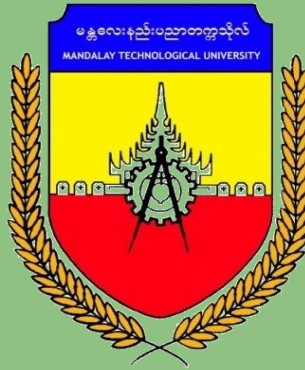


MANDALAY TECHNOLOGICAL UNIVERSITY
DEPARTMENT OF MECHATRONIC ENGINEERING



INTRODUCTION TO COMPUTER SYSTEM AND PROGRAMMING

McE-12019

Motto: Creative , Innovative , Mechatronics



BRIEF CONTENTS

1. An Overview of Computers and Programming Languages	1
2. Basic Elements of C++	27
3. Input/Output	117
4. Control Structures I (Selection)	175
5. Control Structures II (Repetition)	247
6. User-Defined Functions I	319
7. User-Defined Functions II	361
8. User-Defined Simple Data Types, Namespaces, and the <code>string</code> Type	433
9. Arrays and Strings	485
10. Applications of Arrays (Searching and Sorting) and the <code>vector</code> Type	563
11. Records (<code>structs</code>)	611
12. Classes and Data Abstraction	649
13. Inheritance and Composition	723
14. Pointers, Classes, Virtual Functions, and Abstract Classes	793
15. Overloading and Templates	861
16. Exception Handling	951
17. Recursion	991



1 CHAPTER

AN OVERVIEW OF COMPUTERS AND PROGRAMMING LANGUAGES

IN THIS CHAPTER, YOU WILL:

- Learn about different types of computers
- Explore the hardware and software components of a computer system
- Learn about the language of a computer
- Learn about the evolution of programming languages
- Examine high-level programming languages
- Discover what a compiler is and what it does
- Examine a C++ program
- Explore how a C++ program is processed
- Learn what an algorithm is and explore problem-solving techniques
- Become aware of structured design and object-oriented design programming methodologies
- Become aware of Standard C++ and ANSI/ISO Standard C++

Introduction

Terms such as “the Internet,” which were unfamiliar just 20 years ago are now common. Students in elementary school regularly “surf” the Internet and use computers to design their classroom projects. Many people use the Internet to look for information and to communicate with others. This is all made possible by the availability of different software, also known as computer programs. Without software, a computer is useless. Software is developed by using programming languages. The programming language C++ is especially well suited for developing software to accomplish specific tasks. Our main objective is to help you learn how to write programs in the C++ programming language. Before you begin programming, it is useful to understand some of the basic terminology and different components of a computer. We begin with an overview of the history of computers.

A Brief Overview of the History of Computers

The first device known to carry out calculations was the abacus. The abacus was invented in Asia but was used in ancient Babylon, China, and throughout Europe until the late middle ages. The abacus uses a system of sliding beads in a rack for addition and subtraction. In 1642, the French philosopher and mathematician Blaise Pascal invented the calculating device called the Pascaline. It had eight movable dials on wheels and could calculate sums up to eight figures long. Both the abacus and Pascaline could perform only addition and subtraction operations. Later in the 17th century, Gottfried von Leibniz invented a device that was able to add, subtract, multiply, and divide. In 1819, Joseph Jacquard, a French weaver, discovered that the weaving instructions for his looms could be stored on cards with holes punched in them. While the cards moved through the loom in sequence, needles passed through the holes and picked up threads of the correct color and texture. A weaver could rearrange the cards and change the pattern being woven. In essence, the cards programmed a loom to produce patterns in cloth. The weaving industry may seem to have little in common with the computer industry. However, the idea of storing information by punching holes on a card proved to be of great importance in the later development of computers.

In the early and mid-1800s, Charles Babbage, an English mathematician and physical scientist, designed two calculating machines—the difference engine and the analytical engine. The difference engine could perform complex operations such as squaring numbers automatically. Babbage built a prototype of the difference engine, but the actual device was never produced. The analytical engine’s design included input device, data storage, a control unit that allowed processing instructions in any sequence, and output devices. However, the designs remained in blueprint stage. Most of Babbage’s work is known through the writings of his colleague Ada Augusta, Countess of Lovelace. Augusta is considered the first computer programmer.

At the end of the 19th century, U.S. Census officials needed help in accurately tabulating the census data. Herman Hollerith invented a calculating machine that ran on electricity and used punched cards to store data. Hollerith’s machine was immensely successful. Hollerith founded the Tabulating Machine Company, which later became the computer and technology corporation known as IBM.

The first computer-like machine was the Mark I. It was built, in 1944, jointly by IBM and Harvard University under the leadership of Howard Aiken. Punched cards were used to feed data into the machine. The Mark I was 52 feet long, weighed 50 tons, and had 750,000 parts. In 1946, the ENIAC (Electronic Numerical Integrator and Calculator) was built at the University of Pennsylvania. It contained 18,000 vacuum tubes and weighed some 30 tons.

The computers that we know today use the design rules given by John von Neumann in the late 1940s. His design included components such as an arithmetic logic unit, a control unit, memory, and input/output devices. These components are described in the next section. Von Neumann's computer design makes it possible to store the programming instructions and the data in the same memory space. In 1951, the UNIVAC (Universal Automatic Computer) was built and sold to the U.S. Census Bureau.

In 1956, the invention of transistors resulted in smaller, faster, more reliable, and more energy-efficient computers. This era also saw the emergence of the software development industry, with the introduction of FORTRAN and COBOL, two early programming languages. In the next major technological advancement, transistors were replaced by tiny integrated circuits, or "chips." Chips are smaller and cheaper than transistors and can contain thousands of circuits on a single chip. They give computers tremendous processing speed.

In 1970, the microprocessor, an entire CPU on a single chip, was invented. In 1977, Stephen Wozniak and Steven Jobs designed and built the first Apple computer in their garage. In 1981, IBM introduced its personal computer (PC). In the 1980s, clones of the IBM PC made the personal computer even more affordable. By the mid-1990s, people from many walks of life were able to afford them. Computers continue to become faster and less expensive as technology advances.

Modern-day computers are powerful, reliable, and easy to use. They can accept spoken-word instructions and imitate human reasoning through artificial intelligence. Expert systems assist doctors in making diagnoses. Mobile computing applications are growing significantly. Using handheld devices, delivery drivers can access global positioning satellites (GPS) to verify customer locations for pickups and deliveries. Cell phones permit you to check your e-mail, make airline reservations, see how stocks are performing, and access your bank accounts.

Although there are several categories of computers, such as mainframe, midsize, and micro, all computers share some basic elements, described in the next section.

Elements of a Computer System

A computer is an electronic device capable of performing commands. The basic commands that a computer performs are input (get data), output (display result), storage, and performance of arithmetic and logical operations.

In today's market, personal computers are sold with descriptions such as a Pentium 4 Processor 2.80 GHz, 1 GB RAM, 250 GB HD, VX750 19" Silver Flat CRT Color Monitor, preloaded with software such as an operating system, games, encyclopedias, and application software such as word processors or money management programs. These descriptions represent two categories: hardware and software. Items such as "Pentium 4

Processor 2.80 GHz, 1GB RAM, 250 GB HD, VX750 19" Silver Flat CRT Color Monitor” fall into the hardware category; items such as “operating system, games, encyclopedias, and application software” fall into the software category. Let’s consider the hardware first.

Hardware

Major hardware components include the central processing unit (CPU); main memory (MM), also called random access memory (RAM); input/output devices; and secondary storage. Some examples of input devices are the keyboard, mouse, and secondary storage. Examples of output devices are the screen, printer, and secondary storage. Let’s look at each of these components in greater detail.

Central Processing Unit and Main Memory

The **central processing unit** is the “brain” of the computer and the single most expensive piece of hardware in a computer. The more powerful the CPU, the faster the computer. Arithmetic and logical operations are carried out inside the CPU. Figure 1-1(a) shows some hardware components.

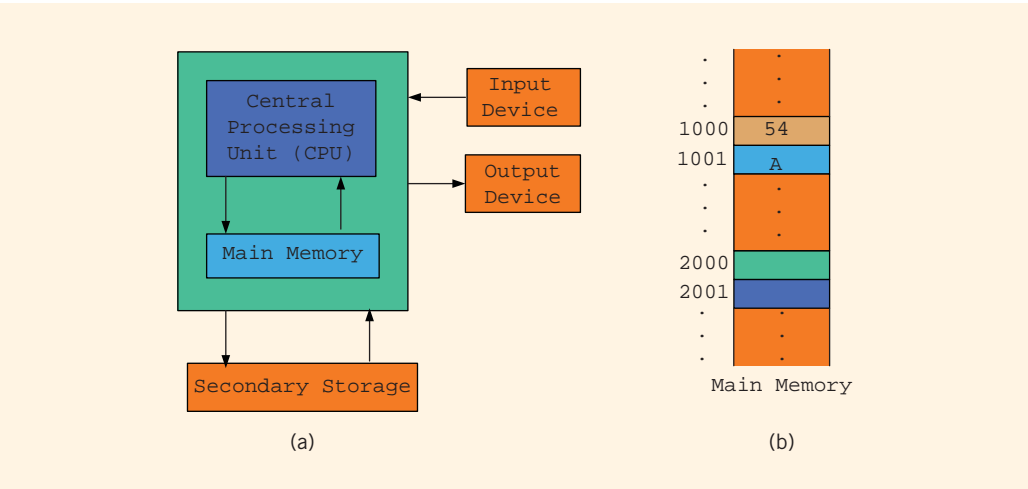


FIGURE 1-1 Hardware components of a computer and main memory

Main memory, or **random access memory**, is connected directly to the CPU. All programs must be loaded into main memory before they can be executed. Similarly, all data must be brought into main memory before a program can manipulate it. When the computer is turned off, everything in main memory is lost.

Main memory is an ordered sequence of cells, called **memory cells**. Each cell has a unique location in main memory, called the **address** of the cell. These addresses help you access the information stored in the cell. Figure 1-1(b) shows main memory with some data.

Today's computers come with main memory consisting of millions to billions of cells. Although Figure 1-1(b) shows data stored in cells, the content of a cell can be either a programming instruction or data. Moreover, this figure shows the data as numbers and letters. However, as explained later in this chapter, main memory stores everything as sequences of 0s and 1s. The memory addresses are also expressed as sequences of 0s and 1s.

SECONDARY STORAGE

Because programs and data must be stored in main memory before processing and because everything in main memory is lost when the computer is turned off, information stored in main memory must be transferred to some other device for permanent storage. The device that stores information permanently (unless the device becomes unusable or you change the information by rewriting it) is called **secondary storage**. To be able to transfer information from main memory to secondary storage, these components must be directly connected to each other. Examples of secondary storage are hard disks, flash drives, floppy disks, ZIP disks, CD-ROMs, and tapes.

Input/Output Devices

For a computer to perform a useful task, it must be able to take in data and programs and display the results of calculations. The devices that feed data and programs into computers are called **input devices**. The keyboard, mouse, and secondary storage are examples of input devices. The devices that the computer uses to display results are called **output devices**. A monitor, printer, and secondary storage are examples of output devices. Figure 1-2 shows some input and output devices.

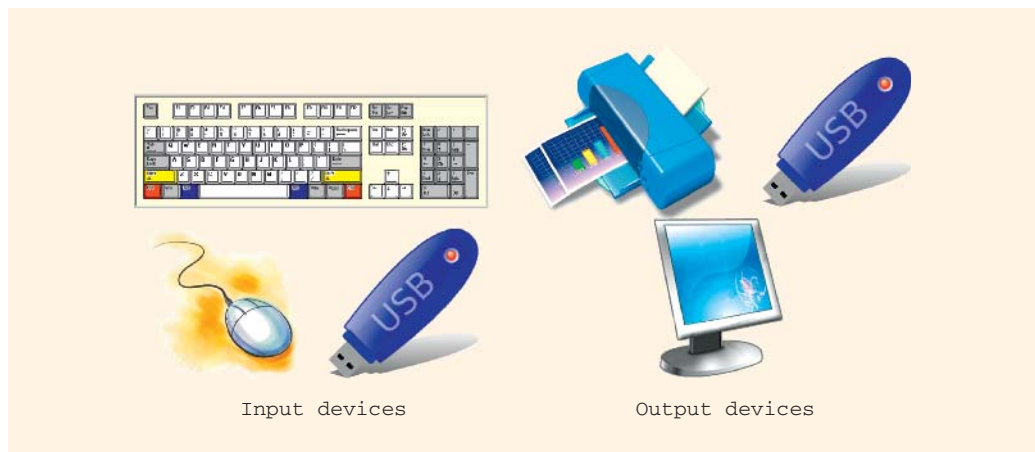


FIGURE 1-2 Some input and output devices

Software

Software are programs written to perform specific tasks. For example, word processors are programs that you use to write letters, papers, and even books. All software is written in programming languages. There are two types of programs: system programs and application programs.

System programs control the computer. The system program that loads first when you turn on your PC is called the operating system. Without an operating system, the computer is useless. The **operating system** monitors the overall activity of the computer and provides services. Some of these services include memory management, input/output activities, and storage management. The operating system has a special program that organizes secondary storage so that you can conveniently access information.

Application programs perform a specific task. Word processors, spreadsheets, and games are examples of application programs. The operating system is the program that runs application programs.

The Language of a Computer

When you press **A** on your keyboard, the computer displays **A** on the screen. But what is actually stored inside the computer's main memory? What is the language of the computer? How does it store whatever you type on the keyboard?

Remember that a computer is an electronic device. Electrical signals are used inside the computer to process information. There are two types of electrical signals: analog and digital. **Analog signals** are continuous wave forms used to represent such things as sound. Audio tapes, for example, store data in analog signals. **Digital signals** represent information with a sequence of 0s and 1s. A 0 represents a low voltage, and a 1 represents a high voltage. Digital signals are more reliable carriers of information than analog signals and can be copied from one device to another with exact precision. You might have noticed that when you make a copy of an audio tape, the sound quality of the copy is not as good as the original tape. On the other hand, when you copy a CD, the copy is as good as the original. Computers use digital signals.

Because digital signals are processed inside a computer, the language of a computer, called **machine language**, is a sequence of 0s and 1s. The digit 0 or 1 is called a **binary digit**, or **bit**. Sometimes a sequence of 0s and 1s is referred to as a **binary code** or a **binary number**.

Bit: A binary digit 0 or 1.

A sequence of eight bits is called a **byte**. Moreover, 2^{10} bytes = 1024 bytes is called a **kilobyte (KB)**. Table 1-1 summarizes the terms used to describe various numbers of bytes.

TABLE 1-1 Binary Units

Unit	Symbol	Bits/Bytes
Byte		8 bits
Kilobyte	KB	2^{10} bytes = 1024 bytes
Megabyte	MB	$1024 \text{ KB} = 2^{10} \text{ KB} = 2^{20}$ bytes = 1,048,576 bytes
Gigabyte	GB	$1024 \text{ MB} = 2^{10} \text{ MB} = 2^{30}$ bytes = 1,073,741,824 bytes
Terabyte	TB	$1024 \text{ GB} = 2^{10} \text{ GB} = 2^{40}$ bytes = 1,099,511,627,776 bytes
Petabyte	PB	$1024 \text{ TB} = 2^{10} \text{ TB} = 2^{50}$ bytes = 1,125,899,906,842,624 bytes
Exabyte	EB	$1024 \text{ PB} = 2^{10} \text{ PB} = 2^{60}$ bytes = 1,152,921,504,606,846,976 bytes
Zettabyte	ZB	$1024 \text{ EB} = 2^{10} \text{ EB} = 2^{70}$ bytes = 1,180,591,620,717,411,303,424 bytes

Every letter, number, or special symbol (such as * or {) on your keyboard is encoded as a sequence of bits, each having a unique representation. The most commonly used encoding scheme on personal computers is the *seven-bit American Standard Code for Information Interchange (ASCII)*. The ASCII data set consists of 128 characters numbered 0 through 127. That is, in the ASCII data set, the position of the first character is 0, the position of the second character is 1, and so on. In this scheme, **A** is encoded as the binary number 1000001. In fact, **A** is the 66th character in the ASCII character code, but its position is 65 because the position of the first character is 0. Furthermore, the binary number 1000001 is the binary representation of 65. The character **3** is encoded as 0110011. Note that in the ASCII character set, the position of the character **3** is 51, so the character **3** is the 52nd character in the ASCII set. It also follows that 0110011 is the binary representation of 51. For a complete list of the printable ASCII character set, refer to Appendix C.

NOTE

The number system that we use in our daily life is called the **decimal system**, or **base 10**. Because everything inside a computer is represented as a sequence of 0s and 1s, that is, binary numbers, the number system that a computer uses is called **binary**, or **base 2**. We indicated in the preceding paragraph that the number 1000001 is the binary representation of 65. Appendix E describes how to convert a number from base 10 to base 2 and vice versa.

Inside the computer, every character is represented as a sequence of *eight* bits, that is, as a byte. Now the eight-bit binary representation of 65 is **01000001**. Note that we added 0 to the left of the seven-bit representation of 65 to convert it to an eight-bit representation. Similarly, the eight-bit binary representation of 51 is **00110011**.

ASCII is a seven-bit code. Therefore, to represent each ASCII character inside the computer, you must convert the seven-bit binary representation of an ASCII character to an eight-bit binary representation. This is accomplished by adding 0 to the left of the seven-bit ASCII encoding of a character. Hence, inside the computer, the character **A** is represented as **01000001**, and the character **3** is represented as **00110011**.

There are other encoding schemes, such as EBCDIC (used by IBM) and Unicode, which is a more recent development. EBCDIC consists of 256 characters; Unicode consists of 65,536 characters. To store a character belonging to Unicode, you need two bytes.

The Evolution of Programming Languages

The most basic language of a computer, the machine language, provides program instructions in bits. Even though most computers perform the same kinds of operations, the designers of the computer may have chosen different sets of binary codes to perform the operations. Therefore, the machine language of one machine is not necessarily the same as the machine language of another machine. The only consistency among computers is that in any modern computer, all data is stored and manipulated as binary codes.

Early computers were programmed in machine language. To see how instructions are written in machine language, suppose you want to use the equation:

wages = rate • hours

to calculate weekly wages. Further, suppose that the binary code **100100** stands for load, **100110** stands for multiplication, and **100010** stands for store. In machine language, you might need the following sequence of instructions to calculate weekly wages:

```
100100 010001
100110 010010
100010 010011
```

To represent the weekly wages equation in machine language, the programmer had to remember the machine language codes for various operations. Also, to manipulate data, the programmer had to remember the locations of the data in the main memory. This need to remember specific codes made programming not only very difficult, but also error-prone.

Assembly languages were developed to make the programmer's job easier. In assembly language, an instruction is an easy-to-remember form called a **mnemonic**. Table 1-2 shows some examples of instructions in assembly language and their corresponding machine language code.

TABLE 1-2 Examples of Instructions in Assembly Language and Machine Language

Assembly Language	Machine Language
LOAD	100100
STOR	100010
MULT	100110
ADD	100101
SUB	100011

Using assembly language instructions, you can write the equation to calculate the weekly wages as follows:

```
LOAD  rate
MULT  hours
STOR  wages
```

As you can see, it is much easier to write instructions in assembly language. However, a computer cannot execute assembly language instructions directly. The instructions first have to be translated into machine language. A program called an **assembler** translates the assembly language instructions into machine language.

Assembler: A program that translates a program written in assembly language into an equivalent program in machine language.

Moving from machine language to assembly language made programming easier, but a programmer was still forced to think in terms of individual machine instructions. The next step toward making programming easier was to devise **high-level languages** that were closer to natural languages, such as English, French, German, and Spanish. Basic, FORTRAN, COBOL, Pascal, C, C++, C#, and Java are all high-level languages. You will learn the high-level language C++ in this book.

In C++, you write the weekly wages equation as follows:

```
wages = rate * hours;
```

The instruction written in C++ is much easier to understand and is self-explanatory to a novice user who is familiar with basic arithmetic. As in the case of assembly language, however, the computer cannot directly execute instructions written in a high-level language. To run on a computer, these C++ instructions first need to be translated into machine language. A program called a **compiler** translates instructions written in high-level languages into machine code.

Compiler: A program that translates instructions written in a high-level language into the equivalent machine language.

Processing a C++ Program

In the previous sections, we discussed machine language and high-level languages and showed a C++ program. Because a computer can understand only machine language, you are ready to review the steps required to process a program written in C++.

Consider the following C++ program:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "My first C++ program." << endl;

    return 0;
}
```

At this point, you need not be too concerned with the details of this program. However, if you run (execute) this program, it will display the following line on the screen:

My first C++ program.

Recall that a computer can understand only machine language. Therefore, in order to run this program successfully, the code must first be translated into machine language. In this section, we review the steps required to execute programs written in C++.

The following steps, as shown in Figure 1-3, are necessary to process a C++ program.

1. You use a text editor to create a C++ program following the rules, or *syntax*, of the high-level language. This program is called the **source code**, or **source program**. The program must be saved in a text file that has the extension **.cpp**. For example, if you saved the preceding program in the file named **FirstCPPProgram**, then its complete name is **FirstCPPProgram.cpp**.

Source program: A program written in a high-level language.

2. The C++ program given in the preceding section contains the statement **#include <iostream>**. In a C++ program, statements that begin with the symbol **#** are called preprocessor directives. These statements are processed by a program called **preprocessor**.
3. After processing preprocessor directives, the next step is to verify that the program obeys the rules of the programming language—that is, the program is syntactically correct—and translate the program into the equivalent machine language. The *compiler* checks the source program for syntax errors and, if no error is found, translates the program into the equivalent machine language. The equivalent machine language program is called an **object program**.

Object program: The machine language version of the high-level language program.

4. The programs that you write in a high-level language are developed using an integrated development environment (IDE). The IDE contains many programs that are useful in creating your program. For example, it contains the necessary code (program) to display the results of the program and several mathematical functions to make the programmer's job somewhat easier. Therefore, if certain code is already available, you can use this code rather than writing your own code. Once the program is developed and successfully compiled, you must still bring the code for the resources used from the IDE into your program to produce a final program that the computer can execute. This prewritten code (program) resides in a place called the **library**. A program called a **linker** combines the object program with the programs from libraries.

Linker: A program that combines the object program with other programs in the library and is used in the program to create the executable code.

5. You must next load the executable program into main memory for execution. A program called a **loader** accomplishes this task.

Loader: A program that loads an executable program into main memory.

6. The final step is to execute the program.

Figure 1-3 shows how a typical C++ program is processed.

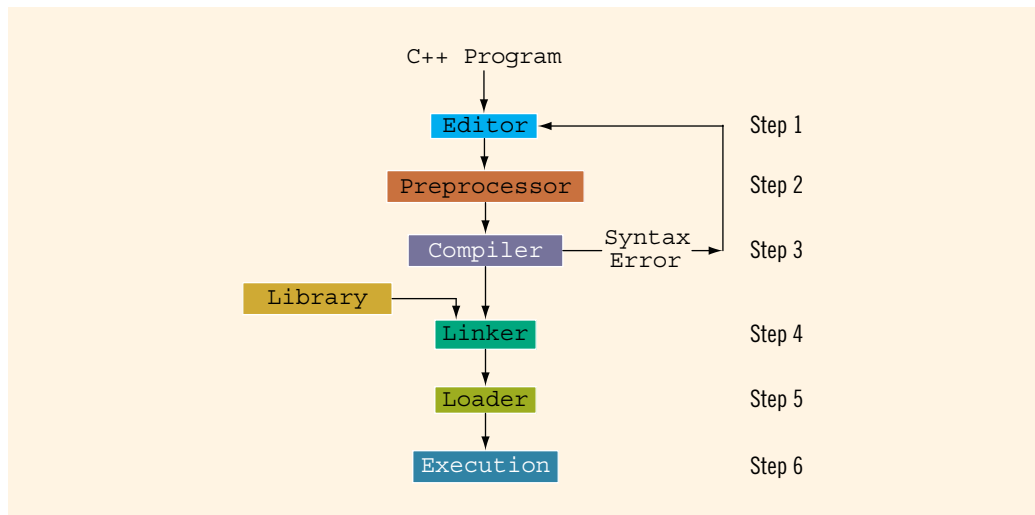


FIGURE 1-3 Processing a C++ program

As a programmer, you need to be concerned only with Step 1. That is, you must learn, understand, and master the rules of the programming language to create source programs.

As noted earlier, programs are developed using an IDE. Well-known IDEs used to create programs in the high-level language C++ include Visual C++ 2008 Express and Visual Studio .NET (from Microsoft), C++ Builder (from Borland), and CodeWarrior (from Metrowerks). These IDEs contain a text editor to create the source program, a compiler to check the source program for syntax errors, a program to link the object code with the IDE resources, and a program to execute the program.

These IDEs are quite user friendly. When you compile your program, the compiler not only identifies the syntax errors, but also typically suggests how to correct them. Moreover, with just a simple command, the object code is linked with the resources used from the IDE. For example, the command that does the linking on Visual C++ 2008 Express and Visual Studio .Net is **Build** or **Rebuild**. (For further clarification regarding the use of these commands, check the documentation of these IDEs.) If the program is not yet compiled, each of these commands first compiles the program and then links and produces the executable code.

The Web site <http://msdn.microsoft.com/en-us/beginner/bb964629.aspx> contains a video that explains how to use Visual C++ 2008 Express to write C++ programs.

Programming with the Problem Analysis–Coding–Execution Cycle

Programming is a process of problem solving. Different people use different techniques to solve problems. Some techniques are nicely outlined and easy to follow. They not only solve the problem, but also give insight into how the solution was reached. These problem-solving techniques can be easily modified if the domain of the problem changes.

To be a good problem solver and a good programmer, you must follow good problem-solving techniques. One common problem-solving technique includes analyzing a problem, outlining the problem requirements, and designing steps, called an **algorithm**, to solve the problem.

Algorithm: A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

In a programming environment, the problem-solving process requires the following three steps:

1. Analyze the problem, outline the problem and its solution requirements, and design an algorithm to solve the problem.
2. Implement the algorithm in a programming language, such as C++, and verify that the algorithm works.
3. Maintain the program by using and modifying it if the problem domain changes.

Figure 1-4 summarizes this three-step programming process.

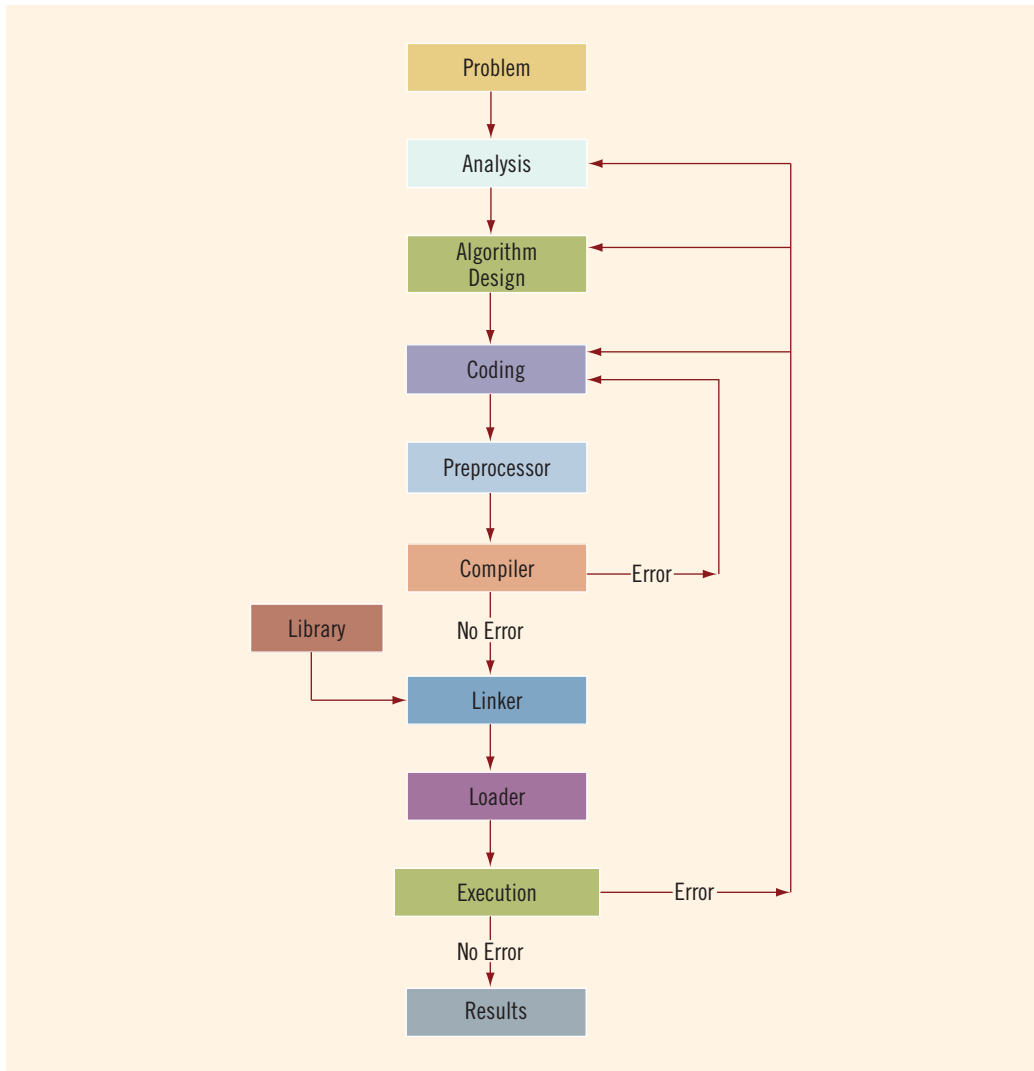


FIGURE 1-4 Problem analysis–coding–execution cycle

To develop a program to solve a problem, you start by analyzing the problem. You then design the algorithm; write the program instructions in a high-level language, or code the program; and enter the program into a computer system.

Analyzing the problem is the first and most important step. This step requires you to do the following:

1. Thoroughly understand the problem.
2. Understand the problem requirements. Requirements can include whether the program requires interaction with the user, whether it manipulates data,

whether it produces output, and what the output looks like. If the program manipulates data, the programmer must know what the data is and how it is represented. That is, you need to look at sample data. If the program produces output, you should know how the results should be generated and formatted.

3. If the problem is complex, divide the problem into subproblems and repeat Steps 1 and 2. That is, for complex problems, you need to analyze each subproblem and understand each subproblem's requirements.

After you carefully analyze the problem, the next step is to design an algorithm to solve the problem. If you broke the problem into subproblems, you need to design an algorithm for each subproblem. Once you design an algorithm, you need to check it for correctness. You can sometimes test an algorithm's correctness by using sample data. At other times, you might need to perform some mathematical analysis to test the algorithm's correctness.

Once you have designed the algorithm and verified its correctness, the next step is to convert it into an equivalent programming code. You then use a text editor to enter the programming code or the program into a computer. Next, you must make sure that the program follows the language's syntax. To verify the correctness of the syntax, you run the code through a compiler. If the compiler generates error messages, you must identify the errors in the code, remove them, and then run the code through the compiler again. When all the syntax errors are removed, the compiler generates the equivalent machine code, the linker links the machine code with the system's resources, and the loader places the program into main memory so that it can be executed.

The final step is to execute the program. The compiler guarantees only that the program follows the language's syntax. It does not guarantee that the program will run correctly. During execution, the program might terminate abnormally due to logical errors, such as division by zero. Even if the program terminates normally, it may still generate erroneous results. Under these circumstances, you may have to reexamine the code, the algorithm, or even the problem analysis.

Your overall programming experience will be successful if you spend enough time to complete the problem analysis before attempting to write the programming instructions. Usually, you do this work on paper using a pen or pencil. Taking this careful approach to programming has a number of advantages. It is much easier to discover errors in a program that is well analyzed and well designed. Furthermore, a carefully analyzed and designed program is much easier to follow and modify. Even the most experienced programmers spend a considerable amount of time analyzing a problem and designing an algorithm.

Throughout this book, you will not only learn the rules of writing programs in C++, but you will also learn problem-solving techniques. Each chapter provides several programming examples that discuss programming problems. These programming examples teach techniques of how to analyze and solve problems, design algorithms, code the algorithms into C++, and also help you understand the concepts discussed in the chapter. To gain the full benefit of this book, we recommend that you work through the programming examples at the end of each chapter.

Next, we provide examples of various problem-analysis and algorithm-design techniques.

EXAMPLE 1-1**1**

In this example, we design an algorithm to find the perimeter and area of a rectangle.

To find the perimeter and area of a rectangle, you need to know the rectangle's length and width.

The perimeter and area of the rectangle are then given by the following formulas:

