Chapter 1

Introductory Concepts

This book offers instruction in computer programming using a popular, structured programming language called C. We will learn how programs can be written in C. In addition, we will see how problems that are initially described in very general terms can be analyzed, outlined and finally transformed into well-organized C programs. These concepts are demonstrated in detail by the many sample problems that are included in the text.

1.1 INTRODUCTION TO COMPUTERS

Today's computers come in many different forms. They range from massive, multipurpose *mainframes* and *supercomputers* to desktop-size *personal computers*. Between these extremes is a vast middle ground of *minicomputers* and *workstations*. Large minicomputers approach mainframes in computing power, whereas workstations are powerful personal computers.

Mainframes and large minicomputers are used by many businesses, universities, hospitals and government agencies to carry out sophisticated scientific and business calculations. These computers are expensive (large computers can cost millions of dollars) and may require a sizeable staff of supporting personnel and a special, carefully controlled environment.

Personal computers, on the other hand, are small and inexpensive. In fact, portable, battery-powered "laptop" computers weighing less than 5 or 6 pounds are now widely used by many students and traveling professionals. Personal computers are used extensively in most schools and businesses and they are rapidly becoming common household items. Most students use personal computers when learning to program with C.

Figure 1.1 shows a student using a laptop computer.

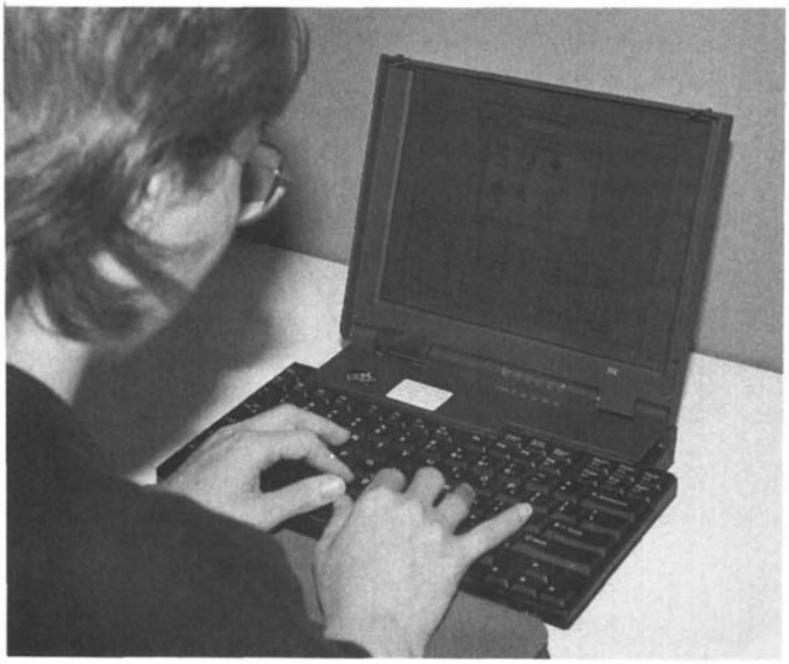


Fig. 1.1

Despite their small size and low cost, modern personal computers approach minicomputers in computing power. They are now used for many applications that formerly required larger, more expensive computers. Moreover, their performance continues to improve dramatically as their cost continues to drop. The design of a personal computer permits a high level of interaction between the user and the computer. Most applications (e.g., word processors, graphics programs, spreadsheets and database management programs) are specifically designed to take advantage of this feature, thus providing the skilled user with a wide variety of creative tools to write, draw or carry out numerical computations. Applications involving high-resolution graphics are particularly common.

Many organizations connect personal computers to larger computers or to other personal computers, thus permitting their use either as stand-alone devices or as terminals within a computer *network*. Connections over telephone lines are also common. When viewed in this context, we see that personal computers often *complement*, rather than *replace*, the use of larger computers.

1.2 COMPUTER CHARACTERISTICS

All digital computers, regardless of their size, are basically electronic devices that can transmit, store, and manipulate *information* (i.e., *data*). Several different types of data can be processed by a computer. These include *numeric data*, *character data* (names, addresses, etc.), *graphic data* (charts, drawings, photographs, etc.), and *sound* (music, speech patterns, etc.). The two most common types, from the standpoint of a beginning programmer, are numeric data and character data. Scientific and technical applications are concerned primarily with numeric data, whereas business applications usually require processing of both numeric and character data.

To process a particular set of data, the computer must be given an appropriate set of instructions called a *program*. These instructions are entered into the computer and then stored in a portion of the computer's *memory*.

A stored program can be *executed* at any time. This causes the following things to happen.

1. A set of information, called the *input data*, will be entered into the computer (from the keyboard, a floppy disk, etc.) and stored in a portion of the computer's memory.
2. The input data will be processed to produce certain desired results, known as the *output data*.
3. The output data, and perhaps some of the input data, will be printed onto a sheet of paper or displayed on a *monitor* (a television receiver specially designed to display computer output).

This three-step procedure can be repeated many times if desired, thus causing a large quantity of data to be processed in rapid sequence. It should be understood, however, that each of these steps, particularly steps 2 and 3, can be lengthy and complicated.

EXAMPLE 1.1 A computer has been programmed to calculate the area of a circle using the formula $a = \pi r^2$, given a numeric value for the radius r as input data. The following steps are required.

1. Read the numeric value for the radius of the circle.
2. Calculate the value of the area using the above formula. This value will be stored, along with the input data, in the computer's memory.
3. Print (display) the values of the radius and the corresponding area.
4. Stop.

