possible. That is, if it is a proven fact that any program can be written with no GOTO statements, then no GOTO statements should be used in writing programs. The exclusive use of the control structures and the complete elimination of the GOTO statement should and eventually will become standard programming practice.

CHECK YOURSELF

1. How many control structures does Pascal implement?
2. What are the generic names used in this book for the control structures available in Pascal?
3. What are the names of the Pascal statements that implement the control structures available in Pascal?
4. Which Pascal statements implement control structures that represent (a) sequential execution? (b) Conditional execution? (c) Repetitive execution?
5. (a) What are generally considered to be the fundamental control structures? (b) Why are they considered fundamental?

6.6 CHARACTERISTICS OF CONTROL STRUCTURES

Before discussing each control structure in detail, two very important characteristics of all control structures should be observed. Each of these properties plays an important role in chapter 7.

6.5 CONTROL STRUCTURE NAMES

The purpose of this section is to name the various control structures available in Pascal and the structured control statements by which they are implemented. After this section, the remainder of this chapter describes each of the control structures in detail.

As you learned in the previous section, the control structures are language independent. Thus, each control structure has a *generic* name that, in general, is also language independent. Therefore, if a certain control structure is implemented in a given language, the *statement* name in that language may not be the same as the generic name. This is no real problem, though, since they are usually quite close, and sometimes even the same. For example, the FOR-DO control structure is the DO statement in FORTRAN and the FOR statement in BASIC and Pascal. Also, the DO-WHILE control structure is implemented in Pascal by means of the WHILE statement.

Now, then, what are the various control structures? Generic names for control structure have not been entirely standardized, but the six structures that have been implemented in Pascal are usually named, as follows: (1) SEQUENCE, (2) SELECTION (also called IF-THEN-ELSE; the IF-THEN structure is a special case of the SELECTION control structure), (3) DO-WHILE, (4) DO-UNTIL (also called PERFORM-UNTIL or REPEAT-UNTIL), (5) FOR-DO (also called ITERATIVE-DO), and (6) SELECT-CASE (also called ON-A-PERFORM). Other structures exist, but they are not directly implemented by any Pascal statement[1].

If six control structures are implemented in Pascal, why does Pascal have seven structured control statements? The answer is that the WITH statement is somewhat of an anomaly. It is a structured control statement, but like the more straightforward compound statement, it also represents the SEQUENCE control structure. The only difference between the two statements is that the WITH statement allows its component statements to drop the record variable name when accessing fields within that record. It is effectively a specialized form of the compound statement and is discussed further in chapter 25.

The Pascal statement names for the previous control structures are as follows:

Generic Name	*Pascal Statement*
SEQUENCE	Compound statement
SELECTION	IF statement
FOR-DO	FOR statement
DO-WHILE	WHILE statement
DO-UNTIL	REPEAT statement
SELECT-CASE	CASE statement

According to the definition, each of these control structures represents one of the three basic types of execution. Therefore, they can be grouped as follows:

Sequential Flow	*Conditional Flow*	*Repetitive Flow*
SEQUENCE	SELECTION	FOR-DO
	SELECT-CASE	DO-WHILE
		DO-UNTIL

The more control structures available in a given programming language, the more powerful that language is for writing programs. Pascal is one of the more powerful languages in this regard. However, there is an interesting theoretical question: What is

Control structure (a) a specific pattern for organizing program logic, that is, a pattern for directing the flow of control in the execution of a program; (b) each of the six patterns available in Pascal is characterized either by sequential flow, conditional flow, or repetitive flow; (c) a control structure is implemented in a high-level language by means of a structured control statement.

There are several points to observe in this definition of control structure:

1. A control structure is a distinguishable and clearly definable pattern for the flow of control in a module.
2. There are many control structures or patterns for the flow of control to follow.
3. However, all flow patterns can be characterized as either sequential, conditional, or repetitive. Each of the many different patterns is simply a variation on one of these themes.
4. Control structures are *language independent*. They are implemented in a specific high-level language by the structured control statements provided in that language. Therefore, each of the structured control statements in a specific language represents a different control structure.
5. Structured control statements are *language dependent*. They are, of course, actual statements in a specific high-level language. They are "structured" in the sense that they consist of other statements as building blocks, and they are "control" statements in the sense that they direct the execution of their component statements: the component statements are executed either sequentially, conditionally, or repetitively. Therefore, they represent the *language-specific* implementations of the *language-independent* control structures.
6. Six control structures or patterns have been implemented in Pascal by seven structured control statements. Pascal has two structured control statements in which their component statements are executed sequentially, two in which their component statements are executed conditionally, and three in which their component statements are executed repetitively. These are all named in the next section.

To summarize: To use control structures, as implemented by structured control statements, in writing a module is to introduce organization into the module. These statements organize the component statements placed in them and therefore the flow of control in the module. One word of caution, however, is in order before concluding this discussion: the organization represented by a structured control statement is destroyed and it ceases to represent a control structure if a GOTO statement is included as one of its component statements. This point is discussed in more detail in the next chapter. There you will see that the concept of organization is quite important to an understanding of the last aspect of structured programming—using structured code.

CHECK YOURSELF

1. What constitutes a well-organized program?
2. What constitutes a well-organized module?
3. What are the three basic patterns of execution?
4. In what two ways does a structured statement organize a module?
5. Why is it impossible for a structured noncontrol statement to exist?
6. What is a control structure?
7. Explain the relationship between a control structure and a structured control statement.

First, what makes a *program* well-organized? *A program is well-organized when it is modularized,* as chapter 4 pointed out. Modular structure for programs was the second major aspect of structured programming.