```
perimeter = 2 * (length + width)
area = length * width
```

The algorithm to find the perimeter and area of the rectangle is:

1. Get the length of the rectangle.
2. Get the width of the rectangle.
3. Find the perimeter using the following equation:

```
perimeter = 2 * (length + width)
```

4. Find the area using the following equation:

```
area = length * width
```

EXAMPLE 1-2

In this example, we design an algorithm that calculates the sales tax and the price of an item sold in a particular state.

The sales tax is calculated as follows: The state's portion of the sales tax is 4%, and the city's portion of the sales tax is 1.5%. If the item is a luxury item, such as a car more than \$50,000, then there is a 10% luxury tax.

To calculate the price of the item, we need to calculate the state's portion of the sales tax, the city's portion of the sales tax, and, if it is a luxury item, the luxury tax. Suppose **salePrice** denotes the selling price of the item, **stateSalesTax** denotes the state's sales tax, **citySalesTax** denotes the city's sales tax, **luxuryTax** denotes the luxury tax, **salesTax** denotes the total sales tax, and **amountDue** denotes the final price of the item.

To calculate the sales tax, we must know the selling price of the item and whether the item is a luxury item.

The **stateSalesTax** and **citySalesTax** can be calculated using the following formulas:

```
stateSalesTax = salePrice * 0.04
citySalesTax = salePrice * 0.015
```

Next, you can determine `luxuryTax` as follows:

```
if (item is a luxury item)
    luxuryTax = salePrice * 0.1
otherwise
    luxuryTax = 0
```

Next, you can determine `salesTax` as follows:

```
salesTax = stateSalesTax + citySalesTax + luxuryTax
```

Finally, you can calculate `amountDue` as follows:

```
amountDue = salePrice + salesTax
```

The algorithm to determine `salesTax` and `amountDue` is, therefore:

1. Get the selling price of the item.
2. Determine whether the item is a luxury item.
3. Find the state's portion of the sales tax using the formula:

```
stateSalesTax = salePrice * 0.04
```

4. Find the city's portion of the sales tax using the formula:

```
citySalesTax = salePrice * 0.015
```

5. Find the luxury tax using the following formula:

```
if (item is a luxury item)
    luxuryTax = salePrice * 0.1
otherwise
    luxuryTax = 0
```

6. Find `salesTax` using the formula:

```
salesTax = stateSalesTax + citySalesTax + luxuryTax
```

7. Find `amountDue` using the formula:

```
amountDue = salePrice + salesTax
```

EXAMPLE 1-3

In this example, we design an algorithm that calculates the monthly paycheck of a salesperson at a local department store.

Every salesperson has a base salary. The salesperson also receives a bonus at the end of each month, based on the following criteria: If the salesperson has been with the store for five years or less, the bonus is \$10 for each year that he or she has worked there. If the salesperson has been with the store for more than five years, the bonus is \$20 for each year that he or she has worked there. The salesperson can earn an additional bonus as follows: If the total sales made

by the salesperson for the month are at least \$5,000 but less than \$10,000, he or she receives a 3% commission on the sale. If the total sales made by the salesperson for the month are at least \$10,000, he or she receives a 6% commission on the sale.

To calculate a salesperson's monthly paycheck, you need to know the base salary, the number of years that the salesperson has been with the company, and the total sales made by the salesperson for that month. Suppose **baseSalary** denotes the base salary, **noOfServiceYears** denotes the number of years that the salesperson has been with the store, **bonus** denotes the bonus, **totalSales** denotes the total sales made by the salesperson for the month, and **additionalBonus** denotes the additional bonus.

You can determine the bonus as follows:

```
if (noOfServiceYears is less than or equal to five)
    bonus = 10 * noOfServiceYears
otherwise
    bonus = 20 * noOfServiceYears
```

Next, you can determine the additional bonus of the salesperson as follows:

```
if (totalSales is less than 5000)
    additionalBonus = 0
otherwise
    if (totalSales is greater than or equal to 5000 and
        totalSales is less than 10000)
        additionalBonus = totalSales * (0.03)
    otherwise
        additionalBonus = totalSales * (0.06)
```

Following the above discussion, you can now design the algorithm to calculate a salesperson's monthly paycheck:

1. Get **baseSalary**.
2. Get **noOfServiceYears**.
3. Calculate bonus using the following formula:

```
if (noOfServiceYears is less than or equal to five)
    bonus = 10 * noOfServiceYears
otherwise
    bonus = 20 * noOfServiceYears
```

4. Get **totalSales**.
5. Calculate **additionalBonus** using the following formula:

```
if (totalSales is less than 5000)
    additionalBonus = 0
otherwise
    if (totalSales is greater than or equal to 5000 and
        totalSales is less than 10000)
        additionalBonus = totalSales * (0.03)
    otherwise
        additionalBonus = totalSales * (0.06)
```

6. Calculate `payCheck` using the equation:

$$\text{payCheck} = \text{baseSalary} + \text{bonus} + \text{additionalBonus}$$

EXAMPLE 1-4

In this example, we design an algorithm to play a number-guessing game.

The objective is to randomly generate an integer greater than or equal to 0 and less than 100. Then prompt the player (user) to guess the number. If the player guesses the number correctly, output an appropriate message. Otherwise, check whether the guessed number is less than the random number. If the guessed number is less than the random number generated, output the message, “Your guess is lower than the number. Guess again!”; otherwise, output the message, “Your guess is higher than the number. Guess again!”. Then prompt the player to enter another number. The player is prompted to guess the random number until the player enters the correct number.

The first step is to generate a random number, as described above. C++ provides the means to do so, which is discussed in Chapter 5. Suppose `num` stands for the random number and `guess` stands for the number guessed by the player.

After the player enters the `guess`, you can compare the `guess` with the random number as follows:

```
if (guess is equal to num)
    Print "You guessed the correct number."
otherwise
    if guess is less than num
        Print "Your guess is lower than the number. Guess again!"
    otherwise
        Print "Your guess is higher than the number. Guess again!"
```

You can now design an algorithm as follows:

1. Generate a random number and call it `num`.
2. *Repeat* the following steps until the player has guessed the correct number:
 - a. Prompt the player to enter `guess`.
 - b.

```
if (guess is equal to num)
    Print "You guessed the correct number."
otherwise
    if guess is less than num
        Print "Your guess is lower than the number. Guess again!"
    otherwise
        Print "Your guess is higher than the number. Guess again!"
```

In Chapter 5, we use this algorithm to write a C++ program to play the guessing the number game.

EXAMPLE 1-5**1**

There are 10 students in a class. Each student has taken five tests, and each test is worth 100 points. We want to design an algorithm to calculate the grade for each student, as well as the class average. The grade is assigned as follows: If the average test score is greater than or equal to 90, the grade is **A**; if the average test score is greater than or equal to 80 and less than 90, the grade is **B**; if the average test score is greater than or equal to 70 and less than 80, the grade is **C**; if the average test score is greater than or equal to 60 and less than 70, the grade is **D**; otherwise, the grade is **F**. Note that the data consists of students' names and their test scores.

This is a problem that can be divided into subproblems as follows: There are five tests, so you design an algorithm to find the average test score. Next, you design an algorithm to determine the grade. The two subproblems are to determine the average test score and to determine the grade.

Let us first design an algorithm to determine the average test score. To find the average test score, add the five test scores and then divide the sum by 5. Therefore, the algorithm is:

1. Get the five test scores.
2. Add the five test scores. Suppose **sum** stands for the sum of the test scores.
3. Suppose **average** stands for the average test score. Then:

average = **sum** / 5;

Next, you design an algorithm to determine the grade. Suppose **grade** stands for the grade assigned to a student. The following algorithm determines the grade:

```

if average is greater than or equal to 90
    grade = A
otherwise
    if average is greater than or equal to 80 and less than 90
        grade = B
    otherwise
        if average is greater than or equal to 70 and less than 80
            grade = C
        otherwise
            if average is greater than or equal to 60 and less than 70
                grade = D
        otherwise
            grade = F

```

You can use the solutions to these subproblems to design the main algorithm as follows: (Suppose **totalAverage** stands for the sum of the averages of each student's test average.)

1. **totalAverage** = 0;
2. *Repeat* the following steps for each student in the class:
 - a. Get student's name.
 - b. Use the algorithm as discussed above to find the average test score.

- c. Use the algorithm as discussed above to find the grade.
 - d. Update `totalAverage` by adding the current student's average test score.
3. Determine the class average as follows:

```
classAverage = totalAverage / 10
```

A programming exercise in Chapter 7 asks you to write a C++ program to determine the average test score and grade for each student in a class.

Programming Methodologies

Two popular approaches to programming design are the structured approach and the object-oriented approach, which are outlined below.

Structured Programming

Dividing a problem into smaller subproblems is called **structured design**. Each subproblem is then analyzed, and a solution is obtained to solve the subproblem. The solutions to all of the subproblems are then combined to solve the overall problem. This process of implementing a structured design is called **structured programming**. The structured-design approach is also known as **top-down design**, **bottom-up design**, **stepwise refinement**, and **modular programming**.

Object-Oriented Programming

Object-oriented design (OOD) is a widely used programming methodology. In OOD, the first step in the problem-solving process is to identify the components called objects, which form the basis of the solution, and to determine how these objects interact with one another. For example, suppose you want to write a program that automates the video rental process for a local video store. The two main objects in this problem are the video and the customer.

After identifying the objects, the next step is to specify for each object the relevant data and possible operations to be performed on that data. For example, for a video object, the *data* might include:

- movie name
- starring actors
- producer
- production company
- number of copies in stock

Some of the *operations* on a video object might include:

- checking the name of the movie
- reducing the number of copies in stock by one after a copy is rented
- incrementing the number of copies in stock by one after a customer returns a particular video

This illustrates that each object consists of data and operations on that data. An object combines data and operations on the data into a single unit. In OOD, the final program is a collection of interacting objects. A programming language that implements OOD is called an **object-oriented programming (OOP)** language. You will learn about the many advantages of OOD in later chapters.

Because an object consists of data and operations on that data, before you can design and use objects, you need to learn how to represent data in computer memory, how to manipulate data, and how to implement operations. In Chapter 2, you will learn the basic data types of C++ and discover how to represent and manipulate data in computer memory. Chapter 3 discusses how to input data into a C++ program and output the results generated by a C++ program.

To create operations, you write algorithms and implement them in a programming language. Because a data element in a complex program usually has many operations, to separate operations from each other and to use them effectively and in a convenient manner, you use functions to implement algorithms. After a brief introduction in Chapters 2 and 3, you will learn the details of functions in Chapters 6 and 7. Certain algorithms require that a program make decisions, a process called selection. Other algorithms might require certain statements to be repeated until certain conditions are met, a process called repetition. Still other algorithms might require both selection and repetition. You will learn about selection and repetition mechanisms, called control structures, in Chapters 4 and 5. Also, in Chapter 9, using a mechanism called an array, you will learn how to manipulate data when data items are of the same type, such as items in a list of sales figures.

Finally, to work with objects, you need to know how to combine data and operations on the data into a single unit. In C++, the mechanism that allows you to combine data and operations on the data into a single unit is called a class. You will learn how classes work, how to work with classes, and how to create classes in the chapter *Classes and Data Abstraction* (later in this book).

As you can see, you need to learn quite a few things before working with the OOD methodology. To make this learning easier and more effective, this book purposely divides control structures into two chapters (4 and 5) and user-defined functions into two chapters (6 and 7).

For some problems, the structured approach to program design will be very effective. Other problems will be better addressed by OOD. For example, if a problem requires manipulating sets of numbers with mathematical functions, you might use the structured design approach and outline the steps required to obtain the solution. The C++ library supplies a wealth of functions that you can use effectively to manipulate numbers. On the other hand, if you want to write a program that would make a candy machine operational, the OOD approach is more effective. C++ was designed especially to implement OOD. Furthermore, *OOD works well and is used in conjunction with structured design*.

Both the structured design and OOD approaches require that you master the basic components of a programming language to be an effective programmer. In Chapters 2 to 9, you will learn the basic components of C++, such as data types, input/output, control structures, user-defined functions, and arrays, required by either type of programming. We illustrate how these concepts work using the structured programming approach. Starting with the chapter *Classes and Data Abstraction*, we use the OOD approach.

ANSI/ISO Standard C++

The programming language C++ evolved from C and was designed by Bjarne Stroustrup at Bell Laboratories in the early 1980s. From the early 1980s through the early 1990s, several C++ compilers were available. Even though the fundamental features of C++ in all compilers were mostly the same, the C++ language, referred to in this book as *Standard C++*, was evolving in slightly different ways in different compilers. As a consequence, C++ programs were not always portable from one compiler to another.

To address this problem, in the early 1990s, a joint committee of the American National Standard Institution (ANSI) and International Standard Organization (ISO) was established to standardize the syntax of C++. In mid-1998, ANSI/ISO C++ language standards were approved. Most of today's compilers comply with these new standards.

This book focuses on the syntax of C++ as approved by ANSI/ISO, referred to as *ANSI/ISO Standard C++*.

QUICK REVIEW

1. A computer is an electronic device capable of performing arithmetic and logical operations.
2. A computer system has two components: hardware and software.
3. The central processing unit (CPU) and the main memory are examples of hardware components.
4. All programs must be brought into main memory before they can be executed.
5. When the power is switched off, everything in main memory is lost.
6. Secondary storage provides permanent storage for information. Hard disks, flash drives, floppy disks, ZIP disks, CD-ROMs, and tapes are examples of secondary storage.
7. Input to the computer is done via an input device. Two common input devices are the keyboard and the mouse.
8. The computer sends its output to an output device, such as the computer screen.
9. Software are programs run by the computer.
10. The operating system monitors the overall activity of the computer and provides services.

11. The most basic language of a computer is a sequence of 0s and 1s called machine language. Every computer directly understands its own machine language.
12. A bit is a binary digit, 0 or 1.
13. A byte is a sequence of eight bits.
14. A sequence of 0s and 1s is referred to as a binary code or a binary number.
15. One kilobyte (KB) is $2^{10} = 1024$ bytes; one megabyte (MB) is $2^{20} = 1,048,576$ bytes; one gigabyte (GB) is $2^{30} = 1,073,741,824$ bytes; one terabyte (TB) is $2^{40} = 1,099,511,627,776$ bytes; one petabyte (PB) is $2^{50} = 1,125,899,906,842,624$ bytes; one exabyte (EB) is $2^{60} = 1,152,921,504,606,846,976$ bytes; and one zettabyte (ZB) is $2^{70} = 1,180,591,620,717,411,303,424$ bytes.
16. Assembly language uses easy-to-remember instructions called mnemonics.
17. Assemblers are programs that translate a program written in assembly language into machine language.
18. Compilers are programs that translate a program written in a high-level language into machine code, called object code.
19. A linker links the object code with other programs provided by the integrated development environment (IDE) and used in the program to produce executable code.
20. Typically, six steps are needed to execute a C++ program: edit, preprocess, compile, link, load, and execute.
21. A loader transfers executable code into main memory.
22. An algorithm is a step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.
23. The problem-solving process has three steps: analyze the problem and design an algorithm, implement the algorithm in a programming language, and maintain the program.
24. Programs written using the structured design approach are easier to understand, easier to test and debug, and easier to modify.
25. In structured design, a problem is divided into smaller subproblems. Each subproblem is solved, and the solutions to all of the subproblems are then combined to solve the problem.
26. In object-oriented design (OOD), a program is a collection of interacting objects.
27. An object consists of data and operations on that data.
28. The ANSI/ISO Standard C++ syntax was approved in mid-1998.

EXERCISES

1. Mark the following statements as true or false.
 - a. The first device known to carry out calculations was the Pascaline.
 - b. Modern-day computers can accept spoken-word instructions but cannot imitate human reasoning.

- c. In ASCII coding, every character is coded as a sequence of 8 bits.
 - d. A compiler translates a high-level program into assembly language.
 - e. The arithmetic operations are performed inside CPU, and if an error is found, it outputs the logical errors.
 - f. A sequence of 0s and 1s is called a decimal code.
 - g. A linker links and loads the object code from main memory into the CPU for execution.
 - h. Development of a C++ program includes six steps.
 - i. A program written in a high-level programming language is called a source program.
 - j. ZB stands for zero byte.
 - k. The first step in the problem-solving process is to analyze the problem.
 - l. In object-oriented design, a program is a collection of interacting functions.
2. Name two input devices.
 3. Name two output devices.
 4. Why is secondary storage needed?
 5. What is the function of an operating system?
 6. What are the two types of programs?
 7. What are the differences between machine languages and high-level languages?
 8. What is a source program?
 9. Why do you need a compiler?
 10. What kind of errors are reported by a compiler?
 11. Why do you need to translate a program written in a high-level language into machine language?
 12. Why would you prefer to write a program in a high-level language rather than a machine language?
 13. What is linking?
 14. What are the advantages of problem analysis and algorithm design over directly writing a program in a high-level language?
 15. Design an algorithm to find the weighted average of four test scores. The four test scores and their respective weights are given in the following format:

```
testscore1 weight1
...
```

For example, sample data is as follows:

```
75 0.20
95 0.35
85 0.15
65 0.30
```

16. Design an algorithm to convert the change given in quarters, dimes, nickels, and pennies into pennies.
17. Given the radius, in inches, and price of a pizza, design an algorithm to find the price of the pizza per square inch.
18. A salesperson leaves his home every Monday and returns every Friday. He travels by company car. Each day on the road, the salesperson records the amount of gasoline put in the car. Given the starting odometer reading (that is, the odometer reading before he leaves on Monday) and the ending odometer reading (the odometer reading after he returns home on Friday), design an algorithm to find the average miles per gallon. Sample data is as follows:
68723 71289 15.75 16.30 10.95 20.65 30.00
19. To make a profit, the prices of the items sold in a furniture store are marked up by 60%. Design an algorithm to find the selling price of an item sold at the furniture store. What information do you need to find the selling price?
20. Suppose a , b , and c denote the lengths of the sides of a triangle. Then the area of the triangle can be calculated using the formula:


$$\sqrt{s(s-a)(s-b)(s-c)},$$

where $s = (1/2)(a + b + c)$. Design an algorithm that uses this formula to find the area of a triangle. What information do you need to find the area?

21. Suppose that the cost of sending an international fax is calculated as follows: Service charges \$3.00; \$.20 per page for the first 10 pages; and \$0.10 for each additional page. Design an algorithm that asks the user to enter the number of pages to be faxed. The algorithm then uses the number of pages to be faxed to calculate the amount due.
22. An ATM allows a customer to withdraw a maximum of \$500 per day. If a customer withdraws more than \$300, the service charge is 4% of the amount over \$300. If the customer does not have sufficient money in the account, the ATM informs the customer about the insufficient fund and gives the option to withdraw the money for a service charge of \$25.00. If there is no money in the account or if the account balance is negative, the ATM does not allow the customer to withdraw any money. If the amount to be withdrawn is greater than \$500, the ATM informs the customer about the maximum amount that can be withdrawn. Write an algorithm that allows the customer to enter the amount to be withdrawn. The algorithm then checks the total amount in the account, dispenses the money to the customer, and debits the account by the amount withdrawn and the service charges, if any.
23. You are given a list of students' names and their test scores. Design an algorithm that does the following:
 - a. Calculates the average test scores.
 - b. Determines and prints the names of all the students whose test scores are below the average test score.

- c. Determines the highest test score.
- d. Prints the names of all the students whose test scores are the same as the highest test score.

(You must divide this problem into subproblems as follows: The first subproblem determines the average test score. The second subproblem determines and prints the names of all the students whose test scores are below the average test score. The third subproblem determines the highest test score. The fourth subproblem prints the names of all the students whose test scores are the same as the highest test score. The main algorithm combines the solutions of the subproblems.)



CHAPTER 2

BASIC ELEMENTS OF C++

IN THIS CHAPTER, YOU WILL:

- Become familiar with the basic components of a C++ program, including functions, special symbols, and identifiers
- Explore simple data types
- Discover how to use arithmetic operators
- Examine how a program evaluates arithmetic expressions
- Learn what an assignment statement is and what it does
- Become familiar with the `string` data type
- Discover how to input data into memory using input statements
- Become familiar with the use of increment and decrement operators
- Examine ways to output results using output statements
- Learn how to use preprocessor directives and why they are necessary
- Learn how to debug syntax errors
- Explore how to properly structure a program, including using comments to document a program
- Learn how to write a C++ program

In this chapter, you will learn the basics of C++. As your objective is to learn the C++ programming language, two questions naturally arise. First, what is a computer program? Second, what is programming? A **computer program**, or a program, is a sequence of statements whose objective is to accomplish a task. **Programming** is a process of planning and creating a program. These two definitions tell the truth, but not the whole truth, about programming. It may very well take an entire book to give a good and satisfactory definition of programming. You might gain a better grasp of the nature of programming from an analogy, so let us turn to a topic about which almost everyone has some knowledge—cooking. A recipe is also a program, and everyone with some cooking experience can agree on the following:

1. It is usually easier to follow a recipe than to create one.
2. There are good recipes and there are bad recipes.
3. Some recipes are easy to follow and some are not easy to follow.
4. Some recipes produce reliable results and some do not.
5. You must have some knowledge of how to use cooking tools to follow a recipe to completion.
6. To create good new recipes, you must have much knowledge and understanding of cooking.

These same six points are also true about programming. Let us take the cooking analogy one step further. Suppose you need to teach someone how to become a chef. How would you go about it? Would you first introduce the person to good food, hoping that a taste for good food develops? Would you have the person follow recipe after recipe in the hope that some of it rubs off? Or would you first teach the use of tools and the nature of ingredients, the foods and spices, and explain how they fit together? Just as there is disagreement about how to teach cooking, there is disagreement about how to teach programming.

Learning a programming language is like learning to become a chef or learning to play a musical instrument. All three require direct interaction with the tools. You cannot become a good chef or even a poor chef just by reading recipes. Similarly, you cannot become a player by reading books about musical instruments. The same is true of programming. You must have a fundamental knowledge of the language, and you must test your programs on the computer to make sure that each program does what it is supposed to do.

A C++ Program

In this chapter, you will learn the basic elements and concepts of the C++ programming language to create C++ programs. In addition to giving examples to illustrate various concepts, we will also show C++ programs to clarify them. In this section, we provide an example of a C++ program. At this point, you need not be too concerned with the details of this program. You only need to know the effect of an *output* statement, which is introduced in this program.

Consider the C++ program in Example 2-1.

EXAMPLE 2-1

```
//*****
// This is a simple C++ program. It displays four lines
// of text, including the sum of two numbers.
//*****

#include <iostream>

using namespace std;

int main()
{
    int num;

    num = 6;

    cout << "My first C++ program." << endl;
    cout << "The sum of 2 and 3 = " << 5 << endl;
    cout << "7 + 8 = " << 7 + 8 << endl;
    cout << "Num = " << num << endl;

    return 0;
}
```

Sample Run: (When you compile and execute this program, the following four lines are displayed on the screen.)

```
My first C++ program.
The sum of 2 and 3 = 5
7 + 8 = 15
Num = 6
```

These lines are displayed by the execution of the following three statements.

```
cout << "My first C++ program." << endl;
cout << "The sum of 2 and 3 = " << 5 << endl;
cout << "7 + 8 = " << 7 + 8 << endl;
cout << "Num = " << num << endl;
```

Next, we explain how this happens. Let us first consider the following statement:

```
cout << "My first C++ program." << endl;
```

This is an example of a C++ *output* statement. It causes the computer to evaluate the *expression* after the pair of symbols << and display the result on the screen.

Usually, a C++ program contains various types of expressions such as arithmetic and strings. For example, $7 + 8$ is an arithmetic expression. Anything in double quotes is a *string*. For example, "My first C++ program." and "7 + 8 = " are strings. Typically, a string evaluates to itself. Arithmetic expressions are evaluated according to rules of arithmetic operations, which you typically learn in an algebra course. Later in this chapter, we explain how arithmetic expressions and strings are formed and evaluated.

Also note that in an output statement, `endl` *causes the insertion point to move to the beginning of the next line.* (On the screen, the insertion point is where the cursor is.) Therefore, the preceding statement causes the system to display the following line on the screen.

My first C++ program.

Let us now consider the following statement.

```
cout << "The sum of 2 and 3 = " << 5 << endl;
```

This output statement consists of two expressions. The first expression (after the first `<<`) is `"The sum of 2 and 3 = "` and the second expression (after the second `<<`) consists of the number 5. The expression `"The sum of 2 and 3 = "` is a string and evaluates to itself. (Notice the space after `=`.) The second expression, which consists of the number 5 evaluates to 5. Thus, the output of the preceding statement is:

```
The sum of 2 and 3 = 5
```

Let us now consider the following statement.