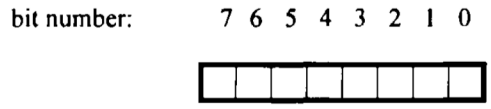
Each of these steps will require one or more instructions in a computer program.

The foregoing discussion illustrates two important characteristics of a digital computer: *memory* and *capability to be programmed*. A third important characteristic is its *speed and reliability*. We will say more about memory, speed and reliability in the next few paragraphs. Programmability will be discussed at length throughout the remainder of this book.

Memory

Every piece of information stored within the computer's memory is encoded as some unique combination of zeros and ones. These zeros and ones are called *bits* (*bi*nary digi*ts*). Each bit is represented by an electronic device that is, in some sense, either "off" (zero) or "on" (one).

Small computers have memories that are organized into 8-bit multiples called *bytes*, as illustrated in Fig. 1.2. Notice that the individual bits are numbered, beginning with 0 (for the rightmost bit) and extending to 7 (the leftmost bit). Normally, a single character (e.g., a letter, a single digit or a punctuation symbol) will occupy one byte of memory. An instruction may occupy 1, 2 or 3 bytes. A single numeric quantity may occupy 1 to 8 bytes, depending on its *precision* (i.e., the number of significant figures) and its *type* (integer, floating-point, etc.).



One byte

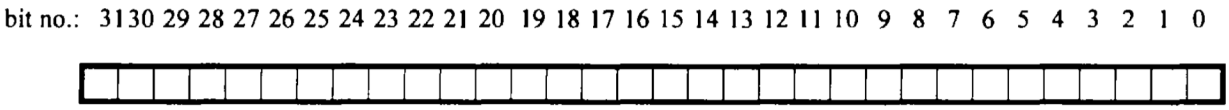
Fig. 1.2

The size of a computer's memory is usually expressed as some multiple of $2^{10} = 1024$ bytes. This is referred to as 1K. Modern small computers have memories whose sizes typically range from 4 to 16 megabytes, where 1 megabyte (1M) is equivalent to $2^{10} \times 2^{10}$ bytes, or 2^{10} K = 1024K bytes.

EXAMPLE 1.2 The memory of a personal computer has a capacity of 16M bytes. Thus, as many as $16 \times 1024 \times 1024 = 16{,}777{,}216$ characters and/or instructions can be stored in the computer's memory. If the entire memory is used to represent character data (which is actually quite unlikely), then over 200,000 names and addresses can be stored within the computer at any one time, assuming 80 characters for each name and address.

If the memory is used to represent numeric data rather than names and addresses, then more than 4 million individual numbers can be stored at any one time, assuming each numeric quantity requires 4 bytes of memory.

Large computers have memories that are organized into *words* rather than bytes. Each word will consist of a relatively large number of bits—typically 32 or 36. The bit-wise organization of a 32-bit word is illustrated in Fig. 1.3. Notice that the bits are numbered, beginning with 0 (for the rightmost bit) and extending to 31 (the leftmost bit).

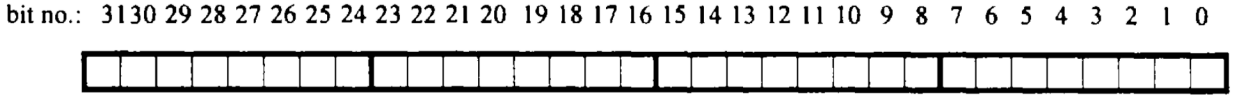


One 32-bit word

Fig. 1.3

Figure 1.4 shows the same 32-bit word organized into 4 consecutive bytes. The bytes are numbered in the same manner as the individual bits, ranging from 0 (for the rightmost byte) to 3 (the leftmost byte).

The use of a 32- or a 36-bit word permits one numeric quantity, or a small *group* of characters (typically 4 or 5), to be represented within a single word of memory. Large computers commonly have several million words (i.e., several megawords) of memory.



One 4-byte (32-bit) word

Fig. 1.4

EXAMPLE 1.3 The memory of a large computer has a capacity of 32M (32,768K) words, which is equivalent to 32 × 1024 × 1024= 33,554,432 words. If the entire memory is used to represent numeric data (which is unlikely), then more than 33 million numbers can be stored within the computer at any one time, assuming each numeric quantity requires one word of memory.

If the memory is used to represent characters rather than numeric data, then about 130 million characters can be stored at any one time, based upon 4 characters per word. This is enough memory to store the contents of several large books.

Most computers also employ *auxiliary storage devices* (e.g., magnetic tapes, disks, optical memory devices) in addition to their primary memories. These devices can store more than 1 gigabyte (1G = 1024M bytes) of information. Moreover, they allow information to be recorded permanently, since they can often be physically disconnected from the computer and stored when not in use. However, the access time (i.e., the time required to store or retrieve information) is considerably greater for these auxiliary devices than for the computer's primary memory.

Speed and Reliability

Because of its extremely high speed, a computer can carry out calculations within minutes that might require many days, perhaps even months or years, if carried out by hand. For example, the end-of-semester grades for all students in a large university can typically be processed in just a few minutes on a large computer.

The time required to carry out simple computational tasks, such as adding two numbers, is usually expressed in terms of *microseconds* (1 μsec = 10^{-6} sec) or *nanoseconds* (1 nsec = 10^{-3} μsec = 10^{-9} sec). Thus, if a computer can add two numbers in 10 nanoseconds (typical of a modern medium-speed computer), 100 million (10^8) additions will be carried out in one second.