Second, what in turn makes a *module* well-organized? *A module is well-organized when it consists exclusively of clearly distinguishable patterns for the flow of control among the statements that form the module.* This is the only reasonable way to define organization among the statements in a module.

Third, how many distinguishable or definable patterns are there for the flow of control in a module? There are many such patterns, six of which are available in Pascal. However, each individual pattern represents one of only three characteristic types of flow possible within a program: the flow can be (1) *sequential,* (2) *conditional,* or (3) *repetitive.*

When the flow of control is sequential, a single series or sequence of statements is executed once and in order. When the flow is conditional, some condition is tested in order to determine which of two or more statements or series of statements is executed. Finally, when the flow of control is repetitive, one or more statements are executed repeatedly; the decision whether to perform the loop again is made on the basis of the truth-value of some condition. In Pascal, there is one example of sequential flow, two variations of conditional flow, and three variations of repetitive flow, making a total of six patterns. These patterns, called *control structures,* are defined below and then described in detail later in this chapter.

The Nature of Structured Statements

According to the definition given earlier in this chapter, a structured statement is constructed by using other statements as building blocks. In addition, a structured statement determines the manner in which the component statements are executed. A structured statement, therefore, organizes two things. First, it organizes the statements of which it is composed. This is obvious: since each structured statement consists of other statements, it represents a particular way to put those statements together. Second, since it determines the manner in which they are executed (sequentially, conditionally, or repetitively), a structured statement also organizes the flow of control in the program.

Two conclusions can now be drawn with regard to structured statements:

1. A structured statement must also be a control statement; there can be no such thing as a structured non-control statement.

The reason is quite simple. Since a structured statement determines the manner in which its component statements are executed, it necessarily affects the flow of control through those statements and is therefore a control statement.

2. A structured statement is the way to implement a control structure in a high-level language.

This follows from the fact that a structured statement organizes the flow of control into a recognizable pattern.

Definition of a Control Structure

With this background on what constitutes a well-organized module and on the nature of structured control statements, you are ready for the formal definition of a control structure:

chart loops should be translated idiomatically rather than literally. In other words, do not construct loops with IF statements and GOTO statements—use the statements that have been designed for automatic loop control and with structured programming in mind.

Now consider pseudocode. Translating pseudocode into Pascal presents no problem whatever. Pseudocode was designed with structured control statements in mind. The previous program would be written in pseudocode as follows:

```
BEGIN main control section
    Set K = 1
    DOWHILE K ≤ 5
        Read A
        Print (A * A)
        Set K = K + 1
    ENDDO
END main control section
```

Despite the problem of translating flowcharts into Pascal code, they do provide a very graphic demonstration of the control structures studied in this chapter. Therefore, in the explanation of control structures in general and Pascal's structured control statments in particular, flowcharts are routinely used.

CHECK YOURSELF

1. What is a program loop?
2. How is a loop flowcharted?
3. What are the three structured control statements available in Pascal for loop control?
4. What do the terms "literal" and "idiomatic" mean in relation to the process of translation?
5. What two high-level language instructions are necessary in order to give a "literal" translation of a flowchart loop?
6. What feature of the WHILE loop (and other loop-control structures) makes it possible to eliminate the GOTO statement?

6.4 DEFINITION OF A CONTROL STRUCTURE

We have seen that the building blocks of a module and, therefore, of a program are the individual statements available in the high-level language being used. What are the different ways these building blocks can be put together to form modules? There are several aspects to this question, and it is best to treat them separately.

Well-organized Modules

Is it possible to distinguish a "well-organized" module from an "unorganized" module? The answer is yes. The subject of how modules are organized can be presented by asking three questions.

```
PROGRAM SQUARES (OUTPUT);
LABEL BEGINLOOP;
VAR
  K: INTEGER;
  A: REAL;
BEGIN
  K := 1;
  BEGINLOOP:
  IF K <= 5 THEN
    BEGIN
      READLN (A);
      WRITELN (A * A);
      K := K + 1;
      GOTO BEGINLOOP
    END
END.
```

The preceding Pascal translation does not make use of any of Pascal's structured control statements that were designed for looping. However, these statements were made a part of Pascal to perform this very activity. Therefore, an idiomatic translation would make use of one of them. For example:

```
PROGRAM SQUARES (OUTPUT);
VAR
  K: INTEGER;
  A: REAL;
BEGIN
  K := 1;
  WHILE K <= 5 DO
    BEGIN
      READLN (A);
      WRITELN (A * A);
      K := K + 1
    END
END.
```

The WHILE statement was chosen to perform this loop. Notice that there is no explicit IF statement or GOTO statement in the program. Checking the condition "K ≤ 5" and branching back to the beginning of the loop are both performed automatically. The WHILE loop translation is considered better and more "idiomatic" because the WHILE statement was designed to do the very thing represented by the flowchart. The GOTO statement is not needed. Here is our first illustration of an extremely important point in this and the next chapter: *an adequate supply of control structures renders the GOTO statement unnecessary.*

For another illustration, you should study again Example 5.1 in chapter 5. The flowchart in Figure 5.2 also has two possible Pascal translations: a literal one using IF and GOTO statements and an idiomatic translation using the REPEAT statement, another of Pascal's structured control statements designed for looping. Looking back now, you will see that the flowchart was actually translated idiomatically.

To appreciate fully the contrast being drawn here, you must understand a bit of history. In the early days of computer programming, there were no WHILE statements, REPEAT statements, or even FOR statements. All program loops had to be constructed by means of IF statements and GOTO statements. It was during that time that flowcharts were developed. This explains why flowcharts are so convenient for displaying "GOTO" logic and also why flowcharting (to date, at least) has no special symbol for loops as such. Now, however, with the advent of these loop control statements, flow-