```
cout << "7 + 8 = " << 7 + 8 << endl;
```

In this output statement, the expression `"7 + 8 = "`, which is a string, evaluates to itself. Let us consider the second expression, `7 + 8`. This expression consists of the numbers 7 and 8 and the C++ arithmetic operator `+`. Therefore, the result of the expression `7 + 8` is the sum of 7 and 8, which is 15. Thus, the output of the preceding statement is:

```
7 + 8 = 15
```

Finally, consider the statement:

```
cout << "Num = " << num << endl;
```

This statement consists of the string `"Num = "`, which evaluates to itself, and the word `num`. The statement `num = 6;` assigns the value 6 to `num`. Therefore, the expression `num`, after the second `<<`, evaluates to 6. It now follows that the output of the previous statement is:

```
Num = 6
```

The last statement, that is,

```
return 0;
```

returns the value 0 to the operating system when the program terminates. We will elaborate on this statement later in this chapter.

In the next chapter, until we explain how to properly construct a C++ program, we will be using output statements such as the preceding ones to explain various concepts. After finishing Chapter 2, you should be able to write C++ programs well enough to do some computations and show results.

Before leaving this section, let us note the following about the previous C++ program. A C++ program is a collection of functions, one of which is the function `main`. Roughly

speaking, a **function** is a collection of statements, and when it is executed, it accomplishes something. The preceding program consists of the function **main**.

The first line of the program, that is,

```
#include <iostream>
```

allows us to use the (predefined object) **cout** to generate output and the (manipulator) **endl**. The second line, which is

```
using namespace std;
```

allows you to use **cout** and **endl** without the prefix **std::**. It means that if you do not include this statement, then **cout** should be used as **std::cout** and **endl** should be used as **std::endl**. We will elaborate on this later in this chapter.

The seventh line consists of the following:

```
int main()
```

This is the heading of the function **main**. The eighth line consists of a left brace. This marks the beginning of the (body) of the function **main**. The right brace (at the last line of the program) matches this left brace and marks the end of the body of the function **main**. We will explain the meaning of the other terms, such as those shown in blue, later in this book. Note that in C++, << is an operator, called the *stream insertion operator*.

The Basics of a C++ Program

A C++ program is a collection of one or more subprograms, called functions. Some functions, called **predefined** or **standard** functions, are already written and are provided as part of the system. But to accomplish most tasks, programmers must learn to write their own functions.

Every C++ program has a function called **main**. Thus, if a C++ program has only one function, it must be the function **main**. Until Chapter 6, other than using some of the predefined functions, you will mainly deal with the function **main**. By the end of this chapter, you will have learned how to write the function **main**.

If you have never seen a program written in a programming language, the C++ program in Example 2-1 may look like a foreign language. To make meaningful sentences in a foreign language, you must learn its alphabet, words, and grammar. The same is true of a programming language. To write meaningful programs, you must learn the programming language's special symbols, words, and syntax rules. The **syntax rules** tell you which statements (instructions) are legal, or accepted by the programming language, and which are not. You must also learn **semantic rules**, which determine the meaning of the instructions. The programming language's rules, symbols, and special words enable you to write programs to solve problems. The syntax rules determine which instructions are valid.

Programming language: A set of rules, symbols, and special words.

In the remainder of this section, you will learn about some of the special symbols of a C++ program. Additional special symbols are introduced as other concepts are encountered in later chapters. Similarly, syntax and semantic rules are introduced and discussed throughout the book.

Comments

The program that you write should be clear not only to you, but also to the reader of your program. Part of good programming is the inclusion of comments in the program. Typically, comments can be used to identify the authors of the program, give the date when the program is written or modified, give a brief explanation of the program, and explain the meaning of key statements in a program. In the programming examples, for the programs that we write, we will not include the date when the program is written, consistent with the standard convention for writing such books.

Comments are for the reader, not for the compiler. So when a compiler compiles a program to check for the syntax errors, it completely ignores comments. Throughout this book, comments are shown in green.

The program in Example 2-1 contains the following comments:

```
// This is a C++ program. It prints the sentence:
// Welcome to C++ Programming.
```

There are two common types of comments in a C++ program—single-line comments and multiple-line comments.

Single-line comments begin with `//` and can be placed anywhere in the line. Everything encountered in that line after `//` is ignored by the compiler. For example, consider the following statement:

```
cout << "7 + 8 = " << 7 + 8 << endl;
```

You can put comments at the end of this line as follows:

```
cout << "7 + 8 = " << 7 + 8 << endl; //prints: 7 + 8 = 15
```

This comment could be meaningful for a beginning programmer.

Multiple-line comments are enclosed between `/*` and `*/`. The compiler ignores anything that appears between `/*` and `*/`. For example, the following is an example of a multiple-line comment:

```
/*
    You can include comments that can
    occupy several lines.
*/
```

Special Symbols

The smallest individual unit of a program written in any language is called a **token**. C++'s tokens are divided into special symbols, word symbols, and identifiers.

Following are some of the special symbols:

+	-	*	/
.	;	?	,
<=	!=	==	>=

The first row includes mathematical symbols for addition, subtraction, multiplication, and division. The second row consists of punctuation marks taken from English grammar. Note that the comma is also a special symbol. In C++, commas are used to separate items in a list. Semicolons are used to end a C++ statement. Note that a blank, which is not shown above, is also a special symbol. You create a blank symbol by pressing the space bar (only once) on the keyboard. The third row consists of tokens made up of two characters that are regarded as a single symbol. No character can come between the two characters in the token, not even a blank.

Reserved Words (Keywords)

A second category of tokens is word symbols. Some of the word symbols include the following:

`int`, `float`, `double`, `char`, `const`, `void`, `return`

Reserved words are also called **keywords**. The letters that make up a reserved word are always lowercase. Like the special symbols, each is considered to be a single symbol. Furthermore, word symbols cannot be redefined within any program; that is, they cannot be used for anything other than their intended use. For a complete list of reserved words, see Appendix A.

NOTE

Throughout this book, reserved words are shown in blue.

Identifiers

A third category of tokens is identifiers. Identifiers are names of things that appear in programs, such as variables, constants, and functions. All identifiers must obey C++'s rules for identifiers.

Identifier: A C++ identifier consists of letters, digits, and the underscore character (`_`) and must begin with a letter or underscore.

Some identifiers are predefined; others are defined by the user. In the C++ program in Example 2-1, `cout` is a predefined identifier and `num` is a user-defined identifier. Two predefined identifiers that you will encounter frequently are `cout` and `cin`. You have already seen the effect of `cout`. Later in this chapter, you will learn how `cin`, which is used to input data, works. Unlike reserved words, predefined identifiers can be redefined, but it would not be wise to do so.

Identifiers can be made of only letters, digits, and the underscore character; no other symbols are permitted to form an identifier.

NOTE C++ is case sensitive—uppercase and lowercase letters are considered different. Thus, the identifier **NUMBER** is not the same as the identifier **number**. Similarly, the identifiers **X** and **x** are different.

In C++, identifiers can be of any length.

EXAMPLE 2-2

The following are legal identifiers in C++:

```
first
conversion
payRate
counter1
```

Table 2-1 shows some illegal identifiers and explains why they are illegal.

TABLE 2-1 Examples of Illegal Identifiers

Illegal Identifier	Description
employee Salary	There can be no space between employee and Salary .
Hello!	The exclamation mark cannot be used in an identifier.
one + two	The symbol + cannot be used in an identifier.
2nd	An identifier cannot begin with a digit.

NOTE Compiler vendors usually begin certain identifiers with an underscore (**_**). When the linker links the object program with the system resources provided by the integrated development environment (IDE), certain errors could occur. Therefore, it is advisable that you should not begin identifiers in your program with an underscore (**_**).

Whitespaces

Every C++ program contains whitespaces. Whitespaces include blanks, tabs, and newline characters. In a C++ program, whitespaces are used to separate special symbols, reserved words, and identifiers. Whitespaces are nonprintable in the sense that when they are printed on a white sheet of paper, the space between special symbols, reserved words, and identifiers is white. Proper utilization of whitespaces in a program is important. They can be used to make the program readable.

Data Types

The objective of a C++ program is to manipulate data. Different programs manipulate different data. A program designed to calculate an employee's paycheck will add, subtract, multiply, and divide numbers, and some of the numbers might represent hours worked and pay rate. Similarly, a program designed to alphabetize a class list will manipulate names. You wouldn't expect a cherry pie recipe to help you bake cookies. Similarly, you wouldn't use a program designed to perform arithmetic calculations to manipulate alphabetic characters. Furthermore, you wouldn't multiply or subtract names. Reflecting these kinds of underlying differences, C++ categorizes data into different types, and only certain operations can be performed on particular types of data. Although at first it may seem confusing, by being so type conscious, C++ has built-in checks to guard against errors.

Data type: A set of values together with a set of operations.

C++ data types fall into the following three categories and are illustrated in Figure 2-1:

1. Simple data type
2. Structured data type
3. Pointers

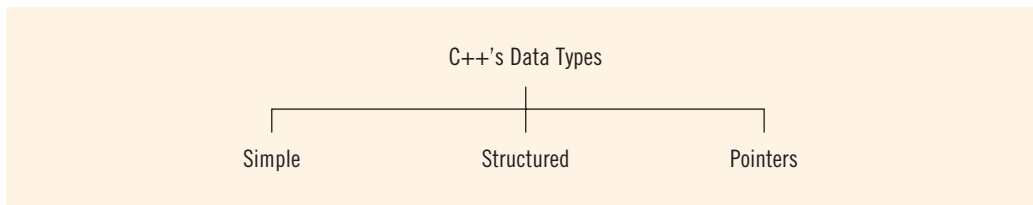


FIGURE 2-1 C++ data types

For the next few chapters, you will be concerned only with simple data types.

Simple Data Types

The simple data type is the fundamental data type in C++ because it becomes a building block for the structured data type, which you start learning about in Chapter 9. There are three categories of simple data:

1. **Integral**, which is a data type that deals with integers, or numbers without a decimal part
2. **Floating-point**, which is a data type that deals with decimal numbers
3. **Enumeration**, which is a user-defined data type

NOTE

The enumeration type is C++'s method for allowing programmers to create their own simple data types. This data type will be discussed in Chapter 8.

Integral data types are further classified into the following nine categories: `char`, `short`, `int`, `long`, `bool`, `unsigned char`, `unsigned short`, `unsigned int`, and `unsigned long`.

Why are there so many categories of the same data type? Every data type has a different set of values associated with it. For example, the `char` data type is used to represent integers between -128 and 127 . The `int` data type is used to represent integers between -2147483648 and 2147483647 , and the data type `short` is used to represent integers between -32768 and 32767 .

Note that the identifier `num` in Example 2-1 can be assigned any value belonging to the `int` data type.

Which data type you use depends on how big a number your program needs to deal with. In the early days of programming, computers and main memory were very expensive. Only a small amount of memory was available to execute programs and manipulate the data. As a result, programmers had to optimize the use of memory. Because writing a program and making it work is already a complicated process, not having to worry about the size of the memory makes for one less thing to think about. Thus, to effectively use memory, a programmer can look at the type of data used in a program and figure out which data type to use.

Newer programming languages have only five categories of simple data types: `integer`, `real`, `char`, `bool`, and the enumeration type. The integral data types that are used in this book are `int`, `bool`, and `char`.

Table 2-2 gives the range of possible values associated with these three data types and the size of memory allocated to manipulate these values.

TABLE 2-2 Values and Memory Allocation for Three Simple Data Types

Data Type	Values	Storage (in bytes)
<code>int</code>	-2147483648 to 2147483647	4
<code>bool</code>	<code>true</code> and <code>false</code>	1
<code>char</code>	-128 to 127	1

NOTE

Use this table only as a guide. Different compilers may allow different ranges of values. Check your compiler's documentation. To find the exact size of the integral data types on a particular system, you can run a program given in Appendix G (Memory Size of a System). Furthermore, to find the maximum and minimum values of these data types, you can run another program given in Appendix F (Header File `climits`).

int DATA TYPE

This section describes the `int` data type. This discussion also applies to other integral data types.

Integers in C++, as in mathematics, are numbers such as the following:

`-6728, -67, 0, 78, 36782, +763`

Note the following two rules from these examples:

1. Positive integers do not need a `+` sign in front of them.
2. No commas are used within an integer. Recall that in C++, commas are used to separate items in a list. So `36,782` would be interpreted as two integers: `36` and `782`.

bool DATA TYPE

The data type `bool` has only two values: `true` and `false`. Also, `true` and `false` are called the *logical (Boolean) values*. The central purpose of this data type is to manipulate logical (Boolean) expressions. Logical (Boolean) expressions will be formally defined and discussed in detail in Chapter 4. In C++, `bool`, `true`, and `false` are reserved words.

char DATA TYPE

The data type `char` is the smallest integral data type. It is mainly used to represent characters—that is, letters, digits, and special symbols. Thus, the `char` data type can represent every key on your keyboard. When using the `char` data type, you enclose each character represented within single quotation marks. Examples of values belonging to the `char` data type include the following:

`'A', 'a', '0', '*', '+', '$', '&', ' '`

Note that a blank space is a character and is written as `' '`, with a space between the single quotation marks.

The data type `char` allows only one symbol to be placed between the single quotation marks. Thus, the value `'abc'` is not of the type `char`. Furthermore, even though `'!='` and similar special symbols are considered to be one symbol, they are not regarded as possible values of the data type `char`. All the individual symbols located on the keyboard that are printable may be considered as possible values of the `char` data type.

Several different character data sets are currently in use. The most common are the American Standard Code for Information Interchange (ASCII) and Extended Binary-Coded Decimal Interchange Code (EBCDIC). The ASCII character set has 128 values. The EBCDIC character set has 256 values and was created by IBM. Both character sets are described in Appendix C.

Each of the 128 values of the ASCII character set represents a different character. For example, the value `65` represents `'A'`, and the value `43` represents `'+'`. Thus, each

character has a predefined ordering, which is called a **collating sequence**, in the set. The collating sequence is used when you compare characters. For example, the value representing 'B' is 66, so 'A' is smaller than 'B'. Similarly, '+' is smaller than 'A' because 43 is smaller than 65.

The 14th character in the ASCII character set is called the newline character and is represented as '\n'. (Note that the position of the newline character in the ASCII character set is 13 because the position of the first character is 0.) Even though the newline character is a combination of two characters, it is treated as one character. Similarly, the horizontal tab character is represented in C++ as '\t' and the null character is represented as '\0' (backslash followed by zero). Furthermore, the first 32 characters in the ASCII character set are nonprintable. (See Appendix C for a description of these characters.)

Floating-Point Data Types

To deal with decimal numbers, C++ provides the floating-point data type, which we discuss in this section. To facilitate the discussion, let us review a concept from a high school or college algebra course.

You may be familiar with scientific notation. For example:

```
43872918 = 4.3872918 * 107      {10 to the power of seven}
.0000265 = 2.65 * 10-5         {10 to the power of minus five}
47.9832 = 4.79832 * 101       {10 to the power of one}
```

To represent real numbers, C++ uses a form of scientific notation called **floating-point notation**. Table 2-3 shows how C++ might print a set of real numbers using one machine's interpretation of floating-point notation. In the C++ floating-point notation, the letter E stands for the exponent.

TABLE 2-3 Examples of Real Numbers Printed in C++ Floating-Point Notation

Real Number	C++ Floating-Point Notation
75.924	7.592400E1
0.18	1.800000E-1
0.0000453	4.530000E-5
-1.482	-1.482000E0
7800.0	7.800000E3

C++ provides three data types to manipulate decimal numbers: **float**, **double**, and **long double**. As in the case of integral data types, the data types **float**, **double**, and **long double** differ in the set of values.

NOTE

On most newer compilers, the data types `double` and `long double` are the same. Therefore, only the data types `float` and `double` are described here.

float: The data type `float` is used in C++ to represent any real number between $-3.4\text{E}+38$ and $3.4\text{E}+38$. The memory allocated for a value of the `float` data type is four bytes.

double: The data type `double` is used in C++ to represent any real number between $-1.7\text{E}+308$ and $1.7\text{E}+308$. The memory allocated for a value of the `double` data type is eight bytes.

The maximum and minimum values of the data types `float` and `double` are system dependent. To find these values on a particular system, you can check your compiler's documentation or, alternatively, you can run a program given in Appendix F (Header File `cfloat`).

Other than the set of values, there is one more difference between the data types `float` and `double`. The maximum number of significant digits—that is, the number of decimal places—in `float` values is six or seven. The maximum number of significant digits in values belonging to the `double` type is 15.

NOTE

For values of the `double` type, for better precision, some compilers might give more than 15 significant digits. Check your compiler's documentation.

The maximum number of significant digits is called the **precision**. Sometimes `float` values are called **single precision**, and values of type `double` are called **double precision**. If you are dealing with decimal numbers, for the most part you need only the `float` type; if you need accuracy to more than six or seven decimal places, you can use the `double` type.

NOTE

In C++, by default, floating-point numbers are considered of type `double`. Therefore, if you use the data type `float` to manipulate floating-point numbers in a program, certain compilers might give you a warning message, such as “truncation from double to float.” To avoid such warning messages, you should use the `double` data type. For illustration purposes and to avoid such warning messages in programming examples, this book mostly uses the data type `double` to manipulate floating-point numbers.

Arithmetic Operators and Operator Precedence

One of the most important uses of a computer is its ability to calculate. You can use the standard arithmetic operators to manipulate integral and floating-point data types. There are five arithmetic operators.

Arithmetic Operators: + (addition), - (subtraction or negation), * (multiplication), / (division), % (mod, (modulus or remainder)).

You can use the operators +, -, *, and / with both integral and floating-point data types. You use % with only the integral data type to find the remainder in ordinary division. When you use / with the integral data type, it gives the quotient in ordinary division. That is, integral division truncates any fractional part; there is no rounding.

Since high school, you have been accustomed to working with arithmetic expressions such as the following:

- i. -5
- ii. 8 - 7
- iii. 3 + 4
- iv. 2 + 3 * 5
- v. 5.6 + 6.2 * 3
- vi. $x + 2 * 5 + 6 / y$

(Note that in expression (vi), x and y are unknown numbers.) Formally, an **arithmetic expression** is constructed by using arithmetic operators and numbers. The numbers appearing in the expression are called **operands**. The numbers that are used to evaluate an operator are called the operands for that operator. In expression (i), the symbol - specifies that the number 5 is negative. In this expression, - has only one operand. Operators that have only one operand are called **unary operators**.

In expression (ii), the symbol - is used to subtract 7 from 8. In this expression, - has two operands, 8 and 7. Operators that have two operands are called **binary operators**.

Unary operator: An operator that has only one operand.

Binary operator: An operator that has two operands.

In expression (iii), that is, 3 + 4, 3 and 4 are the operands for the operator +. In this expression, the operator + has two operands and is a binary operator. Now consider the following expression:

+27

In this expression, the operator + indicates that the number 27 is positive. Here, + has only one operand and so acts as a unary operator.

From the preceding discussion, it follows that - and + are both unary and binary arithmetic operators. However, as arithmetic operators, *, /, and % are binary and so must have two operands.

The following examples show how arithmetic operators—especially / and %—work with integral data types. As you can see from these examples, the operator / represents the quotient in ordinary division when used with integral data types.

EXAMPLE 2-3

Arithmetic Expression	Result	Description
2 + 5	7	
13 + 89	102	
34 - 20	14	
45 - 90	-45	
2 * 7	14	
5 / 2	2	In the division 5 / 2, the quotient is 2 and the remainder is 1. Therefore, 5 / 2 with the integral operands evaluates to the quotient, which is 2.
14 / 7	2	
34 % 5	4	In the division 34 / 5, the quotient is 6 and the remainder is 4. Therefore, 34 % 5 evaluates to the remainder, which is 4.
4 % 6	4	In the division 4 / 6, the quotient is 0 and the remainder is 4. Therefore, 4 % 6 evaluates to the remainder, which is 4.

The following C++ program evaluates the preceding expressions:

```
// This program illustrates how integral expressions are
// evaluated.

#include <iostream>

using namespace std;

int main()
{
    cout << "2 + 5 = " << 2 + 5 << endl;
    cout << "13 + 89 = " << 13 + 89 << endl;
    cout << "34 - 20 = " << 34 - 20 << endl;
    cout << "45 - 90 = " << 45 - 90 << endl;
    cout << "2 * 7 = " << 2 * 7 << endl;
    cout << "5 / 2 = " << 5 / 2 << endl;
    cout << "14 / 7 = " << 14 / 7 << endl;
    cout << "34 % 5 = " << 34 % 5 << endl;
    cout << "4 % 6 = " << 4 % 6 << endl;

    return 0;
}
```

Sample Run:

```
2 + 5 = 7
13 + 89 = 102
34 - 20 = 14
45 - 90 = -45
2 * 7 = 14
```

```

5 / 2 = 2
14 / 7 = 2
34 % 5 = 4
4 % 6 = 4

```

NOTE

You should be careful when evaluating the mod operator with negative integer operands. You might not get the answer you expect. For example, $-34 \% 5 = -4$, because in the division $-34 / 5$, the quotient is -6 and the remainder is -4 . Similarly, $34 \% -5 = 4$, because in the division $-34 / 5$, the quotient is -6 and the remainder is 4 . Also, $-34 \% -5 = -4$, because in the division $-34 / -5$, the quotient is 6 and the remainder is -4 .

The following example shows how arithmetic operators work with floating-point numbers.

EXAMPLE 2-4

The following C++ program evaluates various floating-point expressions. (The details of how the expressions are evaluated are left as an exercise for you.)

```
// This program illustrates how floating-point expressions
// are evaluated.
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    cout << "5.0 + 3.5 = " << 5.0 + 3.5 << endl;
    cout << "3.0 + 9.4 = " << 3.0 + 9.4 << endl;
    cout << "16.3 - 5.2 = " << 16.3 - 5.2 << endl;
    cout << "4.2 * 2.5 = " << 4.2 * 2.5 << endl;
    cout << "5.0 / 2.0 = " << 5.0 / 2.0 << endl;
    cout << "34.5 / 6.0 = " << 34.5 / 6.0 << endl;
    cout << "34.5 / 6.5 = " << 34.5 / 6.5 << endl;
```

```
    return 0;
```

```
}
```

Sample Run:

```

5.0 + 3.5 = 8.5
3.0 + 9.4 = 12.4
16.3 - 5.2 = 11.1
4.2 * 2.5 = 10.5
5.0 / 2.0 = 2.5
34.5 / 6.0 = 5.75
34.5 / 6.5 = 5.30769

```

Order of Precedence

When more than one arithmetic operator is used in an expression, C++ uses the operator precedence rules to evaluate the expression. According to the order of precedence rules for arithmetic operators,

`*`, `/`, `%`

are at a higher level of precedence than:

`+`, `-`

Note that the operators `*`, `/`, and `%` have the same level of precedence. Similarly, the operators `+` and `-` have the same level of precedence.

When operators have the same level of precedence, the operations are performed from left to right. To avoid confusion, you can use parentheses to group arithmetic expressions. For example, using the order of precedence rules,

`3 * 7 - 6 + 2 * 5 / 4 + 6`

means the following:

```
(( (3 * 7) - 6) + ((2 * 5) / 4 )) + 6
= ((21 - 6) + (10 / 4)) + 6      (Evaluate *)
= ((21 - 6) + 2) + 6            (Evaluate /. Note that this is an integer division.)
= (15 + 2) + 6                  (Evaluate -)
= 17 + 6                        (Evaluate first +)
= 23                            (Evaluate +)
```

Note that the use of parentheses in the second example clarifies the order of precedence. You can also use parentheses to override the order of precedence rules (see Example 2-5).

EXAMPLE 2-5

In the expression:

`3 + 4 * 5`

`*` is evaluated before `+`. Therefore, the result of this expression is 23. On the other hand, in the expression:

`(3 + 4) * 5`

`+` is evaluated before `*` and the result of this expression is 35.

Because arithmetic operators are evaluated from left to right, unless parentheses are present, the **associativity** of the arithmetic operators is said to be from left to right.

NOTE

(Character Arithmetic) Because the `char` data type is also an integral data type, C++ allows you to perform arithmetic operations on `char` data. However, you should use this ability carefully. There is a difference between the character `'8'` and the integer `8`. The integer value of `8` is `8`. The integer value of `'8'` is `56`, which is the ASCII collating sequence of the character `'8'`.

When evaluating arithmetic expressions, $8 + 7 = 15$; $'8' + '7' = 56 + 55$, which yields `111`; and $'8' + 7 = 56 + 7$, which yields `63`. Furthermore, because $'8' * '7' = 56 * 55 = 3080$ and the ASCII character set has only 128 values, $'8' * '7'$ is undefined in the ASCII character data set.

These examples illustrate that many things can go wrong when you are performing character arithmetic. If you must employ them, use arithmetic operations on the `char` data type with caution.

Expressions

To this point, we have discussed only arithmetic operators. In this section, we now discuss arithmetic expressions in detail. Arithmetic expressions were introduced in the last section.

If all operands (that is, numbers) in an expression are integers, the expression is called an **integral expression**. If all operands in an expression are floating-point numbers, the expression is called a **floating-point** or **decimal expression**. An integral expression yields an integral result; a floating-point expression yields a floating-point result. Looking at some examples will help clarify these definitions.

EXAMPLE 2-6

Consider the following C++ integral expressions:

```
2 + 3 * 5
3 + x - y / 7
x + 2 * (y - z) + 18
```

In these expressions, `x`, `y`, and `z` represent variables of the integer type; that is, they can hold integer values. Variables are discussed later in this chapter.

EXAMPLE 2-7

Consider the following C++ floating-point expressions:

```
12.8 * 17.5 - 34.50
x * 10.5 + y - 16.2
```

Here, `x` and `y` represent variables of the floating-point type; that is, they can hold floating-point values. Variables are discussed later in this chapter.

Evaluating an integral or a floating-point expression is straightforward. As before, when operators have the same precedence, the expression is evaluated from left to right. You can always use parentheses to group operands and operators to avoid confusion.

Mixed Expressions

An expression that has operands of different data types is called a **mixed expression**. A mixed expression contains both integers and floating-point numbers. The following expressions are examples of mixed expressions:

```
2 + 3.5
6 / 4 + 3.9
5.4 * 2 - 13.6 + 18 / 2
```

In the first expression, the operand `+` has one integer operand and one floating-point operand. In the second expression, both operands for the operator `/` are integers, the first operand of `+` is the result of `6 / 4`, and the second operand of `+` is a floating-point number. The third example is an even more complicated mix of integers and floating-point numbers. The obvious question is: How does C++ evaluate mixed expressions?

Two rules apply when evaluating a mixed expression:

1. When evaluating an operator in a mixed expression:
 - a. If the operator has the same types of operands (that is, either both integers or both floating-point numbers), the operator is evaluated according to the type of the operands. Integer operands thus yield an integer result; floating-point numbers yield a floating-point number.
 - b. If the operator has both types of operands (that is, one is an integer and the other is a floating-point number), then during calculation, the integer is changed to a floating-point number with the decimal part of zero and the operator is evaluated. The result is a floating-point number.
2. The entire expression is evaluated according to the precedence rules; the multiplication, division, and modulus operators are evaluated before the addition and subtraction operators. Operators having the same level of precedence are evaluated from left to right. Grouping is allowed for clarity.

From these rules, it follows that when evaluating a mixed expression, you concentrate on one operator at a time, using the rules of precedence. If the operator to be evaluated has operands of the same data type, evaluate the operator using Rule 1(a). That is, an operator with integer operands will yield an integer result, and an operator with floating-point operands will yield a floating-point result. If the operator to be evaluated has one integer operand and one floating-point operand, before evaluating this operator, convert the integer operand to a floating-point number with the decimal part of 0. The following examples show how to evaluate mixed expressions.

EXAMPLE 2-8

Mixed Expression	Evaluation	Rule Applied
$3 / 2 + 5.5$	$= 1 + 5.5$ $= 6.5$	$3 / 2 = 1$ (integer division; Rule 1(a)) $(1 + 5.5$ $= 1.0 + 5.5$ (Rule 1(b)) $= 6.5)$
$15.6 / 2 + 5$	$= 7.8 + 5$ $= 12.8$	$15.6 / 2$ $= 15.6 / 2.0$ (Rule 1(b)) $= 7.8$ $7.8 + 5$ $= 7.8 + 5.0$ (Rule 1(b)) $= 12.8$
$4 + 5 / 2.0$	$= 4 + 2.5$ $= 6.5$	$5 / 2.0 = 5.0 / 2.0$ (Rule 1(b)) $= 2.5$ $4 + 2.5 = 4.0 + 2.5$ (Rule 1(b)) $= 6.5$
$4 * 3 + 7 / 5 - 25.5$	$= 12 + 7 / 5 - 25.5$ $= 12 + 1 - 25.5$ $= 13 - 25.5$ $= -12.5$	$4 * 3 = 12$; (Rule 1(a)) $7 / 5 = 1$ (integer division; Rule 1(a)) $12 + 1 = 13$; (Rule 1(a)) $13 - 25.5 = 13.0 - 25.5$ (Rule 1(b)) $= -12.5$

The following C++ program evaluates the preceding expressions:

// This program illustrates how mixed expressions are evaluated.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "3 / 2 + 5.5 = " << 3 / 2 + 5.5 << endl;
    cout << "15.6 / 2 + 5 = " << 15.6 / 2 + 5 << endl;
    cout << "4 + 5 / 2.0 = " << 4 + 5 / 2.0 << endl;
    cout << "4 * 3 + 7 / 5 - 25.5 = "
        << 4 * 3 + 7 / 5 - 25.5
        << endl;

    return 0;
}
```

Sample Run:

```
3 / 2 + 5.5 = 6.5
15.6 / 2 + 5 = 12.8
4 + 5 / 2.0 = 6.5
4 * 3 + 7 / 5 - 25.5 = -12.5
```

These examples illustrate that an integer is not converted to a floating-point number unless the operator to be evaluated has one integer and one floating-point operand.

Type Conversion (Casting)

In the previous section, you learned that when evaluating an arithmetic expression, if the operator has mixed operands, the integer value is changed to a floating-point value with the zero decimal part. When a value of one data type is automatically changed to another data type, an **implicit type coercion** is said to have occurred. As the examples in the preceding section illustrate, if you are not careful about data types, implicit type coercion can generate unexpected results.

To avoid implicit type coercion, C++ provides for explicit type conversion through the use of a cast operator. The **cast operator**, also called **type conversion** or **type casting**, takes the following form:

```
static_cast<dataTypeName> (expression)
```

First, the expression is evaluated. Its value is then converted to a value of the type specified by `dataTypeName`. In C++, `static_cast` is a reserved word.

When converting a floating-point (decimal) number to an integer using the cast operator, you simply drop the decimal part of the floating-point number. That is, the floating-point number is truncated. Example 2-9 shows how cast operators work. Be sure you understand why the last two expressions evaluate as they do.

EXAMPLE 2-9

Expression	Evaluates to
<code>static_cast<int> (7.9)</code>	7
<code>static_cast<int> (3.3)</code>	3
<code>static_cast<double> (25)</code>	25.0
<code>static_cast<double> (5 + 3)</code>	= <code>static_cast<double> (8)</code> = 8.0
<code>static_cast<double> (15) / 2</code>	= 15.0 / 2 (because <code>static_cast<double> (15)</code> = 15.0) = 15.0 / 2.0 = 7.5
<code>static_cast<double> (15 / 2)</code>	= <code>static_cast<double> (7)</code> (because 15 / 2 = 7) = 7.0
<code>static_cast<int> (7.8 +</code> <code>static_cast<double> (15) / 2)</code>	= <code>static_cast<int> (7.8 + 7.5)</code> = <code>static_cast<int> (15.3)</code> = 15
<code>static_cast<int> (7.8 +</code> <code>static_cast<double> (15 / 2))</code>	= <code>static_cast<int> (7.8 + 7.0)</code> = <code>static_cast<int> (14.8)</code> = 14

The following C++ program evaluates the preceding expressions:

```
// This program illustrates how explicit type conversion works.

#include <iostream>
using namespace std;

int main()
{
    cout << "static_cast<int>(7.9) = "
          << static_cast<int>(7.9)
          << endl;
    cout << "static_cast<int>(3.3) = "
          << static_cast<int>(3.3)
          << endl;
    cout << "static_cast<double>(25) = "
          << static_cast<double>(25)
          << endl;
    cout << "static_cast<double>(5 + 3) = "
          << static_cast<double>(5 + 3)
          << endl;
    cout << "static_cast<double>(15) / 2 = "
          << static_cast<double>(15) / 2
          << endl;
    cout << "static_cast<double>(15 / 2) = "
          << static_cast<double>(15 / 2)
          << endl;
    cout << "static_cast<int>(7.8 + static_cast<double>(15) / 2) = "
          << static_cast<int>(7.8 + static_cast<double>(15) / 2)
          << endl;
    cout << "static_cast<int>(7.8 + static_cast<double>(15 / 2)) = "
          << static_cast<int>(7.8 + static_cast<double>(15 / 2))
          << endl;

    return 0;
}
```

Sample Run:

```
static_cast<int>(7.9) = 7
static_cast<int>(3.3) = 3
static_cast<double>(25) = 25
static_cast<double>(5 + 3) = 8
static_cast<double>(15) / 2 = 7.5
static_cast<double>(15 / 2) = 7
static_cast<int>(7.8 + static_cast<double>(15) / 2) = 15
static_cast<int>(7.8 + static_cast<double>(15 / 2)) = 14
```

Note that the value of the expression `static_cast<double>(25)` is 25.0. However, it is output as 25 rather than 25.0. This is because we have not yet discussed how to output decimal numbers with 0 decimal parts to show the decimal point and the trailing zeros. Chapter 3 explains how to output decimal numbers in a desired format. Similarly, the output of other decimal numbers with zero decimal parts is without the decimal point and the 0 decimal part.