This very high speed is accompanied by an equally high level of reliability. Thus, computers never make mistakes of their own accord. Highly publicized "computer errors," such as a person's receiving a tax refund of several million dollars, are the result of programming errors or data entry errors rather than errors caused by the computer itself.

1.3 MODES OF OPERATION

There are two different ways that a large computer can be shared by many different users. These are the *batch mode* and the *interactive mode*. Each has its own advantages for certain types of problems.

Batch Processing

In *batch processing*, a number of jobs are entered into the computer, stored internally, and then processed sequentially. (A *job* refers to a computer program and its associated sets of input data.) After the job is processed, the output, along with a listing of the computer program, is printed on multiple sheets of paper by a high-speed printer. Typically, the user will pick up the printed output at some convenient time, after the job has been processed.

In *classical batch processing* (which is now obsolete), the program and the data were recorded on *punched cards*. This information was read into the computer by means of a mechanical card reader and then processed. In the early days of computing, all jobs were processed in this manner.

Modern batch processing is generally tied into a timesharing system (see below). Thus, the program and the data are typed into the computer via a *timesharing terminal* or a personal computer acting as a terminal. The information is then stored within the computer's memory and processed in its proper sequence. This form of batch processing is preferable to classical batch processing, since it eliminates the need for punched cards and allows the input information (program and data) to be edited while it is being entered.

Large quantities of information (both programs and data) can be transmitted into and out of the computer very quickly in batch processing. Furthermore, the user need not be present while the job is being processed. Therefore, this mode of operation is well-suited to jobs that require large amounts of computer time or are physically lengthy. On the other hand, the total time required for a job to be processed in this manner may vary from several minutes to several hours, even though the job may require only a second or two of actual computer time. (Each job must wait its turn before it can be read, processed, and the results displayed.) Thus, batch processing is undesirable when processing small, simple jobs that must be returned as quickly as possible (as, for example, when learning computer programming).

Timesharing

Timesharing allows many different users to use a single computer simultaneously. The host computer may be a mainframe, a minicomputer or a large desktop computer. The various users communicate with the computer through their own individual terminals. In a modern timesharing network, personal computers are often used as timesharing terminals. Since the host computer operates much faster than a human sitting at a terminal, the host computer can support many terminals at the same time. Thus, each user will be unaware of the presence of any other users and will seem to have the host computer at his or her own disposal.

An individual timesharing terminal may be wired directly to the host computer, or it may be connected to the computer over telephone lines, a microwave circuit, or even an earth satellite. Thus, the terminal can be located far—perhaps hundreds of miles—from its host computer. Systems in which personal computers are connected to large mainframes over telephone lines are particularly common. Such systems make use of *modems* (i.e., *mo*dulator/*dem*odulator devices) to convert the digitized computer signals into analog telephone signals and vice versa. Through such an arrangement a person working at home, on his or her own personal computer, can easily access a remote computer at school or at the office.

Timesharing is best suited for processing relatively simple jobs that do not require extensive data transmission or large amounts of computer time. Many applications that arise in schools and commercial offices have these characteristics. Such applications can be processed quickly, easily, and at minimum expense using timesharing.

EXAMPLE 1.4 A major university has a computer timesharing capability consisting of 200 hard-wired timesharing terminals and 80 additional telephone connections. The timesharing terminals are located at various places around the campus and are wired directly to a large mainframe computer. Each terminal is able to transmit information to or from the central computer at a maximum speed of 960 characters per second.

The telephone connections allow students who are not on campus to connect their personal computers to the central computer. Each personal computer can transmit data to or from the central computer at a maximum speed of 240 characters per second. Thus, all 280 terminals and personal computers can interact with the central computer at the same time, though each student will be unaware that others are simultaneously sharing the computer.

Interactive Computing

Interactive computing is a type of computing environment that originated with commercial timesharing systems and has been refined by the widespread use of personal computers. In an interactive computing environment, the user and the computer interact with each other during the computational session. Thus, the user may periodically be asked to provide certain information that will determine what subsequent actions are to be taken by the computer and vice versa.

EXAMPLE 1.5 A student wishes to use a personal computer to calculate the radius of a circle whose area has a value of 100. A program is available that will calculate the area of a circle, given the radius. (Note that this is just the opposite of what the student wishes to do.) This program isn't exactly what is needed, but it does allow the student to obtain an answer by trial and error. The procedure will be to guess a value for the radius and then calculate a corresponding area. This trial-and-error procedure continues until the student has found a value for the radius that yields an area sufficiently close to 100.

Once the program execution begins, the message

```
Radius = ?
```

is displayed. The student then enters a value for the radius. Let us assume that the student enters a value of 5 for the radius. The computer will respond by displaying

```
Area = 78.5398

Do you wish to repeat the calculation?
```

The student then types either `yes` or `no`. If the student types `yes`, the message

```
Radius = ?
```

again appears, and the entire procedure is repeated. If the student types `no`, the message

```
Goodbye
```

is displayed and the computation is terminated.

Shown below is a printed copy of the information displayed during a typical interactive session using the program described above. In this session, an approximate value of $r = 5.6$ was determined after only three calculations. The information typed by the student is underlined.

```
Radius = ? 5
Area = 78.5398

Do you wish to repeat the calculation? yes

Radius = ? 6
Area = 113.097

Do you wish to repeat the calculation? yes

Radius = ? 5.6
Area = 98.5204

Do you wish to repeat the calculation? no

Goodbye
```

Notice the manner in which the student and the computer appear to be conversing with one another. Also, note that the student waits until he or she sees the calculated value of the area before deciding whether or not to carry out another calculation. If another calculation is initiated, the new value for the radius supplied by the student will depend on the previously calculated results.

Programs designed for interactive computing environments are sometimes said to be *conversational* in nature. Computerized games are excellent examples of such interactive applications. This includes fast-action, graphical arcade games, even though the user's responses may be reflexive rather than numeric or verbal.

1.4 TYPES OF PROGRAMMING LANGUAGES

There are many different languages can be used to program a computer. The most basic of these is *machine language*—a collection of very detailed, cryptic instructions that control the computer's internal circuitry. This is the natural dialect of the computer. Very few computer programs are actually written in machine language, however, for two significant reasons: First, because machine language is very cumbersome to work with and second, because every different type of computer has its own unique instruction set. Thus, a machine-language program written for one type of computer cannot be run on another type of computer without significant alterations.

Usually, a computer program will be written in some *high-level* language, whose instruction set is more compatible with human languages and human thought processes. Most of these are *general-purpose* languages such as C. (Some other popular general-purpose languages are Pascal, Fortran and BASIC.) There are also various *special-purpose* languages that are specifically designed for some particular type of application. Some common examples are CSMP and SIMAN, which are special-purpose *simulation* languages, and LISP, a *list-processing* language that is widely used for artificial intelligence applications.

As a rule, a single instruction in a high-level language will be equivalent to several instructions in machine language. This greatly simplifies the task of writing complete, correct programs. Furthermore, the rules for programming in a particular high-level language are much the same for all computers, so that a program written for one computer can generally be run on many different computers with little or no alteration. Thus, we see that a high-level language offers three significant advantages over machine language: *simplicity, uniformity* and *portability* (i.e., machine independence).

A program that is written in a high-level language must, however, be translated into machine language before it can be executed. This is known as *compilation* or *interpretation*, depending on how it is carried out. (Compilers translate the entire program into machine language before executing any of the instructions. Interpreters, on the other hand, proceed through a program by translating and then executing single instructions or small groups of instructions.) In either case, the translation is carried out automatically within the computer. In fact, inexperienced programmers may not even be aware that this process is taking place, since they typically see only their original high-level program, the input data, and the calculated results. Most implementations of C operate as compilers.

A compiler or interpreter is itself a computer program. It accepts a program written in a high-level language (e.g., C) as input, and generates a corresponding machine-language program as output. The original high-level program is called the *source* program, and the resulting machine-language program is called the *object* program. Every computer must have its own compiler or interpreter for a particular high-level language.

It is generally more convenient to develop a new program using an interpreter rather than a compiler. Once an error-free program has been developed, however, a compiled version will normally execute much faster than an interpreted version. The reasons for this are beyond the scope of our present discussion.

1.5 INTRODUCTION TO C

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English *keywords* such as `if`, `else`, `for`, `do` and `while`. In this respect C resembles other high-level structured programming languages such as Pascal and Fortran. C also contains certain additional features, however, that allow it to be used at a lower level, thus bridging the gap between machine language and the more conventional high-level languages. This flexibility allows C to be used for *systems programming* (e.g., for writing operating systems) as well as for *applications programming* (e.g., for writing a program to solve a complicated system of mathematical equations, or for writing a program to bill customers).

C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language. It has a relatively small instruction set, though actual implementations include extensive *library functions* which enhance the basic instructions. Furthermore, the language encourages users to write additional library functions of their own. Thus the features and capabilities of the language can easily be extended by the user.

C compilers are commonly available for computers of all sizes, and C interpreters are becoming increasingly common. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages. The interpreters are less efficient, though they are easier to use when developing a new program. Many programmers begin with an interpreter, and then switch to a compiler once the program has been debugged (i.e., once all of the programming errors have been removed).

Another important characteristic of C is that its programs are highly portable, even more so than with other high-level languages. The reason for this is that C relegates most computer-dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for the particular characteristics of the host computer. These library functions are relatively standardized, however, and each individual library function is generally accessed in the same manner from one version of C to another. Therefore, most C programs can be processed on many different computers with little or no alteration.

History of C

C was originally developed in the 1970s by Dennis Ritchie at Bell Telephone Laboratories, Inc. (now a part of AT&T). It is an outgrowth of two earlier languages, called BCPL and B, which were also developed at Bell Laboratories. C was largely confined to use within Bell Laboratories until 1978, when Brian Kernighan and Ritchie published a definitive description of the language.* The Kernighan and Ritchie description is commonly referred to as "K&R C."

Following the publication of the K&R description, computer professionals, impressed with C's many desirable features, began to promote the use of the language. By the mid 1980s, the popularity of C had become widespread. Numerous C compilers and interpreters had been written for computers of all sizes, and many commercial application programs had been developed. Moreover, many commercial software products that were originally written in other languages were rewritten in C in order to take advantage of its efficiency and its portability.

Early commercial implementations of C differed somewhat from Kernighan and Ritchie's original definition, resulting in minor incompatibilities between different implementations of the language. These differences diminished the portability that the language attempted to provide. Consequently, the American National Standards Institute** (ANSI committee X3J11) has developed a standardized definition of the C language. Virtually all commercial C compilers and interpreters now adhere to the ANSI standard. Many also provide additional features of their own.

In the early 1980s, another high-level programming language, called C++, was developed by Bjarne Stroustrup*** at the Bell Laboratories. C++ is built upon C, and hence all standard C features are available within C++. However, C++ is not merely an extension of C. Rather, it incorporates several new fundamental concepts that form a basis for *object-oriented programming*—a new programming paradigm that is of interest to professional programmers. We will not describe C++ in this book, except to mention that a knowledge of C is an excellent starting point for learning C++.