NOTE

In C++, the cast operator can also take the form `dataType (expression)`. This form is called C-like casting. For example, `double (5) = 5.0` and `int (17.6) = 17`. However, `static_cast` is more stable than C-like casting.

2

You can also use cast operators to explicitly convert `char` data values into `int` data values and `int` data values into `char` data values. To convert `char` data values into `int` data values, you use a collating sequence. For example, in the ASCII character set, `static_cast<int>('A')` is 65 and `static_cast<int>('8')` is 56. Similarly, `static_cast<char>(65)` is 'A' and `static_cast<char>(56)` is '8'.

Earlier in this chapter, you learned how arithmetic expressions are formed and evaluated in C++. If you want to use the value of one expression in another expression, first you must save the value of the expression. There are many reasons to save the value of an expression. Some expressions are complex and may require a considerable amount of computer time to evaluate. By calculating the values once and saving them for further use, you not only save computer time and create a program that executes more quickly, you also avoid possible typographical errors. In C++, expressions are evaluated, and if the value is not saved, it is lost. That is, unless it is saved, the value of an expression cannot be used in later calculations. In the next section, you will learn how to save the value of an expression and use it in subsequent calculations.

Before leaving the discussion of data types, let us discuss one more data type—`string`.

string Type

The data type `string` is a programmer-defined data type. It is not directly available for use in a program like the simple data types discussed earlier. To use this data type, you need to access program components from the library, which will be discussed later in this chapter. The data type `string` is a feature of ANSI/ISO Standard C++.

NOTE

Prior to the ANSI/ISO C++ language standard, the standard C++ library did not provide a `string` data type. Compiler vendors often supplied their own programmer-defined `string` type, and the syntax and semantics of string operations often varied from vendor to vendor.

A `string` is a sequence of zero or more characters. Strings in C++ are enclosed in double quotation marks. A string containing no characters is called a **null** or **empty** string. The following are examples of strings. Note that `""` is the empty string.

```
"William Jacob"
"Mickey"
""
```

Every character in a string has a relative position in the string. The position of the first character is 0, the position of the second character is 1, and so on. The length of a string is the number of characters in it.

EXAMPLE 2-10

String	Position of a Character in the String	Length of the String
"William Jacob"	Position of 'W' is 0. Position of the first 'i' is 1. Position of ' ' (the space) is 7. Position of 'J' is 8. Position of 'b' is 12.	13
"Mickey"	Position of 'M' is 0. Position of 'i' is 1. Position of 'c' is 2. Position of 'k' is 3. Position of 'e' is 4. Position of 'y' is 5.	6

When determining the length of a string, you must also count any spaces in the string. For example, the length of the following string is 22.

"It is a beautiful day."

The string type is very powerful and more complex than simple data types. It provides many operations to manipulate strings. For example, it provides operations to find the length of a string, extract part of a string, and compare strings. You will learn about this data over the next few chapters.

Input

As noted earlier, the main objective of a C++ program is to perform calculations and manipulate data. Recall that data must be loaded into main memory before it can be manipulated. In this section, you will learn how to put data into the computer's memory. Storing data in the computer's memory is a two-step process:

1. Instruct the computer to allocate memory.
2. Include statements in the program to put data into the allocated memory.

Allocating Memory with Constants and Variables

When you instruct the computer to allocate memory, you tell it not only what names to use for each memory location, but also what type of data to store in those memory locations. Knowing the location of data is essential, because data stored in one memory location might be needed at several places in the program. As you saw earlier, knowing what data type you have is crucial for performing accurate calculations. It is also critical to know whether your data needs to remain fixed throughout program execution or whether it should change.

Some data must stay the same throughout a program. For example, the pay rate is usually the same for all part-time employees. A conversion formula that converts inches into

centimeters is fixed, because 1 inch is always equal to 2.54 centimeters. When stored in memory, this type of data needs to be protected from accidental changes during program execution. In C++, you can use a **named constant** to instruct a program to mark those memory locations in which data is fixed throughout program execution.

Named constant: A memory location whose content is not allowed to change during program execution.

To allocate memory, we use C++'s declaration statements. The syntax to declare a named constant is:

```
const dataType identifier = value;
```

In C++, `const` is a reserved word.

EXAMPLE 2-11

Consider the following C++ statements:

```
const double CONVERSION = 2.54;  
const int NO_OF_STUDENTS = 20;  
const char BLANK = ' ';
```

The first statement tells the compiler to allocate memory (eight bytes) to store a value of type `double`, call this memory space `CONVERSION`, and store the value 2.54 in it. Throughout a program that uses this statement, whenever the conversion formula is needed, the memory space `CONVERSION` can be accessed. The meaning of the other statements is similar.

Note that the identifier for a named constant is in uppercase letters. Even though there are no written rules, C++ programmers typically prefer to use uppercase letters to name a named constant. Moreover, if the name of a named constant is a combination of more than one word, called a *run-together word*, then the words are separated using an underscore. For example, in the preceding example, `NO_OF_STUDENTS` is a run-together word.

NOTE

As noted earlier, the default type of floating-point numbers is `double`. Therefore, if you declare a named constant of type `float`, then you must specify that the value is of type `float` as follows:

```
const float CONVERSION = 2.54f;
```

Otherwise, the compiler will generate a warning message. Notice that `2.54f` says that it is a `float` value. Recall that the memory size for `float` values is four bytes; for `double` values, eight bytes. Because memory size is of little concern these days, as indicated earlier, we will mostly use the type `double` to work with floating-point values.

Using a named constant to store fixed data, rather than using the data value itself, has one major advantage. If the fixed data changes, you do not need to edit the entire program and change the old value to the new value wherever the old value is used. Instead, you can make the change at just one place, recompile the program, and execute it using the new value throughout. In addition, by storing a value and referring to that memory location whenever the value is needed, you avoid typing the same value again and again and prevent accidental typos. If you misspell the name of the constant value's location, the computer will warn you through an error message, but it will not warn you if the value is mistyped.

In some programs, data needs to be modified during program execution. For example, after each test, the average test score and the number of tests taken changes. Similarly, after each pay increase, the employee's salary changes. This type of data must be stored in those memory cells whose contents can be modified during program execution. In C++, memory cells whose contents can be modified during program execution are called variables.

Variable: A memory location whose content may change during program execution.

The syntax for declaring one variable or multiple variables is:

```
dataType identifier, identifier, . . . ;
```

EXAMPLE 2-12

Consider the following statements:

```
double amountDue;  
int counter;  
char ch;  
int x, y;  
string name;
```

The first statement tells the compiler to allocate enough memory to store a value of the type `double` and call it `amountDue`. The second and third statements have similar conventions. The fourth statement tells the compiler to allocate two different memory spaces, each large enough to store a value of the type `int`; name the first memory space `x`; and name the second memory space `y`. The fifth statement tells the compiler to allocate memory space to store a string and call it `name`.

As in the case of naming named constants, there are no written rules for naming variables. However, C++ programmers typically use lowercase letters to declare variables. If a variable name is a combination of more than one word, then the first letter of each word, except the first word, is uppercase. (For example, see the variable `amountDue` in the preceding example.)

From now on, when we say “variable,” we mean a variable memory location.

NOTE

In C++, you must declare all identifiers before you can use them. If you refer to an identifier without declaring it, the compiler will generate an error message (syntax error), indicating that the identifier is not declared. Therefore, to use either a named constant or a variable, you must first declare it.

2

Now that data types, variables, and constants have been defined and discussed, it is possible to offer a formal definition of simple data types. A data type is called **simple** if the variable or named constant of that type can store only one value at a time. For example, if **x** is an **int** variable, at a given time, only one value can be stored in **x**.

Putting Data into Variables

Now that you know how to declare variables, the next question is: How do you put data into those variables? In C++, you can place data into a variable in two ways:

1. Use C++'s assignment statement.
2. Use input (read) statements.

Assignment Statement

The assignment statement takes the following form:

```
variable = expression;
```

In an assignment statement, the value of the **expression** should match the data type of the **variable**. The expression on the right side is evaluated, and its value is assigned to the variable (and thus to a memory location) on the left side.

A variable is said to be **initialized** the first time a value is placed in the variable.

In C++, = is called the **assignment operator**.

EXAMPLE 2-13

Suppose you have the following variable declarations:

```
int num1, num2;  
double sale;  
char first;  
string str;
```

Now consider the following assignment statements:

```
num1 = 4;  
num2 = 4 * 5 - 11;  
sale = 0.02 * 1000;  
first = 'D';  
str = "It is a sunny day.";
```

For each of these statements, the computer first evaluates the expression on the right and then stores that value in a memory location named by the identifier on the left. The first statement stores the value 4 in `num1`, the second statement stores 9 in `num2`, the third statement stores 20.00 in `sale`, and the fourth statement stores the character D in `first`. The fifth statement stores the string "It is a sunny day." in the variable `str`.

The following C++ program shows the effect of the preceding statements:

```
// This program illustrates how data in the variables are
// manipulated.
```

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    int num1, num2;
    double sale;
    char first;
    string str;

    num1 = 4;
    cout << "num1 = " << num1 << endl;

    num2 = 4 * 5 - 11;
    cout << "num2 = " << num2 << endl;

    sale = 0.02 * 1000;
    cout << "sale = " << sale << endl;

    first = 'D';
    cout << "first = " << first << endl;

    str = "It is a sunny day.";
    cout << "str = " << str << endl;

    return 0;
}
```

Sample Run:

```
num1 = 4
num2 = 9
sale = 20
first = D
str = It is a sunny day.
```


For the most part, the preceding program is straightforward. Let us take a look at the output statement:

```
cout << " num1 = " << num1 << endl;
```

This output statement consists of the string " num1 = ", the operator <<, and the variable num1. Here, first the value of the string " num1 = " is output, and then the value of the variable num1 is output. The meaning of the other output statements is similar.

A C++ statement such as:

```
num = num + 2;
```

means “evaluate whatever is in num, add 2 to it, and assign the new value to the memory location num.” The expression on the right side must be evaluated first; that value is then assigned to the memory location specified by the variable on the left side. Thus, the sequence of C++ statements:

```
num = 6;
num = num + 2;
```

and the statement:

```
num = 8;
```

both assign 8 to num. Note that if num has not been initialized, the statement num = num + 2 might give unexpected results and/or the compiler might generate a warning message indicating that the variable has not been initialized.

The statement num = 5; is read as “num becomes 5” or “num gets 5” or “num is assigned the value 5.” Reading the statement as “num equals 5” is incorrect, especially for statements such as num = num + 2;. Each time a new value is assigned to num, the old value is overwritten.

EXAMPLE 2-14

Suppose that num1, num2, and num3 are `int` variables and the following statements are executed in sequence.

1. num1 = 18;
2. num1 = num1 + 27;
3. num2 = num1;
4. num3 = num2 / 5;
5. num3 = num3 / 4;

The following table shows the values of the variables after the execution of each statement. (A ? indicates that the value is unknown. The orange color in a box shows that the value of that variable is changed.)

	Values of the Variables			Explanation
Before Statement 1	?	?	?	
	num1	num2	num3	
After Statement 1	18	?	?	
	num1	num2	num3	
After Statement 2	45	?	?	$\text{num1} + 27 = 18 + 27 = 45$. This value is assigned to num1 , which replaces the old value of num1 .
	num1	num2	num3	
After Statement 3	45	45	?	Copy the value of num1 into num2 .
	num1	num2	num3	
After Statement 4	45	45	9	$\text{num2} / 5 = 45 / 5 = 9$. This value is assigned to num3 . So num3 = 9.
	num1	num2	num3	
After Statement 5	45	45	2	$\text{num3} / 4 = 9 / 4 = 2$. This value is assigned to num3 , which replaces the old value of num3 .
	num1	num2	num3	

Thus, after the execution of the statement in Line 5, **num1** = 45, **num2** = 45, and **num3** = 2.

Tracing values through a sequence, called a **walk-through**, is a valuable tool to learn and practice. Try it in the sequence above. You will learn more about how to walk through a sequence of C++ statements later in this chapter.

NOTE

Suppose that **x**, **y**, and **z** are **int** variables. The following is a legal statement in C++:

```
x = y = z;
```

In this statement, first the value of **z** is assigned to **y**, and then the new value of **y** is assigned to **x**. Because the assignment operator, **=**, is evaluated from right to left, the **associativity** of the **assignment operator** is said to be from right to left.

Saving and Using the Value of an Expression

Now that you know how to declare variables and put data into them, you can learn how to save the value of an expression. You can then use this value in a later expression without using the expression itself, thereby answering the question raised earlier in this chapter. To save the value of an expression and use it in a later expression, do the following:

1. Declare a variable of the appropriate data type. For example, if the result of the expression is an integer, declare an **int** variable.

2. Assign the value of the expression to the variable that was declared, using the assignment statement. This action saves the value of the expression into the variable.
3. Wherever the value of the expression is needed, use the variable holding the value. The following example further illustrates this concept.

EXAMPLE 2-15

Suppose that you have the following declaration:

```
int a, b, c, d;  
int x, y;
```

Further suppose that you want to evaluate the expressions $-b + (b^2 - 4ac)$ and $-b - (b^2 - 4ac)$ and assign the values of these expressions to **x** and **y**, respectively. Because the expression $b^2 - 4ac$ appears in both expressions, you can first calculate the value of this expression and save its value in **d**. You can then use the value of **d** to evaluate the expressions, as shown by the following statements:

```
d = b * b - 4 * a * c;  
x = -b + d;  
y = -b - d;
```

Earlier, you learned that if a variable is used in an expression, the expression would yield a meaningful value only if the variable has first been initialized. You also learned that after declaring a variable, you can use an assignment statement to initialize it. It is possible to initialize and declare variables at the same time. Before we discuss how to use an input (read) statement, we address this important issue.

Declaring and Initializing Variables

When a variable is declared, C++ may not automatically put a meaningful value in it. In other words, C++ may not automatically initialize variables. For example, the **int** and **double** variables may not be initialized to 0, as happens in some programming languages. This does not mean, however, that there is no value in a variable after its declaration. When a variable is declared, memory is allocated for it.

Recall from Chapter 1 that main memory is an ordered sequence of cells, and each cell is capable of storing a value. Also, recall that the machine language is a sequence of 0s and 1s, or bits. Therefore, data in a memory cell is a sequence of bits. These bits are nothing but electrical signals, so when the computer is turned on, some of the bits are 1 and some are 0. The state of these bits depends on how the system functions. However, when you instruct the computer to store a particular value in a memory cell, the bits are set according to the data being stored.

During data manipulation, the computer takes the value stored in particular cells and performs a calculation. If you declare a variable and do not store a value in it, the memory cell still has a value—usually the value of the setting of the bits from their last use—and you have no way to know what this value is.

If you only declare a variable and do not instruct the computer to put data into the variable, the value of that variable is garbage. However, the computer does not warn us, regards whatever values are in memory as legitimate, and performs calculations using those values in memory. Using a variable in an expression without initializing it produces erroneous results. To avoid these pitfalls, C++ allows you to initialize variables while they are being declared. For example, consider the following C++ statements in which variables are first declared and then initialized:

```
int first, second;
char ch;
double x;

first = 13;
second = 10;
ch = ' ';
x = 12.6;
```

You can declare and initialize these variables at the same time using the following C++ statements:

```
int first = 13, second = 10;
char ch = ' ';
double x = 12.6;
```

The first C++ statement declares two `int` variables, `first` and `second`, and stores 13 in `first` and 10 in `second`. The meaning of the other statements is similar.

In reality, not all variables are initialized during declaration. It is the nature of the program or the programmer's choice that dictates which variables should be initialized during declaration. The key point is that all variables must be initialized before they are used.

Input (Read) Statement

Previously, you learned how to put data into variables using the assignment statement. In this section, you will learn how to put data into variables from the *standard input device*, using C++'s input (or read) statements.

NOTE

In most cases, the standard input device is the keyboard.

When the computer gets the data from the keyboard, the user is said to be acting interactively.

Putting data into variables from the standard input device is accomplished via the use of `cin` and the operator `>>`. The syntax of `cin` together with `>>` is:

```
cin >> variable >> variable ...;
```

This is called an **input (read)** statement. In C++, `>>` is called the **stream extraction operator**.

NOTE

In a syntax, the shading indicates the part of the definition that is optional. Furthermore, throughout this book, the syntax is enclosed in yellow boxes.

EXAMPLE 2-16

Suppose that `miles` is a variable of type `double`. Further suppose that the input is `73.65`. Consider the following statements:

```
cin >> miles;
```

This statement causes the computer to get the input, which is `73.65`, from the standard input device and stores it in the variable `miles`. That is, after this statement executes, the value of the variable `miles` is `73.65`.

Example 2-17 further explains how to input numeric data into a program.

EXAMPLE 2-17

Suppose we have the following statements:

```
int feet;  
int inches;
```

Suppose the input is:

```
23 7
```

Next, consider the following statement:

```
cin >> feet >> inches;
```

This statement first stores the number `23` into the variable `feet` and then the number `7` into the variable `inches`. Notice that when these numbers are entered via the keyboard, they are separated with a blank. In fact, they can be separated with one or more blanks or lines or even the tab character.

The following C++ program shows the effect of the preceding input statements:

```
// This program illustrates how input statements work.

#include <iostream>

using namespace std;

int main()
{
    int feet;
    int inches;

    cout << "Enter two integers separated by spaces: ";
    cin >> feet >> inches;
    cout << endl;

    cout << "Feet = " << feet << endl;
    cout << "Inches = " << inches << endl;

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

Enter two integers separated by spaces: 23 7

Feet = 23
Inches = 7

The C++ program in Example 2-18 illustrates how to read strings and numeric data.

EXAMPLE 2-18

```
// This program illustrates how to read strings and numeric data.

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string firstName;           //Line 1
    string lastName;           //Line 2
    int age;                    //Line 3
    double weight;              //Line 4

    cout << "Enter first name, last name, age, "
         << "and weight, separated by spaces."
         << endl;               //Line 5
}
```

```

    cin >> firstName >> lastName;           //Line 6
    cin >> age >> weight;                     //Line 7

    cout << "Name: " << firstName << " "
         << lastName << endl;               //Line 8

    cout << "Age: " << age << endl;          //Line 9
    cout << "Weight: " << weight << endl;    //Line 10

    return 0;                               //Line 11
}

```

2

Sample Run: In this sample run, the user input is shaded.

Enter first name, last name, age, and weight, separated by spaces.

Sheila Mann 23 120.5

Name: Sheila Mann

Age: 23

Weight: 120.5

The preceding program works as follows: The statements in Lines 1 to 4 declare the variables `firstName` and `lastName` of type `string`, `age` of type `int`, and `weight` of type `double`. The statement in Line 5 is an output statement and tells the user what to do. (Such output statements are called prompt lines.) As shown in the sample run, the input to the program is:

Sheila Mann 23 120.5

The statement in Line 6 first reads and stores the string `Sheila` into the variable `firstName` and then skips the space after `Sheila` and reads and stores the string `Mann` into the variable `lastName`. Next, the statement in Line 7 first skips the blank after `Mann` and reads and stores `23` into the variable `age` and then skips the blank after `23` and reads and stores `120.5` into the variable `weight`.

The statements in Lines 8, 9, and 10 produce the third, fourth, and fifth lines of the sample run.

NOTE

During programming execution, if more than one value is entered in a line, these values must be separated by at least one blank or tab. Alternately, one value per line can be entered.

Variable Initialization

Remember, there are two ways to initialize a variable: by using the assignment statement and by using a read statement. Consider the following declaration:

```

int feet;
int inches;

```

Consider the following two sets of code:

```
(a) feet = 35;
    inches = 6;
    cout << "Total inches = "
        << 12 * feet + inches;

(b) cout << "Enter feet: ";
    cin >> feet;
    cout << endl;
    cout << "Enter inches: ";
    cin >> inches;
    cout << endl;
    cout << "Total inches = "
        << 12 * feet + inches;
```

In (a), **feet** and **inches** are initialized using assignment statements, and in (b), these variables are initialized using input statements. However, each time the code in (a) executes, **feet** and **inches** are initialized to the same value unless you edit the source code, change the value, recompile, and run. On the other hand, in (b), each time the program runs, you are prompted to enter values for **feet** and **inches**. Therefore, a read statement is much more versatile than an assignment statement.

Sometimes it is necessary to initialize a variable by using an assignment statement. This is especially true if the variable is used only for internal calculation and not for reading and storing data.

Recall that C++ does not automatically initialize variables when they are declared. Some variables can be initialized when they are declared, whereas others must be initialized using either an assignment statement or a read statement.

NOTE

When the program is compiled, some of the newer IDEs might give warning messages if the program uses the value of a variable without first properly initializing that variable. In this case, if you ignore the warning and execute the program, the program might terminate abnormally with an error message.

NOTE

Suppose you want to store a character into a **char** variable using an input statement. During program execution, when you enter the character, you do not include the single quotes. For example, suppose that **ch** is a **char** variable. Consider the following input statement:

```
cin >> ch;
```

If you want to store **K** into **ch** using this statement, during program execution, you only enter **K**. Similarly, if you want to store a string into a **string** variable using an input statement, during program execution, you enter only the string without the double quotes.

EXAMPLE 2-19

This example further illustrates how assignment statements and input statements manipulate variables. Consider the following declarations:

```
int firstNum, secondNum;
double z;
char ch;
string name;
```

Also, suppose that the following statements execute in the order given:

1. `firstNum = 4;`
2. `secondNum = 2 * firstNum + 6;`
3. `z = (firstNum + 1) / 2.0;`
4. `ch = 'A';`
5. `cin >> secondNum;`
6. `cin >> z;`
7. `firstNum = 2 * secondNum + static_cast<int>(z);`
8. `cin >> name;`
9. `secondNum = secondNum + 1;`
10. `cin >> ch;`
11. `firstNum = firstNum + static_cast<int>(ch);`
12. `z = firstNum - z;`

In addition, suppose the input is:

8 16.3 Jenny D

This line has four values, 8, 16.3, Jenny, and D, and each value is separated from the others by a blank.

Let's now determine the values of the declared variables after the last statement executes. To explicitly show how a particular statement changes the value of a variable, the values of the variables after each statement executes are shown. (In the following figures, a question mark [?] in a box indicates that the value in the box is unknown.)

Before statement 1 executes, all variables are uninitialized, as shown in Figure 2-2.

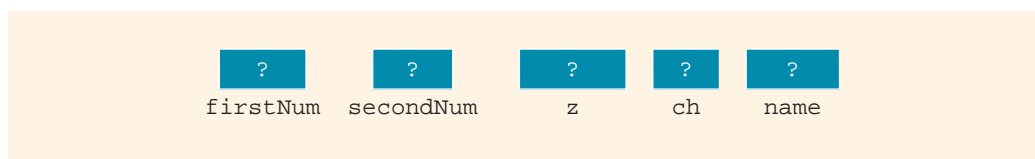


FIGURE 2-2 Variables before statement 1 executes

Next, we show the values of the variables after the execution of each statement.

After St.	Values of the Variables	Explanation
1	<div> <div>4</div> <div>?</div> <div>?</div> <div>?</div> <div>?</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	Store 4 into firstNum .
2	<div> <div>4</div> <div>14</div> <div>?</div> <div>?</div> <div>?</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	$2 * \text{firstNum} + 6 = 2 * 4 + 6 = 14$. Store 14 into secondNum .
3	<div> <div>4</div> <div>14</div> <div>2.5</div> <div>?</div> <div>?</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	$(\text{firstNum} + 1) / 2.0 = (4 + 1) / 2.0 = 5 / 2.0 = 2.5$. Store 2.5 into z .
4	<div> <div>4</div> <div>14</div> <div>2.5</div> <div>A</div> <div>?</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	Store 'A' into ch .
5	<div> <div>4</div> <div>8</div> <div>2.5</div> <div>A</div> <div>?</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	Read a number from the keyboard (which is 8) and store it into secondNum . This statement replaces the old value of secondNum with this new value.
6	<div> <div>4</div> <div>8</div> <div>16.3</div> <div>A</div> <div>?</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	Read a number from the keyboard (which is 16.3) and store this number into z . This statement replaces the old value of z with this new value.
7	<div> <div>32</div> <div>8</div> <div>16.3</div> <div>A</div> <div>?</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	$2 * \text{secondNum} + \text{static_cast<int>}(z) = 2 * 8 + \text{static_cast<int>}(16.3) = 16 + 16 = 32$. Store 32 into firstNum . This statement replaces the old value of firstNum with this new value.
8	<div> <div>32</div> <div>8</div> <div>16.3</div> <div>A</div> <div>Jenny</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	Read the next input, Jenny , from the keyboard and store it into name .
9	<div> <div>32</div> <div>9</div> <div>16.3</div> <div>A</div> <div>Jenny</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	$\text{secondNum} + 1 = 8 + 1 = 9$. Store 9 into secondNum .
10	<div> <div>32</div> <div>9</div> <div>16.3</div> <div>D</div> <div>Jenny</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	Read the next input from the keyboard (which is D) and store it into ch . This statement replaces the old value of ch with the new value.

After St.	Values of the Variables	Explanation
11	<div> <div>100</div> <div>9</div> <div>16.3</div> <div>D</div> <div>Jenny</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	<code>firstNum + static_cast<int>(ch) = 32 + static_cast<int>('D') = 32 + 68 = 100.</code> Store 100 into <code>firstNum</code> .
12	<div> <div>100</div> <div>9</div> <div>83.7</div> <div>D</div> <div>Jenny</div> </div> <div> <div>firstNum</div> <div>secondNum</div> <div>z</div> <div>ch</div> <div>name</div> </div>	<code>firstNum - z = 100 - 16.3 = 100.0 - 16.3 = 83.7.</code> Store 83.7 into <code>z</code> .

NOTE

When something goes wrong in a program and the results it generates are not what you expected, you should do a walk-through of the statements that assign values to your variables. Example 2-19 illustrates how to do a walk-through of your program. This is a very effective debugging technique. The Web site accompanying this book contains a C++ program that shows the effect of the 12 statements listed at the beginning of Example 2-19. The program is named `Example_2_19.cpp`.

NOTE

If you assign the value of an expression that evaluates to a floating-point value—without using the cast operator—to a variable of type `int`, the fractional part is dropped. In this case, the compiler most likely will issue a warning message about the implicit type conversion.

Increment and Decrement Operators

Now that you know how to declare a variable and enter data into a variable, in this section, you will learn about two more operators: the **increment operators**. These operators are used frequently by C++ programmers and are useful programming tools.

Suppose `count` is an `int` variable. The statement:

```
count = count + 1;
```

increments the value of `count` by 1. To execute this assignment statement, the computer first evaluates the expression on the right, which is `count + 1`. It then assigns this value to the variable on the left, which is `count`.

As you will see in later chapters, such statements are frequently used to keep track of how many times certain things have happened. To expedite the execution of such statements, C++ provides the **increment operator**, `++`, which increases the value of a variable by 1, and the **decrement operator**, `--`, which decreases the value of a variable by 1.

Increment and decrement operators each have two forms, pre and post. The syntax of the increment operator is:

Pre-increment: `++variable`

Post-increment: `variable++`

The syntax of the decrement operator is:

Pre-decrement: `--variable`

Post-decrement: `variable--`

Let's look at some examples. The statement:

```
++count;
```

or:

```
count++;
```

increments the value of `count` by 1. Similarly, the statement:

```
--count;
```

or:

```
count--;
```

decrements the value of `count` by 1.

Because both the increment and decrement operators are built into C++, the value of the variable is quickly incremented or decremented without having to use the form of an assignment statement.

Now, both the pre- and post-increment operators increment the value of the variable by 1. Similarly, the pre- and post-decrement operators decrement the value of the variable by 1. What is the difference between the pre and post forms of these operators? The difference becomes apparent when the variable using these operators is employed in an expression.

Suppose that `x` is an `int` variable. If `++x` is used in an expression, first the value of `x` is incremented by 1, and then the new value of `x` is used to evaluate the expression. On the other hand, if `x++` is used in an expression, first the current value of `x` is used in the expression, and then the value of `x` is incremented by 1. The following example clarifies the difference between the pre- and post-increment operators.

Suppose that `x` and `y` are `int` variables. Consider the following statements:

```
x = 5;
y = ++x;
```

The first statement assigns the value 5 to `x`. To evaluate the second statement, which uses the pre-increment operator, first the value of `x` is incremented to 6, and then this value, 6, is assigned to `y`. After the second statement executes, both `x` and `y` have the value 6.