This book describes the features of C that are included in the ANSI standard and are supported by commercial C compilers and interpreters. The reader who has mastered this material should have no difficulty in customizing a C program to any particular implementation of the language.

Structure of a C Program

Every C program consists of one or more modules called *functions*. One of the functions must be called `main`. The program will always begin by executing the `main` function, which may access other functions. Any other function definitions must be defined separately, either ahead of or after `main` (more about this later, in Chaps. 7 and 8).

---

* Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

** ANSI Standard X3.159-1989. American National Standards Institute, 1430 Broadway, New York, NY, 10018. (See also Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2d ed., Prentice-Hall, 1988.)

*** Stroustrup, Bjarne, *The C++ Programming Language*, 2d ed., Addison-Wesley, 1991.

Each function must contain:

1. A function *heading*, which consists of the function name, followed by an optional list of *arguments*, enclosed in parentheses.
2. A list of argument *declarations*, if arguments are included in the heading.
3. A *compound statement*, which comprises the remainder of the function.

The arguments are symbols that represent information being passed between the function and other parts of the program. (Arguments are also referred to as *parameters*.)

Each compound statement is enclosed within a pair of braces, i.e., { }. The braces may contain one or more elementary statements (called *expression statements*) and other compound statements. Thus compound statements may be nested, one within another. Each expression statement must end with a semicolon (;).

Comments (remarks) may appear anywhere within a program, as long as they are placed within the delimiters /* and */ (e.g., /* this is a comment */). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

These program components will be discussed in much greater detail later in this book. For now, the reader should be concerned only with an overview of the basic features that characterize most C programs.

EXAMPLE 1.6 Area of a Circle Here is an elementary C program that reads in the radius of a circle, calculates its area and then writes the calculated result.

```
/* program to calculate the area of a circle */        /* TITLE (COMMENT) */
#include <stdio.h>                                     /* LIBRARY FILE ACCESS */
main()                                                 /* FUNCTION HEADING */
{
    float radius, area;                          /* VARIABLE DECLARATIONS */
    printf("Radius = ? ");                       /* OUTPUT STATEMENT (PROMPT) */
    scanf("%f", &radius);                        /* INPUT STATEMENT */
    area = 3.14159 * radius * radius;            /* ASSIGNMENT STATEMENT */
    printf("Area = %f", area);                   /* OUTPUT STATEMENT */
}
```

The comments at the end of each line have been added in order to emphasize the overall program organization. Normally a C program will not look like this. Rather, it might appear as shown below.

```
/* program to calculate the area of a circle */
#include <stdio.h>
main()
{
    float radius, area;
    printf("Radius = ? ");
    scanf("%f", &radius);
    area = 3.14159 * radius * radius;
    printf("Area = %f", area);
}
```

The following features should be pointed out in this last program.

1. The program is typed in lowercase. Either upper- or lowercase can be used, though it is customary to type ordinary instructions in lowercase. Most comments are also typed in lowercase, though comments are sometimes typed in uppercase for emphasis, or to distinguish certain comments from the instructions.

(Uppercase and lowercase characters are not equivalent in C. Later in this book we will see some special situations that are characteristically typed in uppercase.)

2. The first line is a comment that identifies the purpose of the program.
3. The second line contains a reference to a special file (called `stdio.h`) which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler.
4. The third line is a heading for the function `main`. The empty parentheses following the name of the function indicate that this function does not include any arguments.
5. The remaining five lines of the program are indented and enclosed within a pair of braces. These five lines comprise the compound statement within `main`.
6. The first indented line is a *variable declaration*. It establishes the symbolic names `radius` and `area` as *floating-point variables* (more about this in the next chapter).
7. The remaining four indented lines are expression statements. The second indented line (`printf`) generates a request for information (namely, a value for the radius). This value is entered into the computer via the third indented line (`scanf`).
8. The fourth indented line is a particular type of expression statement called an *assignment statement*. This statement causes the area to be calculated from the given value of the radius. Within this statement the asterisks (`*`) represent multiplication signs.
9. The last indented line (`printf`) causes the calculated value for the area to be displayed. The numerical value will be preceded by a brief label.
10. Notice that each expression statement within the compound statement ends with a semicolon. This is required of all expression statements.
11. Finally, notice the liberal use of spacing and indentation, creating *whitespace* within the program. The blank lines separate different parts of the program into logically identifiable components, and the indentation indicates subordinate relationships among the various instructions. These features are not grammatically essential, but their presence is strongly encouraged as a matter of good programming practice.

Execution of the program results in an interactive dialog such as that shown below. The user's response is underlined, for clarity.

```
Radius = ? 3
Area = 28.274309
```

1.6 SOME SIMPLE C PROGRAMS

In this section we present several C programs that illustrate some commonly used features of the language. All of the programs are extensions of Example 1.6; that is, each program calculates the area of a circle, or the areas of several circles. Each program illustrates a somewhat different approach to this problem.

The reader should not attempt to understand the syntactic details of these examples, though experienced programmers will recognize features similar to those found in other programming languages. Beginners should focus their attention only on the overall program logic. The details will be provided later in this book.

EXAMPLE 1.7 Area of a Circle Here is a variation of the program given in Example 1.6 for calculating the area of a circle.

```
/* program to calculate the area of a circle */

#include <stdio.h>

#define PI 3.14159

float process(float radius);          /* function prototype */
```

```
main()
{
   float radius, area;                    /* variable declaration */
   printf("Radius = ? ");
   scanf("%f", &radius);
   area = process(radius);
   printf("Area = %f", area);
}

float process(float r)                    /* function definition */
{
   float a;                               /* local variable declaration */
   a = PI * r * r;
   return(a);
}
```

This version utilizes a separate programmer-defined function, called `process`, to carry out the actual calculations (i.e., to process the data). Within this function, `r` is an argument (also called a *parameter*) that represents the value of the radius supplied to `process` from `main`, and `a` is the calculated result that is returned to `main`. A reference to the function appears in `main`, within the statement

```
area = process(radius);
```

The `main` function is preceded by a *function declaration*, which indicates that `process` accepts a floating-point argument and returns a floating-point value. The use of functions will be discussed in detail in Chap. 7.

This program also contains a *symbolic constant*, `PI`, that represents the numerical value 3.14159. This is a form of shorthand that exists for the programmer's convenience. When the program is actually compiled, the symbolic constant will automatically be replaced by its equivalent numerical value.

When this program is executed, it behaves in the same manner as the program shown in Example 1.6.

EXAMPLE 1.8 Area of a Circle with Error Checking Here is a variation of the program given in Example 1.7.

```
/* program to calculate the area of a circle, with error checking */
#include <stdio.h>
#define PI 3.14159
float process(float radius);              /* function prototype */

main()
{
    float radius, area;                   /* variable declaration */
    printf("Radius = ? ");
    scanf("%f", &radius);
    if (radius < 0)
       area = 0;
    else
       area = process(radius);
    printf("Area = %f", area);
}
```

```
float process(float r)        /* function definition */

{
    float a;                  /* local variable declaration */

    a = PI * r * r;
    return(a);
}
```

This program again calculates the area of a circle. It includes the function `process`, and the symbolic constant `PI`, as discussed in the previous example. Now, however, we have added a simple error correction routine, which tests to see if the value of the radius is less than zero. (Mathematically, a negative value for the radius does not make any sense.) The test is carried out within `main`, using an `if - else` statement (see Sec. 6.6). Thus, if `radius` has a negative value, a value of zero is assigned to `area`; otherwise, the value for `area` is calculated within `process`, as before.

EXAMPLE 1.9 Areas of Several Circles The following program expands the previous sample programs by calculating the areas of several circles.

```
/* program to calculate the areas of circles, using a for loop */

#include <stdio.h>

#define PI 3.14159

float process(float radius);          /* function prototype */

main()

{
    float radius, area;               /* variable declaration */
    int count, n;                     /* variable declaration */

    printf("How many circles? ");
    scanf("%d", &n);

    for (count = 1; count <= n; ++count)   {
        printf("\nCircle no. %d:   Radius = ? ", count);
        scanf("%f", &radius);

        if (radius < 0)
            area = 0;
        else
            area = process(radius);

        printf("Area = %f\n", area);
    }
}

float process(float r)        /* function definition */

{
    float a;                  /* local variable declaration */

    a = PI * r * r;
    return(a);
}
```

In this case the total number of circles, represented by the integer variable `n`, must be entered into the computer before any calculation is carried out. The `for` statement is then used to calculate the areas repeatedly, for all `n` circles (see Sec. 6.4).

Note the use of the variable count, which is used as a counter within the for loop (i.e., within the repeated portion of the program). The value of count will increase by 1 during each pass through the loop. Also, notice the expression ++count which appears in the for statement. This is a shorthand notation for increasing the value of the counter by 1: i.e., it is equivalent to count = count + 1 (see Sec. 3.2).

When the program is executed, it generates an interactive dialog, such as that shown below. The user's responses are again underlined.

```
How many circles? 3

Circle no. 1:   Radius = ? 3
Area = 28.274309

Circle no. 2:   Radius = ? 4
Area = 50.265442

Circle no. 3:   Radius = ? 5
Area = 78.539749
```

EXAMPLE 1.10 Areas of an Unspecified Number of Circles The previous program can be improved by processing an unspecified number of circles, where the calculations continue until a value of zero is entered for the radius. This avoids the need to count, and then specify, the number of circles in advance. This feature is especially helpful when there are many sets of data to be processed.

Here is the complete program.

```
/* program to calculate the areas of circles, using a for loop;
   the number of circles is unspecified */

#include <stdio.h>

#define PI 3.14159

float process(float radius);          /* function prototype */

main()

{
    float radius, area;               /* variable declaration */
    int count;                        /* variable declaration */

    printf("To STOP, enter 0 for the radius\n");
    printf("\nRadius = ? ");
    scanf("%f", &radius);

    for (count = 1; radius != 0; ++count)   {

        if (radius < 0)
           area = 0;
        else
           area = process(radius);

        printf("Area = %f\n", area);

        printf("\nRadius = ? ");
        scanf("%f", &radius);
    }
}
```

```
float process(float r)        /* function definition */

{
    float a;                  /* local variable declaration */

    a = PI * r * r;
    return(a);
}
```

Notice that this program will display a message at the beginning of the program execution, telling the user how to end the computation.