Now, consider the following statements:

```
x = 5;  
y = x++;
```

As before, the first statement assigns 5 to **x**. In the second statement, the post-increment operator is applied to **x**. To execute the second statement, first the value of **x**, which is 5, is used to evaluate the expression, and then the value of **x** is incremented to 6. Finally, the value of the expression, which is 5, is stored in **y**. After the second statement executes, the value of **x** is 6, and the value of **y** is 5.

The following example further illustrates how the pre and post forms of the increment operator work.

EXAMPLE 2-20

Suppose **a** and **b** are `int` variables and:

```
a = 5;  
b = 2 + (++a);
```

The first statement assigns 5 to **a**. To execute the second statement, first the expression `2 + (++a)` is evaluated. Because the pre-increment operator is applied to **a**, first the value of **a** is incremented to 6. Then 2 is added to 6 to get 8, which is then assigned to **b**. Therefore, after the second statement executes, **a** is 6 and **b** is 8.

On the other hand, after the execution of the following statements:

```
a = 5;  
b = 2 + (a++);
```

the value of **a** is 6 while the value of **b** is 7.

This book will most often use the increment and decrement operators with a variable in a stand-alone statement. That is, the variable using the increment or decrement operator will not be part of any expression.

Output

In the preceding sections, you have seen how to put data into the computer's memory and how to manipulate that data. We also used certain output statements to show the results on the *standard output device*. This section explains in some detail how to further use output statements to generate the desired results.

NOTE

The standard output device is usually the screen.

In C++, output on the standard output device is accomplished via the use of `cout` and the operator `<<`. The general syntax of `cout` together with `<<` is:

```
cout << expression or manipulator << expression or manipulator...;
```

This is called an **output statement**. In C++, `<<` is called the **stream insertion operator**. Generating output with `cout` follows two rules:

1. The expression is evaluated, and its value is printed at the current insertion point on the output device.
2. A manipulator is used to format the output. The simplest manipulator is `endl` (the last character is the letter `l`), which causes the insertion point to move to the beginning of the next line.

NOTE

On the screen, the insertion point is where the cursor is.

The next example illustrates how an output statement works. In an output statement, a string or an expression involving only one variable or a single value evaluates to itself.

NOTE

When an output statement outputs `char` values, it outputs only the character without the single quotes (unless the single quotes are part of the output statement).

For example, suppose `ch` is a `char` variable and `ch = 'A'`; . The statement:

```
cout << ch;
```

or:

```
cout << 'A';
```

outputs:

A

Similarly, when an output statement outputs the value of a string, it outputs only the string without the double quotes (unless you include double quotes as part of the output).

EXAMPLE 2-21

Consider the following statements. The output is shown to the right of each statement.

Statement	Output
1 <code>cout << 29 / 4 << endl;</code>	7
2 <code>cout << "Hello there." << endl;</code>	Hello there.
3 <code>cout << 12 << endl;</code>	12
4 <code>cout << "4 + 7" << endl;</code>	4 + 7

```

5 cout << 4 + 7 << endl;           11
6 cout << 'A' << endl;             A
7 cout << "4 + 7 = " << 4 + 7 << endl; 4 + 7 = 11
8 cout << 2 + 3 * 5 << endl;        17
9 cout << "Hello \nthere." << endl; Hello
                                   there.

```

Look at the output of statement 9. Recall that in C++, the newline character is `'\n'`; it causes the insertion point to move to the beginning of the next line before printing there. Therefore, when `\n` appears in a string in an output statement, it causes the insertion point to move to the beginning of the next line on the output device. This fact explains why **Hello** and **there.** are printed on separate lines.

NOTE

In C++, `\` is called the escape character and `\n` is called newline escape sequence.

Recall that all variables must be properly initialized; otherwise, the value stored in them may not make much sense. Also recall that C++ does not automatically initialize variables.

If `num` is an `int` variable, then the output of the C++ statement:

```
cout << num << endl;
```

is meaningful provided that `num` has been given a value. For example, the sequence of C++ statements:

```
num = 45;
cout << num << endl;
```

will produce the output 45.

EXAMPLE 2-22

Consider the following C++ program.

```

// This program illustrates how output statements work.

#include <iostream>

using namespace std;

int main()
{
    int a, b;

    a = 65;           //Line 1
    b = 78;           //Line 2

```

```

    cout << 29 / 4 << endl;           //Line 3
    cout << 3.0 / 2 << endl;         //Line 4
    cout << "Hello there.\n";        //Line 5
    cout << 7 << endl;               //Line 6
    cout << 3 + 5 << endl;           //Line 7
    cout << "3 + 5";                 //Line 8
    cout << endl;                   //Line 9
    cout << a << endl;               //Line 10
    cout << "a" << endl;            //Line 11
    cout << (a + 5) * 6 << endl;     //Line 12
    cout << 2 * b << endl;          //Line 13

    return 0;
}

```

In the following output, the column marked “Output of Statement at” and the line numbers are not part of the output. The line numbers are shown in this column to make it easy to see which output corresponds to which statement.

	Output of Statement at
7	Line 3
1.5	Line 4
Hello there.	Line 5
7	Line 6
8	Line 7
3 + 5	Line 8
65	Line 10
a	Line 11
420	Line 12
156	Line 13

For the most part, the output is straightforward. Look at the output of the statements in Lines 7, 8, 9, and 10. The statement in Line 7 outputs the result of $3 + 5$, which is 8, and moves the insertion point to the beginning of the next line. The statement in Line 8 outputs the string $3 + 5$. Note that the statement in Line 8 consists only of the string $3 + 5$. Therefore, after printing $3 + 5$, the insertion point stays positioned after 5; it does not move to the beginning of the next line.

The output statement in Line 9 contains only the manipulator `endl`, which moves the insertion point to the beginning of the next line. Therefore, when the statement in Line 10 executes, the output starts at the beginning of the line. Note that in this output, the column “Output of Statement at” does not contain Line 9. This is due to the fact that the statement in Line 9 does not produce any printable output. It simply moves the insertion point to the beginning of the next line. Next, the statement in Line 10 outputs the value of `a`, which is 65. The manipulator `endl` then moves the insertion point to the beginning of the next line.

NOTE

Outputting or accessing the value of a variable in an expression does not destroy or modify the contents of the variable.

2

Let us now take a close look at the newline character, '\n'. Consider the following C++ statements:

```
cout << "Hello there.";
cout << "My name is James.";
```

If these statements are executed in sequence, the output is:

```
Hello there.My name is James.
```

Now consider the following C++ statements:

```
cout << "Hello there.\n";
cout << "My name is James.";
```

The output of these C++ statements is:

```
Hello there.
My name is James.
```

When \n is encountered in the string, the insertion point is positioned at the beginning of the next line. Note also that \n may appear anywhere in the string. For example, the output of the statement:

```
cout << "Hello \nthere. \nMy name is James.";
```

is:

```
Hello
there.
My name is James.
```

Also, note that the output of the statement:

```
cout << '\n';
```

is the same as the output of the statement:

```
cout << "\n";
```

which is equivalent to the output of the statement:

```
cout << endl;
```

Thus, the output of the sequence of statements:

```
cout << "Hello there.\n";
cout << "My name is James.";
```

is equivalent to the output of the sequence of statements:

```
cout << "Hello there." << endl;
cout << "My name is James.";
```

EXAMPLE 2-23

Consider the following C++ statements:

```
cout << "Hello there.\nMy name is James.";
```

or:

```
cout << "Hello there.";
```

```
cout << "\nMy name is James.";
```

or:

```
cout << "Hello there.";
```

```
cout << endl << "My name is James.";
```

In each case, the output of the statements is:

```
Hello there.
My name is James.
```

EXAMPLE 2-24

The output of the C++ statements:

```
cout << "Count...\n....1\n.....2\n.....3";
```

or:

```
cout << "Count..." << endl << "....1" << endl
      << ".....2" << endl << ".....3";
```

is:

```
Count...
....1
.....2
.....3
```

EXAMPLE 2-25

Suppose that you want to output the following sentence in one line as part of a message:

```
It is sunny, warm, and not a windy day. We can go golfing.
```

Obviously, you will use an output statement to produce this output. However, in the programming code, this statement may not fit in one line as part of the output statement. Of course, you can use multiple output statements as follows:

```
cout << "It is sunny, warm, and not a windy day. ";
cout << "We can go golfing." << endl;
```

Note the semicolon at the end of the first statement and the identifier `cout` at the beginning of the second statement. Also, note that there is no manipulator `endl` at the end of the first statement. Here, two output statements are used to output the sentence in one line. Equivalently, you can use the following output statement to output this sentence:

```
cout << "It is sunny, warm, and not a windy day. "
      << "We can go golfing." << endl;
```

In this statement, note that there is no semicolon at the end of the first line, and the identifier `cout` does not appear at the beginning of the second line. Because there is no semicolon at the end of the first line, this output statement continues at the second line. Also, note the double quotation marks at the beginning and end of the sentences on each line. The string is broken into two strings, but both strings are part of the same output statement.

If a string appearing in an output statement is long and you want to output the string in one line, you can break the string by using either of the previous two methods. However, the following statement would be incorrect:

```
cout << "It is sunny, warm, and not a windy day.
      We can go golfing." << endl; //illegal
```

In other words, the return (or Enter) key on your keyboard cannot be part of the string. That is, in programming code, a string *cannot* be broken into more than one line by using the return (Enter) key on your keyboard.

Recall that the newline character is `\n`, which causes the insertion point to move to the beginning of the next line. There are many escape sequences in C++, which allow you to control the output. Table 2-4 lists some of the commonly used escape sequences.

TABLE 2-4 Commonly Used Escape Sequences

	Escape Sequence	Description
<code>\n</code>	Newline	Cursor moves to the beginning of the next line
<code>\t</code>	Tab	Cursor moves to the next tab stop
<code>\b</code>	Backspace	Cursor moves one space to the left
<code>\r</code>	Return	Cursor moves to the beginning of the current line (not the next line)
<code>\\</code>	Backslash	Backslash is printed
<code>\'</code>	Single quotation	Single quotation mark is printed
<code>\"</code>	Double quotation	Double quotation mark is printed

The following example shows the effect of some of these escape sequences.

EXAMPLE 2-26

The output of the statement:

```
cout << "The newline escape sequence is \\n" << endl;
```

is:

```
The newline escape sequence is \n
```

The output of the statement:

```
cout << "The tab character is represented as '\\t'" << endl;
```

is:

```
The tab character is represented as '\t'
```

Note that the single quote can also be printed without using the escape sequence. Therefore, the preceding statement is equivalent to the following output statement:

```
cout << "The tab character is represented as '\\t'" << endl;
```

The output of the statement:

```
cout << "The string \"Sunny\" contains five characters." << endl;
```

is:

```
The string "Sunny" contains five characters.
```

NOTE

The Web site accompanying this text contains the C++ program that shows the effect of the statements in Example 2-26. The program is named `Example2_26.cpp`.

To use `cin` and `cout` in a program, you must include a certain header file. The next section explains what this header file is, how to include a header file in a program, and why you need header files in a program. Chapter 3 will provide a full explanation of `cin` and `cout`.

Preprocessor Directives

Only a small number of operations, such as arithmetic and assignment operations, are explicitly defined in C++. Many of the functions and symbols needed to run a C++ program are provided as a collection of libraries. Every library has a name and is referred to by a header file. For example, the descriptions of the functions needed to perform input/output (I/O) are contained in the header file `iostream`. Similarly, the descriptions of some very useful mathematical functions, such as power, absolute, and sine, are contained in the header file `cmath`. If you want to use I/O or math functions, you need to tell the computer where to find the necessary code. You use preprocessor directives and the names of header files to tell the computer the locations of the code provided in libraries. Preprocessor directives are processed by a program called a **preprocessor**.

Preprocessor directives are commands supplied to the preprocessor that cause the preprocessor to modify the text of a C++ program before it is compiled. All preprocessor commands begin with `#`. There are no semicolons at the end of preprocessor commands because they are not C++ statements. To use a header file in a C++ program, use the preprocessor directive `include`.

The general syntax to include a header file (provided by the IDE) in a C++ program is:

```
#include <headerFileName>
```

For example, the following statement includes the header file `iostream` in a C++ program:

```
#include <iostream>
```

Preprocessor directives to include header files are placed as the first line of a program so that the identifiers declared in those header files can be used throughout the program. (Recall that in C++, identifiers must be declared before they can be used.)

Certain header files are required to be provided as part of C++. Appendix F describes some of the commonly used header files. Individual programmers can also create their own header files, which is discussed in the chapter *Classes and Data Abstraction*, later in this book.

Note that the preprocessor commands are processed by the preprocessor before the program goes through the compiler.

From Figure 1-3 (Chapter 1), we can conclude that a C++ system has three basic components: the program development environment, the C++ language, and the C++ library. All three components are integral parts of the C++ system. The program development environment consists of the six steps shown in Figure 1-3. As you learn the C++ language throughout the book, we will discuss components of the C++ library as we need them.

namespace and Using cin and cout in a Program

Earlier, you learned that both `cin` and `cout` are predefined identifiers. In ANSI/ISO Standard C++, these identifiers are declared in the header file `iostream`, but within a `namespace`. The name of this `namespace` is `std`. (The `namespace` mechanism will be formally defined and discussed in detail in Chapter 8. For now, you need to know only how to use `cin` and `cout` and, in fact, any other identifier from the header file `iostream`.)

There are several ways you can use an identifier declared in the namespace `std`. One way to use `cin` and `cout` is to refer to them as `std::cin` and `std::cout` throughout the program.

Another option is to include the following statement in your program:

```
using namespace std;
```

This statement appears after the statement `#include <iostream>`. You can then refer to `cin` and `cout` without using the prefix `std::`. To simplify the use of `cin` and `cout`, this book uses the second form. That is, to use `cin` and `cout` in a program, the programs will contain the following two statements:

```
#include <iostream>
```

```
using namespace std;
```

In C++, `namespace` and `using` are reserved words.

The `namespace` mechanism is a feature of ANSI/ISO Standard C++. As you learn more C++ programming, you will become aware of other header files. For example, the header file `cmath` contains the specifications of many useful mathematical functions. Similarly, the header file `iomanip` contains the specifications of many useful functions and manipulators that help you format your output in a specific manner. However, just like the identifiers in the header file `iostream`, the identifiers in ANSI/ISO Standard C++ header files are declared within a `namespace`.

The name of the `namespace` in each of these header files is `std`. Therefore, whenever certain features of a header file in ANSI/ISO Standard C++ are discussed, this book will refer to the identifiers without the prefix `std::`. Moreover, to simplify the accessing of identifiers in programs, the statement `using namespace std;` will be included. Also, if a program uses multiple header files, only one `using` statement is needed. This `using` statement typically appears after all the header files.

Using the string Data Type in a Program

Recall that the `string` data type is a programmer-defined data type and is not directly available for use in a program. To use the `string` data type, you need to access its definition from the header file `string`. Therefore, to use the `string` data type in a program, you must include the following preprocessor directive:

```
#include <string>
```

Creating a C++ Program

In previous sections, you learned enough C++ concepts to write meaningful programs. You are now ready to create a complete C++ program.

A C++ program is a collection of functions, one of which is the function `main`. Therefore, if a C++ program consists of only one function, then it must be the function `main`. Moreover, a function is a set of instructions designed to accomplish a specific task. Until Chapter 6, you will deal mainly with the function `main`.

The statements to declare variables, the statements to manipulate data (such as assignments), and the statements to input and output data are placed within the function `main`. The statements to declare named constants are usually placed outside of the function `main`.

The syntax of the function `main` used throughout this book has the following form:

```
int main()
{
    statement_1
    .
    .
    .
    statement_n

    return 0;
}
```

In the syntax of the function `main`, each statement (`statement_1, ..., statement_n`) is usually either a declarative statement or an executable statement. The statement `return 0;` must be included in the function `main` and must be the last statement. If the statement `return 0;` is misplaced in the body of the function `main`, the results generated by the program may not be to your liking. The meaning of the statement `return 0;` will be discussed in Chapter 6. In C++, `return` is a reserved word.

A C++ program might use the resources provided by the IDE, such as the necessary code to input the data, which would require your program to include certain header files. You can, therefore, divide a C++ program into two parts: preprocessor directives and the program. The preprocessor directives tell the compiler which header files to include in the program. The program contains statements that accomplish meaningful results. Taken together, the preprocessor directives and the program statements constitute the C++ **source code**. Recall that to be useful, source code must be saved in a file with the file extension `.cpp`. For example, if the source code is saved in the file `firstProgram`, then the complete name of this file is `firstProgram.cpp`. The file containing the source code is called the **source code file** or **source file**.

When the program is compiled, the compiler generates the object code, which is saved in a file with the file extension `.obj`. When the object code is linked with the system resources, the executable code is produced and saved in a file with the file extension `.exe`. Typically, the name of the file containing the object code and the name of the file containing the executable code are the same as the name of the file containing the source

code. For example, if the source code is located in a file named `firstProg.cpp`, the name of the file containing the object code is `firstProg.obj`, and the name of the file containing the executable code is `firstProg.exe`.

The extensions as given in the preceding paragraph—that is, `.cpp`, `.obj`, and `.exe`—are system dependent. Moreover, some IDEs maintain programs in the form of projects. The name of the project and the name of the source file need not be the same. It is possible that the name of the executable file is the name of the project, with the extension `.exe`. To be certain, check your system or IDE documentation.

Because the programming instructions are placed in the function `main`, let us elaborate on this function.

The basic parts of the function `main` are the heading and the body. The first line of the function `main`, that is:

```
int main()
```

is called the heading of the function `main`.

The statements enclosed between the curly braces (`{` and `}`) form the body of the function `main`. The body of the function `main` contains two types of statements:

- Declaration statements
- Executable statements

Declaration statements are used to declare things, such as variables.

In C++, variables or identifiers can be declared anywhere in the program, but they must be declared before they can be used.

EXAMPLE 2-27

The following statements are examples of variable declarations:

```
int a, b, c;
double x, y;
```

Executable statements perform calculations, manipulate data, create output, accept input, and so on.

Some executable statements that you have encountered so far are the assignment, input, and output statements.

EXAMPLE 2-28

The following statements are examples of executable statements:

```
a = 4;                //assignment statement
cin >> b;             //input statement
cout << a << " " << b << endl; //output statement
```


In skeleton form, a C++ program looks like the following:

```
//comments, if needed

preprocessor directives to include header files

using statement

named constants, if necessary

int main()
{
    statement_1
        .
        .
        .
    statement_n

    return 0;
}
```

The C++ program in Example 2-29 shows where include statements, declaration statements, executable statements, and so on typically appear in the program.

EXAMPLE 2-29

```

//*****
// Author: D.S. Malik
//
// This program shows where the include statements, using
// statement, named constants, variable declarations, assignment
// statements, and input and output statements typically appear.
//*****

#include <iostream>                                //Line 1

using namespace std;                               //Line 2

const int NUMBER = 12;                             //Line 3

int main()                                          //Line 4
{                                                  //Line 5
    int firstNum;                                  //Line 6
    int secondNum;                                 //Line 7

    firstNum = 18;                                  //Line 8
    cout << "Line 9: firstNum = " << firstNum
        << endl;                                   //Line 9

    cout << "Line 10: Enter an integer: ";         //Line 10
    cin >> secondNum;                               //Line 11
    cout << endl;                                   //Line 12
}
```

```

    cout << "Line 13: secondNum = " << secondNum
        << endl;                                //Line 13

    firstNum = firstNum + NUMBER + 2 * secondNum; //Line 14

    cout << "Line 15: The new value of "
        << "firstNum = " << firstNum << endl;    //Line 15

    return 0;                                    //Line 16
}                                                 //Line 17

```

Sample Run: In this sample run, the user input is shaded.

Line 9: `firstNum = 18`

Line 10: Enter an integer: **15**

Line 13: `secondNum = 15`

Line 15: The new value of `firstNum = 60`

The preceding program works as follows: The statement in Line 1 includes the header file `iostream` so that program can perform input/output. The statement in Line 2 uses the `using namespace` statement so that identifiers declared in the header file `iostream`, such as `cin`, `cout`, and `endl`, can be used without using the prefix `std::`. The statement in Line 3 declares the named constant `NUMBER` and sets its value to 12. The statement in Line 4 contains the heading of the function `main`, and the left brace in Line 5 marks the beginning of the function `main`. The statements in Lines 6 and 7 declare the variables `firstNum` and `secondNum`.

The statement in Line 8 sets the value of `firstNum` to 18, and the statement in Line 9 outputs the value of `firstNum`. Next, the statement in Line 10 prompts the user to enter an integer. The statement in Line 11 reads and stores the integer into the variable `secondNum`, which is 15 in the sample run. The statement in Line 12 positions the cursor on the screen at the beginning of the next line. The statement in Line 13 outputs the value of `secondNum`. The statement in Line 14 evaluates the expression:

```
firstNum + NUMBER + 2 * secondNum
```

and assigns the value of this expression to the variable `firstNum`, which is 60 in the sample run. The statement in Line 15 outputs the new value of `firstNum`. The statement in Line 16 contains the `return` statement. The right brace in Line 17 marks the end of the function `main`.

Debugging: Understanding and Fixing Syntax Errors

The previous sections of this chapter described the basic components of a C++ program. When you type a program, typos and unintentional syntax errors are likely to occur. Therefore, when you compile a program, the compiler will identify the syntax error. In this section, we show how to identify and fix syntax errors.

Consider the following C++ program:

```

1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int num
8.
9.      num = 18;
10.
11.     tempNum = 2 * num;
12.
13.     cout << "Num = " << num << ", tempNum = " < tempNum << endl;
14.
15.     return ;
16. }
```

(Note that the numbers 1 to 16 on the left side are not part of the program. We have numbered the statements for easy references.) This program contains syntax errors. When you compile this program, the compiler produces the following errors: (This program is compiled using Visual C++ Express 2008.)

```

Example2_Syntax_Errors.cpp
c:\chapter 2 source code\example2_syntax_errors.cpp(9) : error C2146: syntax error :
missing ';' before identifier 'num'
c:\chapter 2 source code\example2_syntax_errors.cpp(11) : error C2065: 'tempNum' :
undeclared identifier
c:\chapter 2 source code\example2_syntax_errors.cpp(13) : error C2065: 'tempNum' :
undeclared identifier
c:\chapter 2 source code\example2_syntax_errors.cpp(13) : error C2563: mismatch in formal
parameter list
c:\chapter 2 source code\example2_syntax_errors.cpp(13) : error C2568: '<<' : unable to
resolve function overload
      c:\program files\microsoft visual studio 9.0\vc\include\ostream(974): could be
'std::basic_ostream<_Elem,_Traits> &std::endl(std::basic_ostream<_Elem,_Traits> &)'
      with
      [
          _Elem=wchar_t,
          _Traits=std::char_traits<wchar_t>
      ]
      c:\program files\microsoft visual studio 9.0\vc\include\ostream(966): or
'std::basic_ostream<_Elem,_Traits> &std::endl(std::basic_ostream<_Elem,_Traits> &)'
      with
      [
          _Elem=char,
          _Traits=std::char_traits<char>
      ]
]
```

```

c:\program files\microsoft visual studio 9.0\vc\include\ostream(940): or
'std::basic_ostream<_Elem,_Traits> &std::endl(std::basic_ostream<_Elem,_Traits> &)'
c:\chapter 2 source code\example2_syntax_errors.cpp(15) : error C2561: 'main' : function
must return a value

f:\cs1 c++ fifth edition\chapter 2\chapter 2 source code and prog ex\chapter 2
source code\example2_syntax_errors.cpp(5) : see declaration of 'main'
Build log was saved at "file:///c:/Documents and Settings\DM\My
Documents\Proj1\Proj1\Debug\BuildLog.htm"
Proj1 - 6 error(s), 0 warning(s)
===== Rebuild All: 0 succeeded, 1 failed, 0 skipped =====

```

First, consider the following error:

```

c:\chapter 2 source code\example2_syntax_errors.cpp(9) : error C2146:
syntax error : missing ';' before identifier 'num'

```

The expression `example2_syntax_errors.cpp(9)` indicates that there is an error in Line 9. The remaining part of this error specifies that there is a missing `;` before the identifier `num`. If we look at Line 7, we find that there is a missing semicolon at the end of the statement `int num`. Therefore, we must insert `;` at the end of the statement in Line 7.

Next, consider the second error:

```

c:\chapter 2 source code\example2_syntax_errors.cpp(11) : error C2065: 'tempNum' :
undeclared identifier

```

This error occurs in Line 11, and it specifies that the identifier `tempNum` is undeclared. When we look at the code, we find that this identifier has not been declared. So we must declare `tempNum` as an `int` variable.

The error:

```

c:\chapter 2 source code\example2_syntax_errors.cpp(13) : error C2065: 'tempNum' :
undeclared identifier

```

occurs in Line 13, and it specifies that the identifier `tempNum` is undeclared. As in the previous error, we must declare `tempNum`. Note that once we declare `tempNum` and recompile, this and the previous error will disappear.

The next error is:

```

c:\chapter 2 source code\example2_syntax_errors.cpp(13) : error C2563: mismatch
in formal parameter list

```

This error occurs in Line 13, and it indicates that some formal parameter list is mismatched. For a beginner, this error is somewhat hard to understand. (In Chapter 15, we will explain the formal parameter list of the operator `<<`.) However, as you practice, you will learn how to interpret and correct syntax errors. This error becomes clear if you look at the next error, the part of which is:

```

c:\chapter 2 source code\example2_syntax_errors.cpp(13) : error C2568: '<<' :
unable to resolve function overload

```

It tells us that this error has something to do with the operator <<. When we carefully look at the statement in Line 13, which is:

```
cout << "Num = " << num << ", tempNum = " < tempNum << endl;
```

we find that in the expression < tempNum, we have unintentionally used < in place of <<. So we must correct this error.

Let us look at the last error, which is:

```
c:\chapter 2 source code\example2_syntax_errors.cpp(15) : error C2561: 'main' :  
function must return a value  
c:\chapter 2 source code\example2_syntax_errors.cpp(5) : see declaration  
of 'main'
```

This error occurs in Line 15. However, at this point, the explanation given, especially for a beginner, is somewhat unclear. However, if you look at the statement `return ;` in Line 15 and remember the syntax of the function `main` as well as all the programs given in this book, we find that the number 0 is missing, that is, this statement must be `return 0;`

From the errors reported by the compiler, we see that the compiler not only identifies the errors, but it also specifies the line numbers where the errors occur and the types of the errors. We can effectively use this information to fix syntax error.

After correcting all of the syntax errors, a correct program is:

```
#include <iostream>

using namespace std;

int main()
{
    int num;
    int tempNum;

    num = 18;

    tempNum = 2 * num;

    cout << "Num = " << num << ", tempNum = " << tempNum << endl;

    return 0;
}
```

The output is:

```
Num = 18, tempNum = 36
```

As you learn C++ and practice writing and executing programs, you will learn how to spot and fix syntax errors. It is possible that the list of errors reported by the compiler is longer than the program itself. This is because a syntax error in one line can cause syntax errors in subsequent lines. In situations like this, correct the syntax errors in the order they

are listed and compile your program, if necessary, after each correction. You will see how quickly the syntax errors list shrinks. The important thing is not to panic.

In the next section, we describe some simple rules that you can follow so that your program is properly structured.

Program Style and Form

In previous sections, you learned enough C++ concepts to write meaningful programs. Before beginning to write programs, however, you need to learn their proper structure, among other things. Using the proper structure for a C++ program makes it easier to understand and subsequently modify the program. There is nothing more frustrating than trying to follow and perhaps modify a program that is syntactically correct but has no structure.

In addition, every C++ program must satisfy certain rules of the language. A C++ program must contain the function `main`. It must also follow the syntax rules, which, like grammar rules, tell what is right and what is wrong and what is legal and what is illegal in the language. Other rules serve the purpose of giving precise meaning to the language; that is, they support the language's semantics.

The following sections are designed to help you learn how to use the C++ programming elements you have learned so far to create a functioning program. These sections cover the syntax; the use of blanks; the use of semicolons, brackets, and commas; semantics; naming identifiers; prompt lines; documentation, including comments; and form and style.

Syntax

The syntax rules of a language tell what is legal and what is not legal. Errors in syntax are detected during compilation. For example, consider the following C++ statements:

```
int x;           //Line 1
int y           //Line 2
double z;       //Line 3

y = w + x;      //Line 4
```

When these statements are compiled, a compilation error will occur at Line 2 because the semicolon is missing after the declaration of the variable `y`. A second compilation error will occur at Line 4 because the identifier `w` is used but has not been declared.

As discussed in Chapter 1, you enter a program into the computer by using a text editor. When the program is typed, errors are almost unavoidable. Therefore, when the program is compiled, you are most likely to see syntax errors. It is quite possible that a syntax error at a particular place might lead to syntax errors in several subsequent statements. It is very common for the omission of a single character to cause four or five error messages. However, when the first syntax error is removed and the program is recompiled, subsequent syntax errors caused by this syntax error may disappear. Therefore, you should correct syntax errors in the order in which the compiler lists them. As you become more

and:

```
const double CENTIMETERS_PER_INCH = 2.54;
double centimeters;
double inches;

centimeters = inches * CENTIMETERS_PER_INCH;
```

The identifiers in the second set of statements, such as `CENTIMETERS_PER_INCH`, are usually called **self-documenting** identifiers. As you can see, self-documenting identifiers can make comments less necessary.

Consider the self-documenting identifier `annualsale`. This identifier is called a **run-together word**. In using self-documenting identifiers, you may inadvertently include run-together words, which may lessen the clarity of your documentation. You can make run-together words easier to understand by either capitalizing the beginning of each new word or by inserting an underscore just before a new word. For example, you could use either `annualSale` or `annual_sale` to create an identifier that is more clear.

Recall that earlier in this chapter, we specified the general rules for naming named constants and variables. For example, an identifier used to name a named constant is all uppercase. If this identifier is a run-together word, then the words are separated with the underscore character.

Prompt Lines

Part of good documentation is the use of clearly written prompts so that users will know what to do when they interact with a program. There is nothing more frustrating than sitting in front of a running program and not having the foggiest notion of whether to enter something or what to enter. **Prompt lines** are executable statements that inform the user what to do. For example, consider the following C++ statements, in which `num` is an `int` variable:

```
cout << "Please enter a number between 1 and 10 and "
      << "press the return key" << endl;
cin >> num;
```

When these two statements execute in the order given, first the output statement causes the following line of text to appear on the screen:

```
Please enter a number between 1 and 10 and press the return key
```

After seeing this line, users know that they must enter a number and press the return key. If the program contained only the second statement, users would have no idea that they must enter a number, and the computer would wait forever for the input. The preceding output statement is an example of a prompt line.

In a program, whenever input is needed from users, you must include the necessary prompt lines. Furthermore, these prompt lines should include as much information as possible about what input is acceptable. For example, the preceding prompt line not

only tells the user to input a number, but also informs the user that the number should be between 1 and 10.

Documentation

The programs that you write should be clear not only to you, but also to anyone else. Therefore, you must properly document your programs. A well-documented program is easier to understand and modify, even a long time after you originally wrote it. You use comments to document programs. Comments should appear in a program to explain the purpose of the program, identify who wrote it, and explain the purpose of particular statements.

Form and Style

You might be thinking that C++ has too many rules. However, in practice, the rules give C++ a great degree of freedom. For example, consider the following two ways of declaring variables:

```
int feet, inch;
double x, y;
```

and:

```
int feet,inches;double x,y;
```

The computer would have no difficulty understanding either of these formats, but the first form is easier to read and follow. Of course, the omission of a single comma or semicolon in either format may lead to all sorts of strange error messages.

What about blank spaces? Where are they significant and where are they meaningless? Consider the following two statements:

```
int a,b,c;
```

and:

```
int    a,    b,    c;
```

Both of these declarations mean the same thing. Here, the blanks between the identifiers in the second statement are meaningless. On the other hand, consider the following statement:

```
inta,b,c;
```

This statement contains a syntax error. The lack of a blank between `int` and the identifier `a` changes the reserved word `int` and the identifier `a` into a new identifier, `inta`.

The clarity of the rules of syntax and semantics frees you to adopt formats that are pleasing to you and easier to understand.

The following example further elaborates on this.

EXAMPLE 2-30

Consider the following C++ program:

```
//An improperly formatted C++ program.

#include <iostream>
#include <string>
using namespace std;

int main()
{
int num; double height;
string name;
cout << "Enter an integer: "; cin >> num; cout << endl;
    cout<<"num: "<<num<<endl;
cout<<"Enter the first name: "; cin>>name;
    cout<<endl; cout <<"Enter the height: ";
cin>>height; cout<<endl;

cout<<"Name: "<<name<<endl;cout<<"Height: "
<<height; cout <<endl;return 0;
}
```

This program is syntactically correct; the C++ compiler would have no difficulty reading and compiling this program. However, this program is very hard to read. The program that you write should be properly indented and formatted. Note the difference when the program is reformatted:

```
//A properly formatted C++ program.

#include <iostream>
#include <string>

using namespace std;

int main()
{
    int num;
    double height;
    string name;

    cout << "Enter an integer: ";
    cin >> num;
    cout << endl;
```

```

    cout << "num: " << num << endl;

    cout << "Enter the first name: ";
    cin >> name;
    cout << endl;
    cout << "Enter the height: ";
    cin >> height;
    cout << endl;

    cout << "Name: " << name << endl;
    cout << "Height: " << height << endl;

    return 0;
}

```

As you can see, this program is easier to read. Your programs should be properly indented and formatted. To document the variables, programmers typically declare one variable per line. Also, always put a space before and after an operator. When you type your program using an IDE, typically, your program is automatically indented.

More on Assignment Statements

The assignment statements you have seen so far are called **simple assignment statements**. In certain cases, you can use special assignment statements called **compound assignment statements** to write simple assignment statements in a more concise notation.

Corresponding to the five arithmetic operators `+`, `-`, `*`, `/`, and `%`, C++ provides five compound operators: `+=`, `-=`, `*=`, `/=`, and `%=`, respectively. Consider the following simple assignment statement, in which `x` and `y` are `int` variables:

```
x = x * y;
```

Using the compound operator `*=`, this statement can be written as:

```
x *= y;
```

In general, using the compound operator `*=`, you can rewrite the simple assignment statement:

```
variable = variable * (expression);
```

as:

```
variable *= expression;
```

The other arithmetic compound operators have similar conventions. For example, using the compound operator `+=`, you can rewrite the simple assignment statement:

```
variable = variable + (expression);
```

as:

```
variable += expression;
```

The compound assignment statement allows you to write simple assignment statements in a concise fashion by combining an arithmetic operator with the assignment operator.

EXAMPLE 2-31

This example shows several compound assignment statements that are equivalent to simple assignment statements.

Simple Assignment Statement

```
i = i + 5;
counter = counter + 1;
sum = sum + number;
amount = amount * (interest + 1);
x = x / ( y + 5);
```

Compound Assignment Statement

```
i += 5;
counter += 1;
sum += number;
amount *= interest + 1;
x /= y + 5;
```

NOTE

Any compound assignment statement can be converted into a simple assignment statement. However, a simple assignment statement may not be (easily) converted to a compound assignment statement. For example, consider the following simple assignment statement:

```
x = x * y + z - 5;
```

To write this statement as a compound assignment statement, the variable `x` must be a common factor in the right side, which is not the case. Therefore, you cannot immediately convert this statement into a compound assignment statement. In fact, the equivalent compound assignment statement is:

```
x *= y + (z - 5) / x;
```

which is more complicated than the simple assignment statement. Furthermore, in the preceding compound statement, `x` cannot be 0. We recommend avoiding such compound expressions.

NOTE

In programming code, this book typically uses only the compound operator `+=`. So statements such as `a = a + b;` are written as `a += b;`.

PROGRAMMING EXAMPLE: Convert Length

Write a program that takes as input given lengths expressed in feet and inches. The program should then convert and output the lengths in centimeters. Assume that the given lengths in feet and inches are integers.

Input Length in feet and inches.

Output Equivalent length in centimeters.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

The lengths are given in feet and inches, and you need to find the equivalent length in centimeters. One inch is equal to 2.54 centimeters. The first thing the program needs to do is convert the length given in feet and inches to all inches. Then, you can use the conversion formula, 1 inch = 2.54 centimeters, to find the equivalent length in centimeters. To convert the length from feet and inches to inches, you multiply the number of feet by 12, as 1 foot is equal to 12 inches, and add the given inches.

For example, suppose the input is 5 feet and 7 inches. You then find the total inches as follows:

```
totalInches = (12 * feet) + inches
             = 12 * 5 + 7
             = 67
```

You can then apply the conversion formula, 1 inch = 2.54 centimeters, to find the length in centimeters.

```
centimeters = totalInches * 2.54
             = 67 * 2.54
             = 170.18
```

Based on this analysis of the problem, you can design an algorithm as follows:

1. Get the length in feet and inches.
2. Convert the length into total inches.
3. Convert total inches into centimeters.
4. Output centimeters.

Variables The input for the program is two numbers: one for feet and one for inches. Thus, you need two variables: one to store feet and the other to store inches. Because the program will first convert the given length into inches, you need another variable to store the total inches. You also need a variable to store the equivalent length in centimeters. In summary, you need the following variables:

```
int feet;           //variable to hold given feet
int inches;         //variable to hold given inches
int totalInches;    //variable to hold total inches
double centimeters; //variable to hold length in centimeters
```

**Named
Constants**

To calculate the equivalent length in centimeters, you need to multiply the total inches by 2.54. Instead of using the value 2.54 directly in the program, you will declare this value as a named constant. Similarly, to find the total inches, you need to multiply the feet by 12 and add the inches. Instead of using 12 directly in the program, you will also declare this value as a named constant. Using a named constant makes it easier to modify the program later.

```
const double CENTIMETERS_PER_INCH = 2.54;
const int INCHES_PER_FOOT = 12;
```

**MAIN
ALGORITHM**

In the preceding sections, we analyzed the problem and determined the formulas to do the calculations. We also determined the necessary variables and named constants. We can now expand the algorithm given in the section Problem Analysis and Algorithm Design to solve the problem given at the beginning of this programming example.

1. Prompt the user for the input. (Without a prompt line, the user will be staring at a blank screen and will not know what to do.)
2. Get the data.
3. Echo the input—that is, output what the program read as input. (Without this step, after the program has executed, you will not know what the input was.)
4. Find the length in inches.
5. Output the length in inches.
6. Convert the length to centimeters.
7. Output the length in centimeters.

**Putting It
Together**

Now that the problem has been analyzed and the algorithm has been designed, the next step is to translate the algorithm into C++ code. Because this is the first complete C++ program you are writing, let's review the necessary steps in sequence.

The program will begin with comments that document its purpose and functionality. As there is both input to this program (the length in feet and inches) and output (the equivalent length in centimeters), you will be using system resources for input/output. In other words, the program will use input statements to get data into the program and output statements to print the results. Because the data will be entered from the keyboard and the output will be displayed on the screen, the program must include the header file `iostream`. Thus, the first statement of the program, after the comments as described above, will be the preprocessor directive to include this header file.

This program requires two types of memory locations for data manipulation: named constants and variables. Typically, named constants hold special data, such as `CENTIMETERS_PER_INCH`. Depending on the nature of a named constant, it can be placed before the function `main` or within the function `main`. If a named constant is to be

used throughout the program, then it is typically placed before the function `main`. We will comment further on where to put named constants within a program in Chapter 7, when we discuss user-defined functions in general. Until then, usually, we will place named constants before the function `main` so that they can be used throughout the program.

This program has only one function, the function `main`, which will contain all of the programming instructions in its body. In addition, the program needs variables to manipulate data, and these variables will be declared in the body of the function `main`. The reasons for declaring variables in the body of the function `main` are explained in Chapter 7. The body of the function `main` will also contain the C++ statements that implement the algorithm. Therefore, the body of the function `main` has the following form:

```
int main()
{
    declare variables

    statements

    return 0;
}
```

To write the complete length conversion program, follow these steps:

1. Begin the program with comments for documentation.
2. Include header files, if any are used in the program.
3. Declare named constants, if any.
4. Write the definition of the function `main`.

COMPLETE PROGRAM LISTING

```

//*****
// Author: D. S. Malik
//
// Program Convert Measurements: This program converts
// measurements in feet and inches into centimeters using
// the formula that 1 inch is equal to 2.54 centimeters.
//*****

//Header file
#include <iostream>

using namespace std;

//Named constants
const double CENTIMETERS_PER_INCH = 2.54;
const int INCHES_PER_FOOT = 12;
```

```

int main ()
{
    //Declare variables
    int feet, inches;
    int totalInches;
    double centimeter;

    //Statements: Step 1 - Step 7
    cout << "Enter two integers, one for feet and "
         << "one for inches: "; //Step 1
    cin >> feet >> inches; //Step 2
    cout << endl;
    cout << "The numbers you entered are " << feet
         << " for feet and " << inches
         << " for inches. " << endl; //Step 3

    totalInches = INCHES_PER_FOOT * feet + inches; //Step 4

    cout << "The total number of inches = "
         << totalInches << endl; //Step 5

    centimeter = CENTIMETERS_PER_INCH * totalInches; //Step 6

    cout << "The number of centimeters = "
         << centimeter << endl; //Step 7

    return 0;
}

```

Sample Run: In this sample run, the user input is shaded.

Enter two integers, one for feet, one for inches: 15 7

The numbers you entered are 15 for feet and 7 for inches.

The total number of inches = 187

The number of centimeters = 474.98

PROGRAMMING EXAMPLE: Make Change

Write a program that takes as input any change expressed in cents. It should then compute the number of half-dollars, quarters, dimes, nickels, and pennies to be returned, returning as many half-dollars as possible, then quarters, dimes, nickels, and pennies, in that order. For example, 483 cents should be returned as 9 half-dollars, 1 quarter, 1 nickel, and 3 pennies.

Input Change in cents.

Output Equivalent change in half-dollars, quarters, dimes, nickels, and pennies.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Suppose the given change is **646** cents. To find the number of half-dollars, you divide **646** by **50**, the value of a half-dollar, and find the quotient, which is **12**, and the remainder, which is **46**. The quotient, **12**, is the number of half-dollars, and the remainder, **46**, is the remaining change.

Next, divide the remaining change by **25** to find the number of quarters. Since the remaining change is **46**, division by **25** gives the quotient **1**, which is the number of quarters, and a remainder of **21**, which is the remaining change. This process continues for dimes and nickels. To calculate the remainder in an integer division, you use the mod operator, `%`.

Applying this discussion to **646** cents yields the following calculations:

1. Change = **646**
2. Number of half-dollars = $646 / 50 = 12$
3. Remaining change = $646 \% 50 = 46$
4. Number of quarters = $46 / 25 = 1$
5. Remaining change = $46 \% 25 = 21$
6. Number of dimes = $21 / 10 = 2$
7. Remaining change = $21 \% 10 = 1$
8. Number of nickels = $1 / 5 = 0$
9. Number of pennies = remaining change = $1 \% 5 = 1$

This discussion translates into the following algorithm:

1. Get the change in cents.
2. Find the number of half-dollars.
3. Calculate the remaining change.
4. Find the number of quarters.
5. Calculate the remaining change.
6. Find the number of dimes.
7. Calculate the remaining change.
8. Find the number of nickels.
9. Calculate the remaining change, which is the number of pennies.

Variables From the previous discussion and algorithm, it appears that the program will need variables to hold the number of half-dollars, quarters, and so on. However, the numbers of half-dollars, quarters, and so on are not used in later calculations, so the program can simply output these values without saving each of them in a variable. The only thing that keeps changing is the change, so the program actually needs only one variable:

```
int change;
```

Named Constants To calculate the equivalent change, the program performs calculations using the values of a half-dollar, which is 50; a quarter, which is 25; a dime, which is 10; and a nickel, which is 5. Because these data are special and the program uses these values more than once, it makes sense to declare them as named constants. Using named constants also simplifies later modification of the program:

```
const int HALF_DOLLAR = 50;
const int QUARTER  = 25;
const int DIME     = 10;
const int NICKEL   = 5;
```

MAIN ALGORITHM

1. Prompt the user for input.
2. Get input.
3. Echo the input by displaying the entered change on the screen.
4. Compute and print the number of half-dollars.
5. Calculate the remaining change.
6. Compute and print the number of quarters.
7. Calculate the remaining change.
8. Compute and print the number of dimes.
9. Calculate the remaining change.
10. Compute and print the number of nickels.
11. Calculate the remaining change.
12. Print the remaining change.

COMPLETE PROGRAM LISTING

```
//*****
// Author: D. S. Malik
//
// Program Make Change: Given any amount of change expressed
// in cents, this program computes the number of half-dollars,
// quarters, dimes, nickels, and pennies to be returned,
// returning as many half-dollars as possible, then quarters,
// dimes, nickels, and pennies in that order.
//*****

//Header file
#include <iostream>

using namespace std;

//Named constants
const int HALF_DOLLAR = 50;
const int QUARTER  = 25;
const int DIME     = 10;
const int NICKEL   = 5;
```

```

int main()
{
    //Declare variable
    int change;

    //Statements: Step 1 - Step 12
    cout << "Enter change in cents: ";           //Step 1
    cin >> change;                               //Step 2
    cout << endl;

    cout << "The change you entered is " << change
        << endl;                               //Step 3

    cout << "The number of half-dollars to be returned "
        << "is " << change / HALF_DOLLAR
        << endl;                               //Step 4

    change = change % HALF_DOLLAR;              //Step 5

    cout << "The number of quarters to be returned is "
        << change / QUARTER << endl;           //Step 6

    change = change % QUARTER;                  //Step 7

    cout << "The number of dimes to be returned is "
        << change / DIME << endl;             //Step 8

    change = change % DIME;                     //Step 9

    cout << "The number of nickels to be returned is "
        << change / NICKEL << endl;           //Step 10

    change = change % NICKEL;                   //Step 11

    cout << "The number of pennies to be returned is "
        << change << endl;                   //Step 12

    return 0;
}

```

Sample Run: In this sample run, the user input is shaded.

Enter change in cents: 583

The change you entered is 583
 The number of half-dollars to be returned is 11
 The number of quarters to be returned is 1
 The number of dimes to be returned is 0
 The number of nickels to be returned is 1
 The number of pennies to be returned is 3

QUICK REVIEW

1. A C++ program is a collection of functions.
2. Every C++ program has a function called `main`.
3. A single-line comment starts with the pair of symbols `//` anywhere in the line.
4. Multiline comments are enclosed between `/*` and `*/`.
5. The compiler skips comments.
6. Reserved words cannot be used as identifiers in a program.
7. All reserved words in C++ consist of lowercase letters (see Appendix A).
8. In C++, identifiers are names of things.
9. A C++ identifier consists of letters, digits, and underscores and must begin with a letter or underscore.
10. Whitespaces include blanks, tabs, and newline characters.
11. A data type is a set of values together with a set of operations.
12. C++ data types fall into the following three categories: simple, structured, and pointers.
13. There are three categories of simple data: integral, floating-point, and enumeration.
14. Integral data types are classified into nine categories: `char`, `short`, `int`, `long`, `bool`, `unsigned char`, `unsigned short`, `unsigned int`, and `unsigned long`.
15. The values belonging to `int` data type are -2147483648 ($= -2^{31}$) to 2147483647 ($= 2^{31} - 1$).
16. The data type `bool` has only two values: `true` and `false`.
17. The most common character sets are ASCII, which has 128 values, and EBCDIC, which has 256 values.
18. The collating sequence of a character is its preset number in the character data set.
19. C++ provides three data types to manipulate decimal numbers: `float`, `double`, and `long double`.
20. The data type `float` is used in C++ to represent any real number between $-3.4E + 38$ and $3.4E + 38$. The memory allocated for a value of the `float` data type is four bytes.
21. The data type `double` is used in C++ to represent any real number between $-1.7E + 308$ and $1.7E + 308$. The memory allocated for a value of the `double` data type is eight bytes.
22. The arithmetic operators in C++ are addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).

23. The modulus operator, `%`, takes only integer operands.
24. Arithmetic expressions are evaluated using the precedence rules and the associativity of the arithmetic operators.
25. All operands in an integral expression, or integer expression, are integers, and all operands in a floating-point expression are decimal numbers.
26. A mixed expression is an expression that consists of both integers and decimal numbers.
27. When evaluating an operator in an expression, an integer is converted to a floating-point number, with a decimal part of 0, only if the operator has mixed operands.
28. You can use the cast operator to explicitly convert values from one data type to another.
29. A string is a sequence of zero or more characters.
30. Strings in C++ are enclosed in double quotation marks.
31. A string containing no characters is called a null or empty string.
32. Every character in a string has a relative position in the string. The position of the first character is 0, the position of the second character is 1, and so on.
33. The length of a string is the number of characters in it.
34. During program execution, the contents of a named constant cannot be changed.
35. A named constant is declared by using the reserved word `const`.
36. A named constant is initialized when it is declared.
37. All variables must be declared before they can be used.
38. C++ does not automatically initialize variables.
39. Every variable has a name, a value, a data type, and a size.
40. When a new value is assigned to a variable, the old value is lost.
41. Only an assignment statement or an input (read) statement can change the value of a variable.
42. In C++, `>>` is called the stream extraction operator.
43. Input from the standard input device is accomplished by using `cin` and the stream extraction operator `>>`.
44. When data is input in a program, the data items, such as numbers, are usually separated by blanks, lines, or tabs.
45. In C++, `<<` is called the stream insertion operator.
46. Output of the program to the standard output device is accomplished by using `cout` and the stream insertion operator `<<`.
47. The manipulator `endl` positions the insertion point at the beginning of the next line on an output device.

48. Outputting or accessing the value of a variable in an expression does not destroy or modify the contents of the variable.
49. The character `\` is called the escape character.
50. The sequence `\n` is called the newline escape sequence.
51. All preprocessor commands start with the symbol `#`.
52. The preprocessor commands are processed by the preprocessor before the program goes through the compiler.
53. The preprocessor command `#include <iostream>` instructs the preprocessor to include the header file `iostream` in the program.
54. To use `cin` and `cout`, the program must include the header file `iostream` and either include the statement `using namespace std;` or refer to these identifiers as `std::cin` and `std::cout`.
55. All C++ statements end with a semicolon. The semicolon in C++ is called the statement terminator.
56. A C++ system has three components: environment, language, and the standard libraries.
57. Standard libraries are not part of the C++ language. They contain functions to perform operations, such as mathematical operations.
58. A file containing a C++ program usually ends with the extension `.cpp`.
59. Prompt lines are executable statements that tell the user what to do.
60. Corresponding to the five arithmetic operators `+`, `-`, `*`, `/`, and `%`, C++ provides five compound operators: `+=`, `-=`, `*=`, `/=`, and `%=`, respectively.

EXERCISES

1. Mark the following statements as true or false.
 - a. An identifier can be any sequence of digits and letters.
 - b. In C++, there is no difference between a reserved word and a pre-defined identifier.
 - c. A C++ identifier can start with a digit.
 - d. The operands of the modulus operator must be integers.
 - e. If `a = 4;` and `b = 3;`, then after the statement `a = b;` the value of `b` is still 3.
 - f. In the statement `cin >> y;`, `y` can only be an `int` or a `double` variable.
 - g. In an output statement, the newline character may be a part of the string.
 - h. The following is a legal C++ program:

```
int main()
{
    return 0;
}
```

- i. In a mixed expression, all the operands are converted to floating-point numbers.
 - j. Suppose `x = 5`. After the statement `y = x++`; executes, `y` is 5 and `x` is 6.
 - k. Suppose `a = 5`. After the statement `++a`; executes, the value of `a` is still 5 because the value of the expression is not saved in another variable.
2. Which of the following are valid C++ identifiers?
- a. `myFirstProgram` b. `MIX-UP` c. `C++Program2` d. `quiz7`
 - e. `ProgrammingLecture2` f. `1footEquals12Inches`
 - g. `Mike'sFirstAttempt` h. `Update Grade` i. `4th`
 - j. `New_Student`
3. Which of the following is a reserved word in C++?
- a. `Const` b. `include` c. `Char` d. `void` e. `int` f. `Return`
4. What is the difference between a keyword and a user-defined identifier?
5. Are the identifiers `firstName` and `FirstName` the same?
6. Evaluate the following expressions.
- a. `25 / 3` b. `20 - 12 / 4 * 2` c. `32 % 7` d. `3 - 5 % 7`
 - e. `18.0 / 4` f. `28 - 5 / 2.0` g. `17 + 5 % 2 - 3`
 - h. `15.0 + 3.0 * 2.0 / 5.0`
7. If `x = 5`, `y = 6`, `z = 4`, and `w = 3.5`, evaluate each of the following statements, if possible. If it is not possible, state the reason.
- a. `(x + z) % y` b. `(x + y) % w` c. `(y + w) % x` d. `(x + y) * w`
 - e. `(x % y) % z` f. `(y % z) % x` g. `(x * z) % y` h. `((x * y) * w) * z`

8. Given:

```
int num1, num2, newNum;
double x, y;
```

Which of the following assignments are valid? If an assignment is not valid, state the reason.

When not given, assume that each variable is declared.

- a. `num1 = 35;`
- b. `newNum = num1 - num2;`
- c. `num1 = 5; num2 = 2 + num1; num1 = num2 / 3;`
- d. `num1 * num2 = newNum;`
- e. `x = 12 * num1 - 15.3;`
- f. `num1 * 2 = newNum + num2;`
- g. `x / y = x * y;`

- h. `num2 = num1 % 2.0;`
 - i. `newNum = static_cast<int> (x) % 5;`
 - j. `x = x + y - 5;`
 - k. `newNum = num1 + static_cast<int> (4.6 / 2);`
9. Do a walk-through to find the value assigned to `e`. Assume that all variables are properly declared.
- ```

a = 3;
b = 4;
c = (a % b) * 6;
d = c / b;
e = (a + b + c + d) / 4;

```
10. Which of the following variable declarations are correct? If a variable declaration is not correct, give the reason(s) and provide the correct variable declaration.
- ```

n = 12;                //Line 1
char letter = ;        //Line 2
int one = 5, two;      //Line 3
double x, y, z;        //Line 4

```
11. Which of the following are valid C++ assignment statements? Assume that `i`, `x`, and `percent` are `double` variables.
- a. `i = i + 5;`
 - b. `x + 2 = x;`
 - c. `x = 2.5 * x;`
 - d. `percent = 10%;`
12. Write C++ statement(s) that accomplish the following.
- a. Declare `int` variables `x` and `y`. Initialize `x` to 25 and `y` to 18.
 - b. Declare and initialize an `int` variable `temp` to 10 and a `char` variable `ch` to 'A'.
 - c. Update the value of an `int` variable `x` by adding 5 to it.
 - d. Declare and initialize a `double` variable `payRate` to 12.50.
 - e. Copy the value of an `int` variable `firstNum` into an `int` variable `tempNum`.
 - f. Swap the contents of the `int` variables `x` and `y`. (Declare additional variables, if necessary.)
 - g. Suppose `x` and `y` are `double` variables. Output the contents of `x`, `y`, and the expression `x + 12 / y - 18`.
 - h. Declare a `char` variable `grade` and set the value of `grade` to 'A'.
 - i. Declare `int` variables to store four integers.
 - j. Copy the value of a `double` variable `z` to the nearest integer into an `int` variable `x`.

13. Write each of the following as a C++ expression.
- 32 times `a` plus `b`
 - The character that represents 8
 - The string that represents the name Julie Nelson.
 - $(b^2 - 4ac) / 2a$
 - $(a + b) / c(e f) - gh$
 - $(-b + (b^2 - 4ac)) / 2a$
14. Suppose `x`, `y`, `z`, and `w` are `int` variables. What value is assigned to each of these variables after the last statement executes?
- ```
x = 5; z = 3;
y = x - z;
z = 2 * y + 3;
w = x - 2 * y + z;
z = w - x;
w++;
```
15. Suppose `x`, `y`, and `z` are `int` variables and `w` and `t` are `double` variables. What value is assigned to each of these variables after the last statement executes?
- ```
x = 17;
y = 15;
x = x + y / 4;
z = x % 3 + 4;
w = 17 / 3 + 6.5;
t = x / 4.0 + 15 % 4 - 3.5;
```
16. Suppose `x`, `y`, and `z` are `int` variables and `x` = 2, `y` = 5, and `z` = 6. What is the output of each of the following statements?
- `cout << "x = " << x << ", y = " << y << ", z = " << z << endl;`
 - `cout << "x + y = " << x + y << endl;`
 - `cout << "Sum of " << x << " and " << z << " is " << x + z << endl;`
 - `cout << "z / x = " << z / x << endl;`
 - `cout << "2 times " << x << " = " << 2 * x << endl;`
17. What is the output of the following statements? Suppose `a` and `b` are `int` variables, `c` is a `double` variable, and `a` = 13, `b` = 5, and `c` = 17.5.
- `cout << a + b - c << endl;`
 - `cout << 15 / 2 + c << endl;`
 - `cout << a / static_cast<double>(b) + 2 * c << endl;`
 - `cout << 14 % 3 + 6.3 + b / a << endl;`
 - `cout << static_cast<int>(c) % 5 + a - b << endl;`
 - `cout << 13.5 / 2 + 4.0 * 3.5 + 18 << endl;`

18. Write C++ statements that accomplish the following.
 - a. Output the newline character.
 - b. Output the tab character.
 - c. Output double quotation mark.
19. Which of the following are correct C++ statements?
 - a. `cout << "Hello There!" << endl;`
 - b. `cout << "Hello";`
`<< " There!" << endl;`
 - c. `cout << "Hello"`
`<< " There!" << endl;`
 - d. `cout << 'Hello There!' << endl;`
20. Give meaningful identifiers for the following variables.
 - a. A variable to store the first name of a student.
 - b. A variable to store the discounted price of an item.
 - c. A variable to store the number of juice bottles.
 - d. A variable to store the number of miles traveled.
 - e. A variable to store the highest test score.
21. Write C++ statements to do the following.
 - a. Declare `int` variable `num1` and `num2`.
 - b. Prompt the user to input two numbers.
 - c. Input the first number in `num1` and the second number in `num2`.
 - d. Output `num1`, `num2`, and 2 times `num1` minus `num2`. Your output must identify each number and the expression.
22. The following program has syntax mistakes. Correct them. On each successive line, assume that any preceding error has been corrected.

```
#include <iostream>

const int SECRET_NUM = 11,213;
const PAY_RATE = 18.35

main()
{
    int one, two;
    double first, second;
    one = 18;
    two = 11;

    first = 25;
    second = first * three;
```

```

second = 2 * SECRET_NUM;
SECRET_NUM = SECRET_NUM + 3;
cout << first << " " << second << SECRET_NUM << endl;

paycheck = hoursWorked * PAY_RATE

cout << "Wages = " << paycheck << endl;
return 0;
}

```

23. The following program has syntax mistakes. Correct them. On each successive line, assume that any preceding error has been corrected.

```

const char = STAR = '*'
const int PRIME = 71;

int main
{
    int count, sum;
    double x;

    count = 1;
    sum = count + PRIME;
    x := 25.67;
    newNum = count * ONE + 2;
    sum + count = sum;
    x = x + sum * COUNT;
    cout << " count = " << count << ", sum = " << sum
        << ", PRIME = " << Prime << endl;
}

```

24. The following program has syntax errors. Correct them. On each successive line, assume that any preceding error has been corrected.

```

#include <iostream>

using namespace std;

int main()
{
    int temp;
    string first;

    cout << "Enter first name: ;
    cin >> first
    cout << endl;

    cout << "Enter last name: ;
    cin >> last;
    cout << endl;

    cout << "Enter today's temperature: ";
    cin >> temperature;
    cout << endl;
}

```

```
cout << first << " " << last << today's temperature is: ";
    << temperature << endl;
```

```
return 0;
```

```
}
```

25. What action must be taken before a variable can be used in a program?
26. Preprocessor directives begin with which of the following symbols:
a. * b. # c. \$ d. ! e. None of these.
27. Write equivalent compound statements if possible.
- a. $x = 2 * x$ b. $x = x + y - 2;$ c. $sum = sum + num;$
d. $z = z * x + 2 * z;$ e. $y = y / (x + 5);$
28. Write the following compound statements as equivalent simple statements.
- a. $x += 5 - z;$ b. $y *= 2 * x + 5 - z;$ c. $w += 2 * z + 4;$
d. $x -= z + y - t;$ e. $sum += num;$
29. Suppose a, b, and c are `int` variables and $a = 5$ and $b = 6$. What value is assigned to each variable after each statement executes? If a variable is undefined at a particular statement, report UND (undefined).

	a	b	c
$a = (b++) + 3;$	—	—	—
$c = 2 * a + (++b);$	—	—	—
$b = 2 * (++c) - (a++);$	—	—	—

30. Suppose a, b, and sum are `int` variables and c is a `double` variable. What value is assigned to each variable after each statement executes? Suppose $a = 3$, $b = 5$, and $c = 14.1$.

	a	b	c	sum
$sum = a + b + c;$	—	—	—	—
$c /= a;$	—	—	—	—
$b += c - a;$	—	—	—	—
$a *= 2 * b + c;$	—	—	—	—

31. What is printed by the following program? Suppose the input is:
20 15

```
#include <iostream>

using namespace std;

const int NUM = 10;
const double X = 20.5;

int main()
{
    int a, b;
    double z;
```

```

char grade;

a = 25;

cout << "a = " << a << endl;

cout << "Enter two integers: ";
cin >> a >> b;
cout << endl;

cout << "The numbers you entered are "
    << a << " and " << b << endl;

z = X + 2 * a - b;
cout << "z = " << z << endl;

grade = 'A';
cout << "Your grade is " << grade << endl;

a = 2 * NUM + z;
cout << "The value of a = " << a << endl;

return 0;
}

```

32. What is printed by the following program? Suppose the input is:

```

Miller
34
340

```

```

#include <iostream>
#include <string>

using namespace std;

const int PRIME_NUM = 11;

int main()
{
    const int SECRET = 17;

    string name;
    int id;
    int num;
    int mysteryNum;

    cout << "Enter last name: ";
    cin >> name;
    cout << endl;

    cout << "Enter a two digit number: ";
    cin >> num;
    cout << endl;

```

```

    id = 100 * num + SECRET;

    cout << "Enter a positive integer less than 1000: ";
    cin >> num;
    cout << endl;

    mysteryNum = num * PRIME_NUM - 3 * SECRET;

    cout << "Name: " << name << endl;
    cout << "Id: " << id << endl;
    cout << "Mystery number: " << mysteryNum << endl;

    return 0;
}

```

33. Rewrite the following program so that it is properly formatted.

```

#include <iostream>
#include <string>
using namespace std;
const double X = 13.45; const int Y=34;
const char BLANK= ' ';
int main()
{string firstName,lastName;int num;
double salary;
cout<<"Enter first name: "; cin>> firstName; cout<<endl;
cout<<"Enter last name: "; cin
>>lastName;cout<<endl;
    cout<<"Enter a positive integer less than 70:";
cin>>num;cout<<endl; salary=num*X;
    cout<<"Name: "<<firstName<<BLANK<<lastName<<endl;cout
<<"Wages: $"<<salary<<endl; cout<<"X = "<<X<<endl;
    cout<<"X+Y = " << X+Y << endl; return 0;
}

```

34. What type of input does the following program require, and in what order does the input need to be provided?

```

#include <iostream>

using namespace std;

int main()
{
    int age;
    double weight;
    string firstName, lastName;

    cin >> firstName >> lastName;
    cin >> age >> weight;

    return 0;
}

```

PROGRAMMING EXERCISES

1. Write a program that produces the following output:

```
*****
*   Programming Assignment 1   *
*   Computer Programming I    *
*   Author: ???               *
*   Due Date: Thursday, Jan. 24 *
*****
```

In your program, substitute ??? with your own name. If necessary, adjust the positions and the number of the stars to produce a rectangle.