The dialog resulting from a typical execution of this program is shown below. Once again, the user's responses are underlined.

```
    To STOP, enter 0 for the radius

    Radius = ? 3
    Area = 28.274309

    Radius = ? 4
    Area = 50.265442

    Radius = ? 5
    Area = 78.539749

    Radius = ? 0
```

EXAMPLE 1.11 Areas of an Unspecified Number of Circles Here is a variation of the program shown in the previous example.

```
/* program to calculate the areas of circles, using a while loop;
   number of circles is unspecified */

#include <stdio.h>

#define PI 3.14159

float process(float radius);          /* function declaration */


main()

{
    float radius, area;               /* variable declaration */

    printf("To STOP, enter 0 for the radius\n");
    printf("\nRadius = ? ");
    scanf("%f", &radius);

    while (radius != 0)   {

        if (radius < 0)
            area = 0;
        else
            area = process(radius);

        printf("Area = %f\n", area);

        printf("\nRadius = ? ");
        scanf("%f", &radius);
    }
}
```

```
float process(float r)        /* function definition */
{
    float a;                  /* local variable declaration */
    a = PI * r * r;
    return(a);
}
```

This program includes the same features as the program shown in the previous example. Now, however, we use a `while` statement rather than a `for` statement to carry out the repeated program execution (see Sec. 6.2). The `while` statement will continue to execute as long as the value assigned to `radius` is not zero.

In more general terms, the `while` statement will continue to execute as long as the expression contained within the parentheses is considered to be *true*. Therefore, the first line of the `while` statment can be written more briefly as

```
while (radius)    {
```

rather than

```
while (radius != 0)    {
```

because any nonzero value for `radius` will be interpreted as a *true* condition.

Some problems are better suited to the use of the `for` statement, while others are better suited to the use of `while`. The `while` statement is somewhat simpler in this particular application. There is also a third type of looping statement, called `do - while`, which is similar to the `while` statement shown above. (More about this in Chap. 6).

When this program is executed, it generates an interactive dialog that is identical to that shown in Example 1.10.

EXAMPLE 1.12 Calculating and Storing the Areas of Several Circles Some problems require that a series of calculated results be stored within the computer, perhaps for recall in a later calculation. The corresponding input data may also be stored internally, along with the calculated results. This can be accomplished through the use of *arrays*.

The following program utilizes two arrays, called `radius` and `area`, to store the radius and the area for as many as 100 different circles. Each array can be thought of as a list of numbers. The individual numbers within each list are referred to as *array elements*. The array elements are numbered, beginning with 0. Thus, the radius of the first circle will be stored within the array element `radius[0]`, the radius of the second circle will be stored within `radius[1]`, and so on. Similarly, the corresponding areas will be stored in `area[0]`, `area[1]`, etc.

Here is the complete program.

```
/* program to calculate the areas of circles, using a while loop;
   the results are stored in an array; the number of circles is unspecified */

#include <stdio.h>

#define PI 3.14159

float process(float radius);          /* function prototype */

main()

{
    int n, i = 0;                     /* variable declaration */
    float radius[100], area[100];     /* array declaration    */

    printf("To STOP, enter 0 for the radius\n\n");
    printf("Radius = ? ");
    scanf("%f", &radius[i]);
```

```
    while (radius[i])   {
        if (radius[i] < 0)
            area[i] = 0;
        else
            area[i] = process(radius[i]);
        printf("Radius = ? ");
        scanf("%f", &radius[++i]);
    }
    n = --i;        /* tag the highest value of i */
    /* display the array elements */
    printf("\nSummary of Results\n\n");
    for (i = 0; i <= n; ++i)
        printf("Radius = %f   Area = %f\n", radius[i], area[i]);
}

float process(float r)       /* function definition */
{
    float a;                 /* local variable declaration */
    a = PI * r * r;
    return(a);
}
```

An unspecified number of radii will be entered into the computer, as before. As each value for the radius is entered (i.e., as the ith value is entered), it is stored within `radius[i]`. Its corresponding area is then calculated and stored within `area[i]`. This process will continue until all of the radii have been entered, i.e., until a value of zero is entered for a radius. The entire set of stored values (i.e., the array elements whose values are nonzero) will then be displayed.

Notice the expression `++i`, which appears twice within the program. Each of these expressions causes the value of `i` to increase by 1; i.e., they are equivalent to `i = i + 1`. Similarly, the statement

```
n = --i;
```

causes the current value of `i` to be decreased by 1 and the new value assigned to `n`. In other words, the statement is equivalent to

```
n = i - 1;
```

Expressions such as `++i` and `--i` are discussed in detail in Chap. 3 (see Sec. 3.2).

When the program is executed it results in an interactive dialog, such as that shown below. The user's responses are once again underlined.

```
To STOP, enter 0 for the radius

Radius = ? 3
Radius = ? 4
Radius = ? 5
Radius = ? 0

Summary of Results

Radius = 3.000000   Area = 28.274309
Radius = 4.000000   Area = 50.265442
Radius = 5.000000   Area = 78.539749
```

This simple program does not make any use of the values that have been stored within the arrays. Its only purpose is to demonstrate the mechanics of utilizing arrays. In a more complex example, we might want to determine an average value for the areas, and then compare each individual area with the average. To do this we would have to recall the individual areas (i.e., the individual array elements `area[0]`, `area[1]`, . . ., etc.).

The use of arrays is discussed briefly in Chap. 2, and extensively in Chap. 9.

EXAMPLE 1.13 Calculating and Storing the Areas of Several Circles Here is a more sophisticated approach to the problem described in the previous example.

```
/* program to calculate the areas of circles, using a while loop;
   the results are stored in an array of structures;
   the number of circles is unspecified;
   a string is entered to identify each data set */

#include <stdio.h>

#define PI 3.14159

float process(float radius);          /* function prototype */

main()

{
    int n, i = 0;                     /* variable declaration */

    struct   {
                char text[20];
                float radius;
                float area;
    }  circle[10];                    /* structure variable declaration */

    printf("To STOP, enter END for the identifier\n");
    printf("\nIdentifier: ");
    scanf("%s", circle[i].text);
    while (circle[i].text[0] != 'E' || circle[i].text[1] != 'N'
                                    || circle[i].text[2] != 'D')   {
          printf("Radius: ");
          scanf("%f", &circle[i].radius);

          if (circle[i].radius < 0)
              circle[i].area = 0;
          else
              circle[i].area = process(circle[i].radius);

          ++i;
          printf("\nIdentifier: ");          /* next set of data */
          scanf("%s", circle[i].text);
    }

    n = --i;          /* tag the highest value of i */

    /* display the array elements */
    printf("\n\nSummary of Results\n\n");
    for (i = 0; i <= n; ++i)
         printf("%s   Radius = %f   Area = %f\n", circle[i].text,
                                                  circle[i].radius,
                                                  circle[i].area);
}
```

```
float process(float r)        /* function definition */
{
    float a;                  /* local variable declaration */
    a = PI * r * r;
    return(a);
}
```

In this program we enter a one-word *descriptor*, followed by a value of the radius, for each circle. The characters that comprise the descriptor are stored in an array called `text`. Collectively, these characters are referred to as a *string constant* (see Sec. 2.4). In this program, the maximum size of each string constant is 20 characters.

The descriptor, the radius and the corresponding area of each circle are defined as the components of a *structure* (see Chap. 11). We then define `circle` as an array of structures. That is, each element of `circle` will be a structure containing the descriptor, the radius and the area. For example, `circle[0].text` refers to the descriptor for the first circle, `circle[0].radius` refers to the radius of the first circle, and `circle[0].area` refers to the area of the first circle. (Remember that the numbering system for array elements begins with 0, not 1.)

When the program is executed, a descriptor is entered for each circle, followed by a value of the radius. This information is stored within `circle[i].text` and `circle[i].radius`. The corresponding area is then calculated and stored in `circle[i].area`. This procedure continues until the descriptor `END` is entered. All of the information stored within the array elements (i.e., the descriptor, the radius and the area for each circle) will then be displayed, and the execution will stop.

Execution of this program results in an interactive dialog, such as that shown below. Note that the user's responses are once again underlined.

```
To STOP, enter END for the identifier

Identifier: RED
Radius: 3

Identifier: WHITE
Radius: 4

Identifier: BLUE
Radius: 5

Identifier: END


Summary of Results

RED    Radius = 3.000000    Area = 28.274309
WHITE    Radius = 4.000000    Area = 50.265442
BLUE    Radius = 5.000000    Area = 78.539749
```

1.7 DESIRABLE PROGRAM CHARACTERISTICS

Before concluding this chapter let us briefly examine some important characteristics of well-written computer programs. These characteristics apply to programs that are written in *any* programming language, not just C. They can provide us with a useful set of guidelines later in this book, when we start writing our own C programs.

1. *Integrity.* This refers to the accuracy of the calculations. It should be clear that all other program enhancements will be meaningless if the calculations are not carried out correctly. Thus, the integrity of the calculations is an absolute necessity in any computer program.

2. *Clarity* refers to the overall readability of the program, with particular emphasis on its underlying logic. If a program is clearly written, it should be possible for another programmer to follow the program logic without undue effort. It should also be possible for the original author to follow his or her own program after being away from the program for an extended period of time. One of the objectives in the design of C is the development of clear, readable programs through an orderly and disciplined approach to programming.
3. *Simplicity*. The clarity and accuracy of a program are usually enhanced by keeping things as simple as possible, consistent with the overall program objectives. In fact, it may be desirable to sacrifice a certain amount of computational efficiency in order to maintain a relatively simple, straightforward program structure.
4. *Efficiency* is concerned with execution speed and efficient memory utilization. These are generally important goals, though they should not be obtained at the expense of clarity or simplicity. Many complex programs require a tradeoff between these characteristics. In such situations, experience and common sense are key factors.
5. *Modularity*. Many programs can be broken down into a series of identifiable subtasks. It is good programming practice to implement each of these subtasks as a separate program module. In C, such modules are written as functions. The use of a modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations.
6. *Generality*. Usually we will want a program to be as general as possible, within reasonable limits. For example, we may design a program to read in the values of certain key parameters rather than placing fixed values into the program. As a rule, a considerable amount of generality can be obtained with very little additional programming effort.

Review Questions

1.1 What is a mainframe computer? Where can mainframes be found? What are they generally used for?

1.2 What is a personal computer? How do personal computers differ from mainframes?

1.3 What is a supercomputer? A minicomputer? A workstation? How do these computers differ from one another? How do they differ from mainframes and personal computers?

1.4 Name four different types of data.

1.5 What is meant by a computer program? What, in general, happens when a computer program is executed?

1.6 What is computer memory? What kinds of information are stored in a computer's memory?

1.7 What is a bit? What is a byte? What is the difference between a byte and a word of memory?

1.8 What terms are used to describe the size of a computer's memory? What are some typical memory sizes?

1.9 Name some typical auxiliary memory devices. How does this type of memory differ from the computer's main memory?

1.10 What time units are used to express the speed with which elementary tasks are carried out by a computer?

1.11 What is the difference between batch processing and timesharing? What are the relative advantages and disadvantages of each?

1.12 What is meant by interactive computing? For what types of applications is interactive computing best suited?

1.13 What is machine language? How does machine language differ from high-level languages?

1.14 Name some commonly used high-level languages. What are the advantages of using high-level languages?

1.15 What is meant by compilation? What is meant by interpretation? How do these two processes differ?

1.16 What is a source program? An object program? Why are these concepts important?

1.17 What are the general characteristics of C?