2. Write a program that produces the following output:

```
CCCCCCCC      ++              ++
CC             ++              ++
CC             ++++++          ++++++
CC             ++++++          ++++++
CC             ++              ++
CCCCCCCC      ++              ++
```

3. Consider the following program segment

```
//include statement(s)
//using namespace statement

int main()
{
    //variable declaration

    //executable statements

    //return statement
}
```

- a. Write C++ statements that include the header files `iostream`.
- b. Write a C++ statement that allows you to use `cin`, `cout`, and `endl` without the prefix `std::`.
- c. Write C++ statements that declare the following variables: `num1`, `num2`, `num3`, and `average` of type `int`.
- d. Write C++ statements that store 125 into `num1`, 28 into `num2`, and -25 into `num3`.
- e. Write a C++ statement that stores the average of `num1`, `num2`, and `num3`, into `average`.
- f. Write C++ statements that output the values of `num1`, `num2`, `num3`, and `average`.
- g. Compile and run your program.

4. Repeat Exercise 3 by declaring `num1`, `num2`, and `num3`, and `average` of type `double`. Store 75.35 into `num1`, -35.56 into `num2`, and 15.76 into `num3`.
5. Consider the following C++ program in which the statements are in the incorrect order. Rearrange the statements so that it prompts the user to input the length and width of a rectangle and output the area and perimeter of the rectangle.

```
#include <iostream>
{
    int main()

        cout << "Enter the length: ";
        cin >> length;
        cout << endl;

        int length;

        area = length * width;

        return 0;

        int width;

        cin>> width;
        cout << "Enter the width: "
        cout << endl;

        cout << "Area = " << area << endl;
        cout << "Perimeter = " << perimeter << endl;

        int area;
        using namespace std;
        int perimeter;
}
```

6. Consider the following program segment:

```
//include statement(s)
//using namespace statement

int main()
{
    //variable declaration

    //executable statements

    //return statement
}
```


- a. Write C++ statements that include the header files `iostream` and `string`.
- b. Write a C++ statement that allows you to use `cin`, `cout`, and `endl` without the prefix `std::`.
- c. Write C++ statements that declare the following variables: `name` of type `string` and `studyHours` of type `double`.
- d. Write C++ statements that prompt and input a string into `name` and a `double` value into `studyHours`.
- e. Write a C++ statement that outputs the values of `name` and `studyHours` with the appropriate text. For example, if the value of `name` is "Donald" and the value of `studyHours` is 4.5, the output is:

Hello, Donald! on Saturday, you need to study 4.5 hours for the exam.

- f. Compile and run your program.
7. Write a program that prompts the user to input a decimal number and outputs the number rounded to the nearest integer.
8. Consider the following program segment:

```
//include statement(s)
//using namespace statement

int main()
{
    //variable declaration

    //executable statements

    //return statement
}
```

- a. Write C++ statements that include the header files `iostream` and `string`.
- b. Write a C++ statement that allows you to use `cin`, `cout`, and `endl` without the prefix `std::`.
- c. Write C++ statements that declare and initialize the following named constants: `SECRET` of type `int` initialized to 11 and `RATE` of type `double` initialized to 12.50.
- d. Write C++ statements that declare the following variables: `num1`, `num2`, and `newNum` of type `int`; `name` of type `string`; and `hoursWorked` and `wages` of type `double`.
- e. Write C++ statements that prompt the user to input two integers and store the first number in `num1` and the second number in `num2`.

- f. Write a C++ statement(s) that outputs the values of `num1` and `num2`, indicating which is `num1` and which is `num2`. For example, if `num1` is 8 and `num2` is 5, then the output is:

The value of `num1` = 8 and the value of `num2` = 5.

- g. Write a C++ statement that multiplies the value of `num1` by 2, adds the value of `num2` to it, and then stores the result in `newNum`. Then, write a C++ statement that outputs the value of `newNum`.
- h. Write a C++ statement that updates the value of `newNum` by adding the value of the named constant `SECRET`. Then, write a C++ statement that outputs the value of `newNum` with an appropriate message.
- i. Write C++ statements that prompt the user to enter a person's last name and then store the last name into the variable `name`.
- j. Write C++ statements that prompt the user to enter a decimal number between 0 and 70 and then store the number entered into `hoursWorked`.
- k. Write a C++ statement that multiplies the value of the named constant `RATE` with the value of `hoursWorked` and then stores the result into the variable `wages`.
- l. Write C++ statements that produce the following output:

```
Name:           //output the value of the variable name
Pay Rate: $      //output the value of the variable rate
Hours Worked:    //output the value of the variable
                 //hoursWorked
Salary: $        //output the value of the variable wages
```

For example, if the value of `name` is "Rainbow" and `hoursWorked` is 45.50, then the output is:

```
Name: Rainbow
Pay Rate: $12.50
Hours Worked: 45.50
Salary: $568.75
```

- m. Write a C++ program that tests each of the C++ statements that you wrote in parts a through l. Place the statements at the appropriate place in the previous C++ program segment. Test run your program (twice) on the following input data:
 - a. `num1 = 13, num2 = 28; name = "Jacobson"; hoursWorked = 48.30.`
 - b. `num1 = 32, num2 = 15; name = "Crawford"; hoursWorked = 58.45.`
- 9. Write a program that prompts the user to enter five test scores and then prints the average test score. (Assume that the test scores are decimal numbers.)

10. Write a program that prompts the user to input five decimal numbers. The program should then add the five decimal numbers, convert the sum to the nearest integer, and print the result.
11. Write a program that does the following:
 - a. Prompts the user to input five decimal numbers.
 - b. Prints the five decimal numbers.
 - c. Converts each decimal number to the nearest integer.
 - d. Adds the five integers.
 - e. Prints the sum and average of the five integers.
12. Write a program that prompts the capacity, in gallons, of an automobile fuel tank and the miles per gallons the automobile can be driven. The program outputs the number of miles the automobile can be driven without refueling.
13. Write a C++ program that prompts the user to input the elapsed time for an event in seconds. The program then outputs the elapsed time in hours, minutes, and seconds. (For example, if the elapsed time is 9630 seconds, then the output is 2:40:30.)
14. Write a C++ program that prompts the user to input the elapsed time for an event in hours, minutes, and seconds. The program then outputs the elapsed time in seconds.
15. To make a profit, a local store marks up the prices of its items by a certain percentage. Write a C++ program that reads the original price of the item sold, the percentage of the marked-up price, and the sales tax rate. The program then outputs the original price of the item, the percentage of the mark-up, the store's selling price of the item, the sales tax rate, the sales tax, and the final price of the item. (The final price of the item is the selling price plus the sales tax.)
16. Write a program that prompts the user to input a length expressed in centimeters. The program should then convert the length to inches (to the nearest inch) and output the length expressed in yards, feet, and inches, in that order. For example, suppose the input for centimeters is 312. To the nearest inch, 312 centimeters is equal to 123 inches. 123 inches would thus be output as:

3 yard(s), 1 foot (foot), and 3 inch(es).
17. Write a program to implement and test the algorithm that you designed for Exercise 15 of Chapter 1. (You may assume that the value of $\pi = 3.141593$. In your program, declare a named constant `PI` to store this value.)
18. A milk carton can hold 3.78 liters of milk. Each morning, a dairy farm ships cartons of milk to a local grocery store. The cost of producing one liter of milk is \$0.38, and the profit of each carton of milk is \$0.27. Write a program that does the following:

- a. Prompts the user to enter the total amount of milk produced in the morning.
 - b. Outputs the number of milk cartons needed to hold milk. (Round your answer to the nearest integer.)
 - c. Outputs the cost of producing milk.
 - d. Outputs the profit for producing milk.
19. Redo Programming Exercise 18 so that the user can also input the cost of producing one liter of milk and the profit on each carton of milk.
20. You found an exciting summer job for five weeks. It pays, say, \$15.50 per hour. Suppose that the total tax you pay on your summer job income is 14%. After paying the taxes, you spend 10% of your net income to buy new clothes and other accessories for the next school year and 1% to buy school supplies. After buying clothes and school supplies, you use 25% of the remaining money to buy savings bonds. For each dollar you spend to buy savings bonds, your parents spend \$0.50 to buy additional savings bonds for you. Write a program that prompts the user to enter the pay rate for an hour and the number of hours you worked each week. The program then outputs the following:
- a. Your income before and after taxes from your summer job.
 - b. The money you spend on clothes and other accessories.
 - c. The money you spend on school supplies.
 - d. The money you spend to buy savings bonds.
 - e. The money your parents spend to buy additional savings bonds for you.
21. A permutation of three objects, a , b , and c , is any arrangement of these objects in a row. For example, some of the permutations of these objects are abc , bca , and cab . The number of permutations of three objects is six. Suppose that these three objects are strings. Write a program that prompts the user to enter three strings. The program then outputs the six permutations of those strings.
22. Write a program that prompts the user to input a number of quarters, dimes, and nickels. The program then outputs the total value of the coins in pennies.
23. Newton's law states that the force, F , between two bodies of masses M_1 and M_2 is given by:


$$F = k \left(\frac{M_1 M_2}{d^2} \right),$$

in which k is the gravitational constant and d is the distance between the bodies. The value of k is approximately 6.67×10^{-8} dyn. cm²/g². Write a

program that prompts the user to input the masses of the bodies and the distance between the bodies. The program then outputs the force between the bodies.

24. One metric ton is approximately 2205 pounds. Write a program that prompts the user to input the amount of rice, in pounds, in a bag. The program outputs the number of bags needed to store one metric ton of rice.
25. Cindy uses the services of a brokerage firm to buy and sell stocks. The firm charges 1.5% service charges on the total amount for each transaction, buy or sell. When Cindy sells stocks, she would like to know if she gained or lost on a particular investment. Write a program that allows Cindy to input the number of shares sold, the purchase price of each share, and the selling price of each share. The program outputs the amount invested, the total service charges, amount gained or lost, and the amount received after selling the stock.

This page intentionally left blank



3 CHAPTER

INPUT/OUTPUT

IN THIS CHAPTER, YOU WILL:

- Learn what a stream is and examine input and output streams
- Explore how to read data from the standard input device
- Learn how to use predefined functions in a program
- Explore how to use the input stream functions `get`, `ignore`, `putback`, and `peek`
- Become familiar with input failure
- Learn how to write data to the standard output device
- Discover how to use manipulators in a program to format output
- Learn how to perform input and output operations with the `string` data type
- Learn how to debug logic errors
- Become familiar with file input and output

In Chapter 2, you were introduced to some of C++’s input/output (I/O) instructions, which get data into a program and print the results on the screen. You used `cin` and the extraction operator `>>` to get data from the keyboard, and `cout` and the insertion operator `<<` to send output to the screen. Because I/O operations are fundamental to any programming language, in this chapter, you will learn about C++’s I/O operations in more detail. First, you will learn about statements that extract input from the standard input device and send output to the standard output device. You will then learn how to format output using manipulators. In addition, you will learn about the limitations of the I/O operations associated with the standard input/output devices and learn how to extend these operations to other devices.

I/O Streams and Standard I/O Devices

A program performs three basic operations: it gets data, it manipulates the data, and it outputs the results. In Chapter 2, you learned how to manipulate numeric data using arithmetic operations. In later chapters, you will learn how to manipulate nonnumeric data. Because writing programs for I/O is quite complex, C++ offers extensive support for I/O operations by providing substantial prewritten I/O operations, some of which you encountered in Chapter 2. In this chapter, you will learn about various I/O operations that can greatly enhance the flexibility of your programs.

In C++, I/O is a sequence of bytes, called a stream, from the source to the destination. The bytes are usually characters, unless the program requires other types of information, such as a graphic image or digital speech. Therefore, a **stream** is a sequence of characters from the source to the destination. There are two types of streams:

Input stream: A sequence of characters from an input device to the computer.

Output stream: A sequence of characters from the computer to an output device.

Recall that the standard input device is usually the keyboard, and the standard output device is usually the screen. To receive data from the keyboard and send output to the screen, every C++ program must use the header file `iostream`. This header file contains, among other things, the definitions of two data types, `istream` (input stream) and `ostream` (output stream). The header file also contains two variable declarations, one for `cin` (pronounced “see-in”), which stands for **common input**, and one for `cout` (pronounced “see-out”), which stands for **common output**.

These variable declarations are similar to the following C++ statements:

```
istream cin;
ostream cout;
```

To use `cin` and `cout`, every C++ program must use the preprocessor directive:

```
#include <iostream>
```


NOTE

From Chapter 2, recall that you have been using the statement `using namespace std;` in addition to including the header file `iostream` to use `cin` and `cout`. Without the statement `using namespace std;`, you refer to these identifiers as `std::cin` and `std::cout`. In Chapter 8, you will learn about the meaning of the statement `using namespace std;` in detail.

3

Variables of type `istream` are called **input stream variables**; variables of type `ostream` are called **output stream variables**. A **stream variable** is either an input stream variable or an output stream variable.

Because `cin` and `cout` are already defined and have specific meanings, to avoid confusion, you should never redefine them in programs.

The variable `cin` has access to operators and functions that can be used to extract data from the standard input device. You have briefly used the extraction operator `>>` to input data from the standard input device. The next section describes in detail how the extraction operator `>>` works. In the following sections, you will learn how to use the functions `get`, `ignore`, `peek`, and `putback` to input data in a specific manner.

`cin` and the Extraction Operator `>>`

In Chapter 2, you saw how to input data from the standard input device by using `cin` and the extraction operator `>>`. Suppose `payRate` is a `double` variable. Consider the following C++ statement:

```
cin >> payRate;
```

When the computer executes this statement, it inputs the next number typed on the keyboard and stores this number in `payRate`. Therefore, if the user types `15.50`, the value stored in `payRate` is `15.50`.

The extraction operator `>>` is binary and thus takes two operands. The left-side operand must be an input stream variable, such as `cin`. Because the purpose of an input statement is to read and store values in a memory location and because only variables refer to memory locations, the right-side operand is a variable.

NOTE

The extraction operator `>>` is defined only for putting data into variables of simple data types. Therefore, the right-side operand of the extraction operator `>>` is a variable of the simple data type. However, C++ allows the programmer to extend the definition of the extraction operator `>>` so that data can also be put into other types of variables by using an input statement. You will learn this mechanism in the chapter entitled *Overloading and Templates*, later in this book.

The syntax of an input statement using `cin` and the extraction operator `>>` is:

```
cin >> variable >> variable...;
```

As you can see in the preceding syntax, a single input statement can read more than one data item by using the operator `>>` several times. Every occurrence of `>>` extracts the next data item from the input stream. For example, you can read both `payRate` and `hoursWorked` via a single input statement by using the following code:

```
cin >> payRate >> hoursWorked;
```

There is no difference between the preceding input statement and the following two input statements. Which form you use is a matter of convenience and style.

```
cin >> payRate;
cin >> hoursWorked;
```

How does the extraction operator `>>` work? When scanning for the next input, `>>` skips all whitespace characters. Recall that whitespace characters consist of blanks and certain nonprintable characters, such as tabs and the newline character. Thus, whether you separate the input data by lines or blanks, the extraction operator `>>` simply finds the next input data in the input stream. For example, suppose that `payRate` and `hoursWorked` are `double` variables. Consider the following input statement:

```
cin >> payRate >> hoursWorked;
```

Whether the input is:

```
15.50 48.30
```

or:

```
15.50  48.30
```

or:

```
15.50
48.30
```

the preceding input statement would store 15.50 in `payRate` and 48.30 in `hoursWorked`. Note that the first input is separated by a blank, the second input is separated by a tab, and the third input is separated by a line.

Now suppose that the input is 2. How does the extraction operator `>>` distinguish between the character 2 and the number 2? The right-side operand of the extraction operator `>>` makes this distinction. If the right-side operand is a variable of the data type `char`, the input 2 is treated as the character 2 and, in this case, the ASCII value of 2 is stored. If the right-side operand is a variable of the data type `int` or `double`, the input 2 is treated as the number 2.

Next, consider the input 25 and the statement:

```
cin >> a;
```

where `a` is a variable of some simple data type. If `a` is of the data type `char`, only the single character 2 is stored in `a`. If `a` is of the data type `int`, 25 is stored in `a`. If `a` is of the data type

double, the input 25 is converted to the decimal number 25.0. Table 3-1 summarizes this discussion by showing the valid input for a variable of the simple data type.

TABLE 3-1 Valid Input for a Variable of the Simple Data Type

Data Type of a	Valid Input for a
char	One printable character except the blank
int	An integer, possibly preceded by a + or – sign
double	A decimal number, possibly preceded by a + or – sign. If the actual data input is an integer, the input is converted to a decimal number with the zero decimal part.

3

When reading data into a **char** variable, after skipping any leading whitespace characters, the extraction operator >> finds and stores only the next character; reading stops after a single character. To read data into an **int** or **double** variable, after skipping all leading whitespace characters and reading the plus or minus sign (if any), the extraction operator >> reads the digits of the number, including the decimal point for floating-point variables, and stops when it finds a whitespace character or a character other than a digit.

EXAMPLE 3-1

Suppose you have the following variable declarations:

```
int a, b;
double z;
char ch;
```

The following statements show how the extraction operator >> works.

Statement	Input	Value Stored in Memory
1 cin >> ch;	A	ch = 'A'
2 cin >> ch;	AB	ch = 'A', 'B' is held for later input
3 cin >> a;	48	a = 48
4 cin >> a;	46.35	a = 46, .35 is held for later input
5 cin >> z;	74.35	z = 74.35
6 cin >> z;	39	z = 39.0
7 cin >> z >> a;	65.78 38	z = 65.78, a = 38

Statement	Input	Value Stored in Memory
8 cin >> a >> b;	4 60	a = 4, b = 60
9 cin >> a >> z;	46 32.4 68	a = 46, z = 32.4, 68 is held for later input

EXAMPLE 3-2

Suppose you have the following variable declarations:

```
int a;  
double z;  
char ch;
```

The following statements show how the extraction operator >> works.

Statement	Input	Value Stored in Memory
1 cin >> a >> ch >> z;	57 A 26.9	a = 57, ch = 'A', z = 26.9
2 cin >> a >> ch >> z;	57 A 26.9	a = 57, ch = 'A', z = 26.9
3 cin >> a >> ch >> z;	57 A 26.9	a = 57, ch = 'A', z = 26.9
4 cin >> a >> ch >> z;	57A26.9	a = 57, ch = 'A', z = 26.9

Note that for statements 1 through 4, the input statement is the same; however, the data is entered differently. For statement 1, data is entered on the same line separated by blanks. For statement 2, data is entered on two lines; the first two input values are separated by two blank spaces, and the third input is on the next line. For statement 3, all three input values are separated by lines, and for statement 4, all three input values are on the same line, but there is no space between them. Note that the second input is a non-numeric character. These statements work as follows.

Statements 1, 2, and 3 are easy to follow. Let us look at statement 4.

In statement 4, first the extraction operator >> extracts 57 from the input stream and stores it in **a**. Then, the extraction operator >> extracts the character 'A' from the input stream and stores it in **ch**. Next, 26.9 is extracted and stored in **z**.

Note that statements 1, 2, and 3 illustrate that regardless of whether the input is separated by blanks or by lines, the extraction operator >> always finds the next input.

EXAMPLE 3-3

Suppose you have the following variable declarations:

```
int a, b;
double z;
char ch, ch1, ch2;
```

The following statements show how the extraction operator >> works.

	Statement	Input	Value Stored in Memory
1	<code>cin >> z >> ch >> a;</code>	36.78B34	<code>z = 36.78</code> , <code>ch = 'B'</code> , <code>a = 34</code>
2	<code>cin >> z >> ch >> a;</code>	36.78 B34	<code>z = 36.78</code> , <code>ch = 'B'</code> , <code>a = 34</code>
3	<code>cin >> a >> b >> z;</code>	11 34	<code>a = 11</code> , <code>b = 34</code> , computer waits for the next number
4	<code>cin >> a >> z;</code>	78.49	<code>a = 78</code> , <code>z = 0.49</code>
5	<code>cin >> ch >> a;</code>	256	<code>ch = '2'</code> , <code>a = 56</code>
6	<code>cin >> a >> ch;</code>	256	<code>a = 256</code> , computer waits for the input value for <code>ch</code>
7	<code>cin >> ch1 >> ch2;</code>	A B	<code>ch1 = 'A'</code> , <code>ch2 = 'B'</code>

In statement 1, because the first right-side operand of >> is `z`, which is a `double` variable, `36.78` is extracted from the input stream, and the value `36.78` is stored in `z`. Next, `'B'` is extracted and stored in `ch`. Finally, `34` is extracted and stored in `a`. Statement 2 works similarly.

In statement 3, `11` is stored in `a`, and `34` is stored in `b`, but the input stream does not have enough input data to fill each variable. In this case, the computer waits (and waits, and waits...) for the next input to be entered. The computer does not continue to execute until the next value is entered.

In statement 4, the first right-side operand of the extraction operator >> is a variable of the type `int`, and the input is `78.49`. Now for `int` variables, after inputting the digits of the number, the reading stops at the first whitespace character or a character other than a digit. Therefore, the operator >> stores `78` into `a`. The next right-side operand of >> is the variable `z`, which is of the type `double`. Therefore, the operator >> stores the value `.49` as `0.49` into `z`.

In statement 5, the first right-side operand of the extraction operator >> is a `char` variable, so the first nonwhitespace character, `'2'`, is extracted from the input stream. The character `'2'` is stored in the variable `ch`. The next right-side operand of the extraction operator >> is an `int` variable, so the next input value, `56`, is extracted and stored in `a`.

In statement 6, the first right-side operator of the extraction operator `>>` is an `int` variable, so the first data item, `256`, is extracted from the input stream and stored in `a`. Now the computer waits for the next data item for the variable `ch`.

In statement 7, `'A'` is stored into `ch1`. The extraction operator `>>` then skips the blank, and `'B'` is stored in `ch2`.

NOTE

Recall that during program execution, when entering character data such as letters, you do not enter the single quotes around the character.

What happens if the input stream has more data items than required by the program? After the program terminates, any values left in the input stream are discarded. When you enter data for processing, the data values should correspond to the data types of the variables in the input statement. Recall that when entering a number for a `double` variable, it is not necessary for the input number to have a decimal part. If the input number is an integer and has no decimal part, it is converted to a decimal value. The computer, however, does not tolerate any other kind of mismatch. For example, entering a `char` value into an `int` or `double` variable causes serious errors, called **input failure**. Input failure is discussed later in this chapter.

The extraction operator, when scanning for the next input in the input stream, skips whitespace such as blanks and the newline character. However, there are situations when these characters must also be stored and processed. For example, if you are processing text in a line-by-line fashion, you must know where in the input stream the newline character is located. Without identifying the position of the newline character, the program would not know where one line ends and another begins. The next few sections teach you how to input data into a program using the input functions, such as `get`, `ignore`, `putback`, and `peek`. These functions are associated with the data type `istream` and are called **istream member functions**. I/O functions, such as `get`, are typically called **stream member functions** or **stream functions**.

Before you can learn about the input functions `get`, `ignore`, `putback`, `peek`, and other I/O functions that are used in this chapter, you need to first understand what a function is and how it works. You will study functions in detail and learn how to write your own in Chapters 6 and 7.

Using Predefined Functions in a Program

As noted in Chapter 2, a function, also called a subprogram, is a set of instructions. When a function executes, it accomplishes something. The function `main`, as you saw in Chapter 2, executes automatically when you run a program. Other functions execute

only when they are activated—that is, called. C++ comes with a wealth of functions, called **predefined functions**, that are already written. In this section, you will learn how to use some predefined functions that are provided as part of the C++ system. Later in this chapter, you will learn how to use stream functions to perform a specific I/O operation.

Recall from Chapter 2 that predefined functions are organized as a collection of libraries, called header files. A particular header file may contain several functions. Therefore, to use a particular function, you need to know the name of the function and a few other things, which are described shortly.

A very useful function, `pow`, called the power function, can be used to calculate x^y in a program. That is, `pow(x, y) = x^y` . For example, `pow(2.0, 3.0) = $2.0^{3.0} = 8.0$` and `pow(4.0, 0.5) = $4.0^{0.5} = \sqrt{4.0} = 2.0$` . The numbers `x` and `y` that you use in the function `pow` are called the **arguments** or **parameters** of the function `pow`. For example, in `pow(2.0, 3.0)`, the parameters are 2.0 and 3.0.

An expression such as `pow(2.0, 3.0)` is called a **function call**, which causes the code attached to the predefined function `pow` to execute and, in this case, computes $2.0^{3.0}$. The header file `cmath` contains the specification of the function `pow`.

To use a predefined function in a program, you need to know the name of the header file containing the specification of the function and include that header file in the program. In addition, you need to know the name of the function, the number of parameters the function takes, and the type of each parameter. You must also be aware of what the function is going to do. For example, to use the function `pow`, you must include the header file `cmath`. The function `pow` has two parameters, which are decimal numbers. The function calculates the first parameter to the power of the second parameter. (Appendix F describes some commonly used header files and predefined functions.)

The program in the following example illustrates how to use predefined functions in a program. More specifically, we use some math functions, from the header file `cmath`, and the `string` function `length`, from the header file `string`. Note that the function `length` determines the length of a `string`.

EXAMPLE 3-4

```
// How to use predefined functions.
#include <iostream>
#include <cmath>
#include <string>

using namespace std;

int main()
{
    double u, v;
    string str;
```

```

    cout << "Line 1: 2 to the power of 6 = "
         << static_cast<int>(pow(2.0, 6.0))
         << endl;                                     //Line 1

    u = 12.5;                                         //Line 2
    v = 3.0;                                         //Line 3
    cout << "Line 4: " << u << " to the power of "
         << v << " = " << pow(u, v) << endl;       //Line 4

    cout << "Line 5: Square root of 24 = "
         << sqrt(24.0) << endl;                     //Line 5

    u = pow(8.0, 2.5);                               //Line 6
    cout << "Line 7: u = " << u << endl;           //Line 7

    str = "Programming with C++";                   //Line 8

    cout << "Line 9: Length of str = "
         << str.length() << endl;                 //Line 9

    return 0;
}

```

Sample Run:

```

Line 1: 2 to the power of 6 = 64
Line 4: 12.5 to the power of 3 = 1953.13
Line 5: Square root of 24 = 4.89898
Line 7: u = 181.019
Line 9: Length of str = 20

```

The preceding program works as follows. The statement in Line 1 uses the function `pow` to determine and output 2^6 . The statement in Line 2 sets `u` to 12.5, and the statement in Line 3 sets `v` to 3.0. The statement in Line 4 determines and outputs u^v . The statement in Line 5 uses the function `sqrt`, of the header file `cmath`, to determine and output the square root of 24.0. The statement in Line 6 determines and assigns $8.0^{2.5}$ to `u`. The statement in Line 7 outputs the value of `u`.

The statement in Line 8 stores the string "Programming with C++" in `str`. The statement in Line 9 uses the string function `length` to determine and output the length of `str`. Note how the function `length` is used. Later in this chapter, we explain the meaning of expressions such as `str.length()`.

Because I/O is fundamental to any programming language and because writing instructions to perform a specific I/O operation is not a job for everyone, every programming language provides a set of useful functions to perform specific I/O operations. In the remainder of this chapter, you will learn how to use some of these functions in a program. As a programmer, you must pay close attention to how these functions are used so that you can get the most out of them. The first function you will learn about here is the function `get`.

cin and the get Function

As you have seen, the extraction operator skips all leading whitespace characters when scanning for the next input value. Consider the variable declarations:

```
char ch1, ch2;
int num;
```

and the input:

A 25

Now consider the following statement:

```
cin >> ch1 >> ch2 >> num;
```

When the computer executes this statement, 'A' is stored in `ch1`, the blank is skipped by the extraction operator `>>`, the character '2' is stored in `ch2`, and 5 is stored in `num`. However, what if you intended to store 'A' in `ch1`, the blank in `ch2`, and 25 in `num`? It is clear that you cannot use the extraction operator `>>` to input this data.

As stated earlier, sometimes you need to process the entire input, including whitespace characters, such as blanks and the newline character. For example, suppose you want to process the entered data on a line-by-line basis. Because the extraction operator `>>` skips the newline character and unless the program captures the newline character, the computer does not know where one line ends and the next begins.

The variable `cin` can access the stream function `get`, which is used to read character data. The `get` function inputs the very next character, including whitespace characters, from the input stream and stores it in the memory location indicated by its argument. The function `get` comes in many forms. Next, we discuss the one that is used to read a character.

The syntax of `cin`, together with the `get` function to read a character, follows:

```
cin.get(varChar);
```

In the `cin.get` statement, `varChar` is a `char` variable. `varChar`, which appears in parentheses following the function name, is called the **argument** or **parameter** of the function. The effect of the preceding statement would be to store the next input character in the variable `varChar`.

Now consider the following input again:

A 25

To store 'A' in `ch1`, the blank in `ch2`, and 25 in `num`, you can effectively use the `get` function as follows:

```
cin.get(ch1);
cin.get(ch2);
cin >> num;
```

Because this form of the `get` function has only one argument and reads only one character and you need to read two characters from the input stream, you need to call this function twice. Notice that you cannot use the `get` function to read data into the variable `num` because `num` is an `int` variable. The preceding form of the `get` function reads values of only the `char` data type.

The preceding set of `cin.get` statements is equivalent to the following statements:

```
cin >> ch1;  
cin.get(ch2);  
cin >> num;
```

NOTE

The function `get` has other forms, one of which you will study in Chapter 9. For the next few chapters, you need only the form of the function `get` introduced here.

`cin` and the `ignore` Function

When you want to process only partial data (say, within a line), you can use the stream function `ignore` to discard a portion of the input. The syntax to use the function `ignore` is:

```
cin.ignore(intExp, chExp);
```

Here, `intExp` is an integer expression yielding an integer value, and `chExp` is a `char` expression yielding a `char` value. In fact, the value of the expression `intExp` specifies the maximum number of characters to be ignored in a line.

Suppose `intExp` yields a value of, say 100. This statement says to ignore the next 100 characters or ignore the input until it encounters the character specified by `chExp`, whichever comes first. To be specific, consider the following statement:

```
cin.ignore(100, '\n');
```

When this statement executes, it ignores either the next 100 characters or all characters until the newline character is found, whichever comes first. For example, if the next 120 characters do not contain the newline character, then only the first 100 characters are discarded and the next input data is the character 101. However, if the 75th character is the newline character, then the first 75 characters are discarded and the next input data is the 76th character. Similarly, the execution of the statement:

```
cin.ignore(100, 'A');
```

results in ignoring the first 100 characters or all characters until the character 'A' is found, whichever comes first.

EXAMPLE 3-5

Consider the declaration:

```
int a, b;
```

and the input:

```
25 67 89 43 72
12 78 34
```

Now consider the following statements:

```
cin >> a;
cin.ignore(100, '\n');
cin >> b;
```

The first statement, `cin >> a;`, stores 25 in `a`. The second statement, `cin.ignore(100, '\n');`, discards all of the remaining numbers in the first line. The third statement, `cin >> b;`, stores 12 (from the next line) in `b`.

EXAMPLE 3-6

Consider the declaration:

```
char ch1, ch2;
```

and the input:

```
Hello there. My name is Mickey.
```

- a. Consider the following statements:

```
cin >> ch1;
cin.ignore(100, '.');
cin >> ch2;
```

The first statement, `cin >> ch1;`, stores 'H' in `ch1`. The second statement, `cin.ignore(100, '.');`, results in discarding all characters until . (period). The third statement, `cin >> ch2;`, stores the character 'M' (from the same line) in `ch2`. (Remember that the extraction operator `>>` skips all leading whitespace characters. Thus, the extraction operator skips the space after . [period] and stores 'M' in `ch2`.)

- b. Suppose that we have the following statement:

```
cin >> ch1;
cin.ignore(5, '.');
cin >> ch2;
```

The first statement, `cin >> ch1;`, stores 'H' in `ch1`. The second statement, `cin.ignore(5, '.');`, results in discarding the next five characters, that is, until `t`. The third statement, `cin >> ch2;`, stores the character 't' (from the same line) in `ch2`.

When the function `ignore` is used without any arguments, then it only skips the very next character. For example, the following statement will skip the very next character:

```
cin.ignore();
```

This statement is typically used to skip the newline character.

The `putback` and `peek` Functions

Suppose you are processing data that is a mixture of numbers and characters. Moreover, the numbers must be read and processed as numbers. You have also looked at many sets of sample data and cannot determine whether the next input is a character or a number. You could read the entire data set character by character and check whether a certain character is a digit. If a digit is found, you could then read the remaining digits of the number and somehow convert these characters into numbers. This programming code would be somewhat complex. Fortunately, C++ provides two very useful stream functions that can be used effectively in these types of situations.

The stream function `putback` lets you put the last character extracted from the input stream by the `get` function back into the input stream. The stream function `peek` looks into the input stream and tells you what the next character is without removing it from the input stream. By using these functions, after determining that the next input is a number, you can read it as a number. You do not have to read the digits of the number as characters and then convert these characters to that number.

The syntax to use the function `putback` is:

```
istreamVar.putback(ch);
```

Here, `istreamVar` is an input stream variable, such as `cin`, and `ch` is a `char` variable.

The `peek` function returns the next character from the input stream but does not remove the character from that stream. In other words, the function `peek` looks into the input stream and checks the identity of the next input character. Moreover, after checking the next input character in the input stream, it can store this character in a designated memory location without removing it from the input stream. That is, when you use the `peek` function, the next input character stays the same, even though you now know what it is.

The syntax to use the function `peek` is:

```
ch = istreamVar.peek();
```

Here, `istreamVar` is an input stream variable, such as `cin`, and `ch` is a `char` variable.

Notice how the function `peek` is used. First, the function `peek` is used in an assignment statement. It is not a stand-alone statement like `get`, `ignore`, and `putback`. Second, the function `peek` has empty parentheses. Until you become comfortable with using a function and learn how to write one, pay close attention to how to use a predefined function.

The following example illustrates how to use the `peek` and `putback` functions.

EXAMPLE 3-7

//Functions peek and putback

```
#include <iostream>

using namespace std;

int main()
{
    char ch;

    cout << "Line 1: Enter a string: ";           //Line 1
    cin.get(ch);                                 //Line 2
    cout << endl;                                //Line 3
    cout << "Line 4: After first cin.get(ch); "
         << "ch = " << ch << endl;               //Line 4

    cin.get(ch);                                 //Line 5
    cout << "Line 6: After second cin.get(ch); "
         << "ch = " << ch << endl;               //Line 6

    cin.putback(ch);                             //Line 7
    cin.get(ch);                                 //Line 8
    cout << "Line 9: After putback and then "
         << "cin.get(ch); ch = " << ch << endl;   //Line 9

    ch = cin.peek();                             //Line 10
    cout << "Line 11: After cin.peek(); ch = "
         << ch << endl;                           //Line 11

    cin.get(ch);                                 //Line 12
    cout << "Line 13: After cin.get(ch); ch = "
         << ch << endl;                           //Line 13

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

Line 1: Enter a string: **abcd**

Line 4: After first `cin.get(ch)`; `ch = a`

Line 6: After second `cin.get(ch)`; `ch = b`

Line 9: After `putback` and then `cin.get(ch); ch = b`
 Line 11: After `cin.peek(); ch = c`
 Line 13: After `cin.get(ch); ch = c`

The user input, `abcd`, allows you to see the effect of the functions `get`, `putback`, and `peek` in the preceding program. The statement in Line 1 prompts the user to enter a string. In Line 2, the statement `cin.get(ch);` extracts the first character from the input stream and stores it in the variable `ch`. So after Line 2 executes, the value of `ch` is `'a'`.

The `cout` statement in Line 4 outputs the value of `ch`. The statement `cin.get(ch);` in Line 5 extracts the next character from the input stream, which is `'b'`, and stores it in `ch`. At this point, the value of `ch` is `'b'`.

The `cout` statement in Line 6 outputs the value of `ch`. The `cin.putback(ch);` statement in Line 7 puts the previous character extracted by the `get` function, which is `'b'`, back into the input stream. Therefore, the next character to be extracted from the input stream is `'b'`.

The `cin.get(ch);` statement in Line 8 extracts the next character from the input stream, which is still `'b'`, and stores it in `ch`. Now the value of `ch` is `'b'`. The `cout` statement in Line 9 outputs the value of `ch` as `'b'`.

In Line 10, the statement `ch = cin.peek();` checks the next character in the input stream, which is `'c'`, and stores it in `ch`. The value of `ch` is now `'c'`. The `cout` statement in Line 11 outputs the value of `ch`. The `cin.get(ch);` statement in Line 12 extracts the next character from the input stream and stores it in `ch`. The `cout` statement in Line 13 outputs the value of `ch`, which is still `'c'`.

Note that the statement `ch = cin.peek();` in Line 10 did not remove the character `'c'` from the input stream; it only peeked into the input stream. The output of Lines 11 and 13 demonstrates this functionality.

The Dot Notation between I/O Stream Variables and I/O Functions: A Precaution

In the preceding sections, you learned how to manipulate an input stream to get data into a program. You also learned how to use the functions `get`, `ignore`, `peek`, and `putback`. It is important that you use these functions exactly as shown. For example, to use the `get` function, you used statements such as the following:

```
cin.get(ch);
```

Omitting the dot—that is, the period between the variable `cin` and the function name `get`—results in a syntax error. For example, in the statement:

```
cin.get(ch);
```

`cin` and `get` are two separate identifiers separated by a dot. In the statement:

```
cinget(ch);
```

`cin.get` becomes a new identifier. If you used `cin.get(ch);` in a program, the compiler would try to resolve an undeclared identifier, which would generate an error. Similarly, missing parentheses, as in `cin.getch;`, result in a syntax error. Also, remember that you must use the input functions together with an input stream variable. If you try to use any of the input functions alone—that is, without the input stream variable—the compiler might generate an error message such as “undeclared identifier.” For example, the statement `get(ch);` could result in a syntax error.

As you can see, several functions are associated with an `istream` variable, each doing a specific job. Recall that the functions `get`, `ignore`, and so on are *members* of the data type `istream`. Called the **dot notation**, the dot separates the input stream variable name from the member, or function, name. In fact, in C++, the dot is an operator called the **member access operator**.

NOTE

C++ has a special name for the data types `istream` and `ostream`. The data types `istream` and `ostream` are called classes. The variables `cin` and `cout` also have special names, called objects. Therefore, `cin` is called an `istream` object, and `cout` is called an `ostream` object. In fact, stream variables are called stream objects. You will learn these concepts in the chapter entitled Inheritance and Composition later in this book.

Input Failure

Many things can go wrong during program execution. A program that is syntactically correct might produce incorrect results. For example, suppose that a part-time employee’s paycheck is calculated by using the following formula:

```
wages = payRate * hoursWorked;
```

If you accidentally type `+` in place of `*`, the calculated wages would be incorrect, even though the statement containing a `+` is syntactically correct.

What about an attempt to read invalid data? For example, what would happen if you tried to input a letter into an `int` variable? If the input data did not match the corresponding variables, the program would run into problems. For example, trying to read a letter into an `int` or `double` variable would result in an **input failure**. Consider the following statements:

```
int a, b, c;
double x;
```

If the input is:

```
W 54
```

then the statement:

```
cin >> a >> b;
```

would result in an input failure, because you are trying to input the character 'w' into the `int` variable `a`. If the input were:

```
35 67.93 48
```

then the input statement:

```
cin >> a >> x >> b;
```

would result in storing 35 in `a`, 67.93 in `x`, and 48 in `b`.

Now consider the following read statement with the previous input (the input with three values):

```
cin >> a >> b >> c;
```

This statement stores 35 in `a` and 67 in `b`. The reading stops at `.` (the decimal point). Because the next variable `c` is of the data type `int`, the computer tries to read `.` into `c`, which is an error. The input stream then enters a state called the **fail state**.

What actually happens when the input stream enters the fail state? Once an input stream enters the fail state, all further I/O statements using that stream are ignored. Unfortunately, the program quietly continues to execute with whatever values are stored in variables and produces incorrect results. The program in Example 3-8 illustrates an input failure. This program on your system may produce different results.

EXAMPLE 3-8

//Input Failure program

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
    int a = 10;           //Line 1
    int b = 20;           //Line 2
    int c = 30;           //Line 3
    int d = 40;           //Line 4

    cout << "Line 5: Enter four integers: "; //Line 5
    cin >> a >> b >> c >> d; //Line 6
    cout << endl;          //Line 7
    cout << "Line 8: The numbers you entered are:"
         << endl;          //Line 8
    cout << "Line 9: a = " << a << ", b = " << b
         << ", c = " << c << ", d = " << d << endl; //Line 9

    return 0;
}
```


Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1

Line 5: Enter four integers: 34 K 67 28

Line 8: The numbers you entered are:

Line 9: a = 34, b = 20, c = 30, d = 40

The statements in Lines 1, 2, 3, and 4 declare and initialize the variables **a**, **b**, **c**, and **d** to 10, 20, 30, and 40, respectively. The statement in Line 5 prompts the user to enter four integers; the statement in Line 6 inputs these four integers into variables **a**, **b**, **c**, and **d**.

In this sample run, the second input value is the character '**K**'. The **cin** statement tries to input this character into the variable **b**. However, because **b** is an **int** variable, the input stream enters the fail state. Note that the values of **b**, **c**, and **d** are unchanged, as shown by the output of the statement in Line 9.

Sample Run 2

Line 5: Enter four integers: 37 653.89 23 76

Line 8: The numbers you entered are:

Line 9: a = 37, b = 653, c = 30, d = 40

In this sample run, the **cin** statement in Line 6 inputs 37 into **a** and 653 into **b** and then tries to input the decimal point into **c**. Because **c** is an **int** variable, the decimal point is regarded as a character, so the input stream enters the fail state. In this sample run, the values of **c** and **d** are unchanged, as shown by the output of the statement in Line 9.

The **clear** Function

When an input stream enters the fail state, the system ignores all further I/O using that stream. You can use the stream function **clear** to restore the input stream to a working state.

The syntax to use the function **clear** is:

```
istreamVar.clear();
```

Here, **istreamVar** is an input stream variable, such as **cin**.

After using the function **clear** to return the input stream to a working state, you still need to clear the rest of the garbage from the input stream. This can be accomplished by using the function **ignore**. Example 3-9 illustrates this situation.

EXAMPLE 3-9**//Input failure and the clear function**

```

#include <iostream>

using namespace std;

int main()
{
    int a = 23;                                //Line 1
    int b = 34;                                //Line 2

    cout << "Line 3: Enter a number followed"
         << " by a character: ";                //Line 3
    cin >> a >> b;                               //Line 4
    cout << endl << "Line 5: a = " << a
         << ", b = " << b << endl;              //Line 5

    cin.clear();                                //Restore input stream; Line 6

    cin.ignore(200, '\n');                      //Clear the buffer; Line 7

    cout << "Line 8: Enter two numbers: ";        //Line 8
    cin >> a >> b;                               //Line 9
    cout << endl << "Line 10: a = " << a
         << ", b = " << b << endl;              //Line 10

    return 0;
}

```

Sample Run: In this sample run, the user input is shaded.

Line 3: Enter a number followed by a character: 78 d

Line 5: a = 78, b = 34

Line 8: Enter two numbers: 65 88

Line 10: a = 65, b = 88

The statements in Lines 1 and 2 declare and initialize the variables **a** and **b** to 23 and 34, respectively. The statement in Line 3 prompts the user to enter a number followed by a character; the statement in Line 4 inputs this number into the variable **a** and then tries to input the character into the variable **b**. Because **b** is an **int** variable, an attempt to input a character into **b** causes the input stream to enter the fail state. The value of **b** is unchanged, as shown by the output of the statement in Line 5.

The statement in Line 6 restores the input stream by using the function **clear**, and the statement in Line 7 ignores the rest of the input. The statement in Line 8 again prompts the user to input two numbers; the statement in Line 9 stores these two numbers into **a** and **b**. Next, the statement in Line 10 outputs the values of **a** and **b**.

Output and Formatting Output

Other than writing efficient programs, generating the desired output is one of a programmer's highest priorities. Chapter 2 briefly introduced the process involved in generating output on the standard output device. More precisely, you learned how to use the insertion operator `<<` and the manipulator `endl` to display results on the standard output device.

However, there is a lot more to output than just displaying results. Sometimes, floating-point numbers must be output in a specific way. For example, a paycheck must be printed to two decimal places, whereas the results of a scientific experiment might require the output of floating-point numbers to six, seven, or perhaps even ten decimal places. Also, you might like to align the numbers in specific columns or fill the empty space between strings and numbers with a character other than the blank. For example, in preparing the table of contents, the space between the section heading and the page number might need to be filled with dots or dashes. In this section, you will learn about various output functions and manipulators that allow you to format your output in a desired way.

Recall that the syntax of `cout` when used together with the insertion operator `<<` is:

```
cout << expression or manipulator << expression or manipulator...;
```

Here, **expression** is evaluated, its value is printed, and **manipulator** is used to format the output. The simplest manipulator that you have used so far is `endl`, which is used to move the insertion point to the beginning of the next line.

Other output manipulators that are of interest include `setprecision`, `fixed`, `showpoint`, and `setw`. The next few sections describe these manipulators.

`setprecision` Manipulator

You use the manipulator `setprecision` to control the output of floating-point numbers. Usually, the default output of floating-point numbers is scientific notation. Some integrated development environments (IDEs) might use a maximum of six decimal places for the default output of floating-point numbers. However, when an employee's paycheck is printed, the desired output is a maximum of two decimal places. To print floating-point output to two decimal places, you use the `setprecision` manipulator to set the precision to 2.

The general syntax of the `setprecision` manipulator is:

```
setprecision(n)
```

where **n** is the number of decimal places.

You use the `setprecision` manipulator with `cout` and the insertion operator. For example, the statement:

```
cout << setprecision(2);
```

formats the output of decimal numbers to two decimal places until a similar subsequent statement changes the precision. Notice that the number of decimal places, or the precision value, is passed as an argument to `setprecision`.

To use the manipulator `setprecision`, the program must include the header file `iomanip`. Thus, the following include statement is required:

```
#include <iomanip>
```

fixed Manipulator

To further control the output of floating-point numbers, you can use other manipulators. To output floating-point numbers in a fixed decimal format, you use the manipulator `fixed`. The following statement sets the output of floating-point numbers in a fixed decimal format on the standard output device:

```
cout << fixed;
```

After the preceding statement executes, all floating-point numbers are displayed in the fixed decimal format until the manipulator `fixed` is disabled. You can disable the manipulator `fixed` by using the stream member function `unsetf`. For example, to disable the manipulator `fixed` on the standard output device, you use the following statement:

```
cout.unsetf(ios::fixed);
```

After the manipulator `fixed` is disabled, the output of the floating-point numbers returns to their default settings. The manipulator `scientific` is used to output floating-point numbers in scientific format.

NOTE

On some compilers, the statements `cin >> fixed;` and `cin >> scientific;` might not work. In this case, you can use `cin.setf(ios::fixed);` in place of `cin >> fixed;` and `cin.setf(ios::scientific);` in place of `cin >> scientific;`.

The following example shows how the manipulators `scientific` and `fixed` work without using the manipulator `setprecision`.

EXAMPLE 3-10

```
//Example: scientific and fixed
```

```
#include <iostream>
```

```
using namespace std;
```

```

int main()
{
    double hours = 35.45;
    double rate = 15.00;
    double tolerance = 0.01000;

    cout << "hours = " << hours << ", rate = " << rate
        << ", pay = " << hours * rate
        << ", tolerance = " << tolerance << endl << endl;

    cout << scientific;
    cout << "Scientific notation: " << endl;
    cout << "hours = " << hours << ", rate = " << rate
        << ", pay = " << hours * rate
        << ", tolerance = " << tolerance << endl << endl;

    cout << fixed;
    cout << "Fixed decimal notation: " << endl;
    cout << "hours = " << hours << ", rate = " << rate
        << ", pay = " << hours * rate
        << ", tolerance = " << tolerance << endl << endl;

    return 0;
}

```

Sample Run:

```
hours = 35.45, rate = 15, pay = 531.75, tolerance = 0.01
```

Scientific notation:

```
hours = 3.545000e+001, rate = 1.500000e+001, pay = 5.317500e+002, tolerance = 1
.000000e-002
```

Fixed decimal notation:

```
hours = 35.450000, rate = 15.000000, pay = 531.750000, tolerance = 0.010000
```

The sample run shows that when the value of **rate** and **tolerance** are printed without setting the **scientific** or **fixed** manipulators, the trailing zeros are not shown and, in the case of **rate**, the decimal point is also not shown. After setting the manipulators, the values are printed to six decimal places. In the next section, we describe the manipulator **showpoint** to force the system to show the decimal point and trailing zeros. We will then give an example to show how to use the manipulators **setprecision**, **fixed**, and **showpoint** to get the desired output.

showpoint Manipulator

Suppose that the decimal part of a decimal number is zero. In this case, when you instruct the computer to output the decimal number in a fixed decimal format, the output may not show the decimal point and the decimal part. To force the output to show the decimal point and

trailing zeros, you use the manipulator `showpoint`. The following statement sets the output of decimal numbers with a decimal point and trailing zeros on the standard input device:

```
cout << showpoint;
```

Of course, the following statement sets the output of a floating-point number in a fixed decimal format with the decimal point and trailing zeros on the standard output device:

```
cout << fixed << showpoint;
```

The program in Example 3-11 illustrates how to use the manipulators `setprecision`, `fixed`, and `showpoint`.

EXAMPLE 3-11

//Example: setprecision, fixed, showpoint

```
#include <iostream>                                //Line 1
#include <iomanip>                                    //Line 2

using namespace std;                                //Line 3

const double PI = 3.14159265;                        //Line 4

int main()                                           //Line 5
{                                                     //Line 6
    double radius = 12.67;                           //Line 7
    double height = 12.00;                           //Line 8

    cout << fixed << showpoint;                       //Line 9

    cout << setprecision(2)
        << "Line 10: setprecision(2)" << endl;        //Line 10
    cout << "Line 11: radius = " << radius << endl;    //Line 11
    cout << "Line 12: height = " << height << endl;    //Line 12
    cout << "Line 13: volume = "
        << PI * radius * radius * height << endl;    //Line 13
    cout << "Line 14: PI = " << PI << endl << endl;    //Line 14

    cout << setprecision(3)
        << "Line 15: setprecision(3)" << endl;        //Line 15
    cout << "Line 16: radius = " << radius << endl;    //Line 16
    cout << "Line 17: height = " << height << endl;    //Line 17
    cout << "Line 18: volume = "
        << PI * radius * radius * height << endl;    //Line 18
    cout << "Line 19: PI = " << PI << endl << endl;    //Line 19

    cout << setprecision(4)
        << "Line 20: setprecision(4)" << endl;        //Line 20
    cout << "Line 21: radius = " << radius << endl;    //Line 21
    cout << "Line 22: height = " << height << endl;    //Line 22
```

```

cout << "Line 23: volume = "
    << PI * radius * radius * height << endl;    //Line 23
cout << "Line 24: PI = " << PI << endl << endl;    //Line 24

cout << "Line 25: "
    << setprecision(3) << radius << ", "
    << setprecision(2) << height << ", "
    << setprecision(5) << PI << endl;            //Line 25

return 0;                                         //Line 26
}                                                 //Line 27

```

Sample Run:

```

Line 10: setprecision(2)
Line 11: radius = 12.67
Line 12: height = 12.00
Line 13: volume = 6051.80
Line 14: PI = 3.14

Line 15: setprecision(3)
Line 16: radius = 12.670
Line 17: height = 12.000
Line 18: volume = 6051.797
Line 19: PI = 3.142

Line 20: setprecision(4)
Line 21: radius = 12.6700
Line 22: height = 12.0000
Line 23: volume = 6051.7969
Line 24: PI = 3.1416

Line 25: 12.670, 12.00, 3.14159

```

In this program, the statement in Line 2 includes the header file `iomanip`, and the statement in Line 4 declares the named constant `PI` and sets the value to eight decimal places. The statements in Lines 7 and 8 declare and initialize the variables `radius` and `height` to store the radius of the base and the height of a cylinder. The statement in Line 10 sets the output of floating-point numbers in a fixed decimal format with a decimal point and trailing zeros.

The statements in Lines 11, 12, 13, and 14 output the values of `radius`, `height`, the volume, and `PI` to two decimal places.

The statements in Lines 16, 17, 18, and 19 output the values of `radius`, `height`, the volume, and `PI` to three decimal places.

The statements in Lines 21, 22, 23, and 24 output the values of `radius`, `height`, the volume, and `PI` to four decimal places.

The statement in Line 25 outputs the value of `radius` to three decimal places, the value of `height` to two decimal places, and the value of `PI` to five decimal places.

Notice how the values of **radius** are printed in Lines 11, 16, and 21. The value of **radius** printed in Line 16 contains a trailing 0. This is because the stored value of **radius** has only two decimal places; a 0 is printed at the third decimal place. In a similar manner, the value of **height** is printed in Lines 12, 17, and 22.

Also, notice how the statements in Lines 13, 18, and 23 calculate and output the volume to two, three, and four decimal places.

Note that the value of **PI** printed in Line 24 is rounded.

The statement in Line 25 first sets the output of floating-point numbers to three decimal places and then outputs the value of **radius** to three decimal places. After printing the value of **radius**, the statement in Line 25 sets the output of floating-point numbers to two decimal places and then outputs the value of **height** to two decimal places. Next, it sets the output of floating-point numbers to five decimal places and then outputs the value of **PI** to four decimal places.

If you omit the statement in Line 9 and recompile and run the program, you will see the default output of the decimal numbers. More specifically, the value of the expression that calculates the volume might be printed in the scientific notation.

setw

The manipulator **setw** is used to output the value of an expression in a specific number of columns. The value of the expression can be either a string or a number. The expression **setw(n)** outputs the value of the next expression in **n** columns. The output is right-justified. Thus, if you specify the number of columns to be 8, for example, and the output requires only four columns, the first four columns are left blank. Furthermore, if the number of columns specified is less than the number of columns required by the output, the output automatically expands to the required number of columns; the output is not truncated. For example, if **x** is an **int** variable, the following statement outputs the value of **x** in five columns on the standard output device:

```
cout << setw(5) << x << endl;
```

To use the manipulator **setw**, the program must include the header file **iomanip**. Thus, the following **include** statement is required:

```
#include <iomanip>
```

Unlike **setprecision**, which controls the output of all floating-point numbers until it is reset, **setw** controls the output of only the next expression.

EXAMPLE 3-12

```
//Example: setw
```

```
#include <iostream>
#include <iomanip>
```



```

using namespace std;

int main()
{
    int x = 19;           //Line 1
    int a = 345;          //Line 2
    double y = 76.384;    //Line 3

    cout << fixed << showpoint; //Line 4

    cout << "12345678901234567890" << endl; //Line 5

    cout << setw(5) << x << endl; //Line 6
    cout << setw(5) << a << setw(5) << "Hi"
        << setw(5) << x << endl << endl; //Line 7

    cout << setprecision(2); //Line 8
    cout << setw(6) << a << setw(6) << y
        << setw(6) << x << endl; //Line 9
    cout << setw(6) << x << setw(6) << a
        << setw(6) << y << endl << endl; //Line 10

    cout << setw(5) << a << x << endl; //Line 11
    cout << setw(2) << a << setw(4) << x << endl; //Line 12

    return 0;
}

```

Sample Run:

```

12345678901234567890
  19
345   Hi   19

  345 76.38   19
  19   345 76.38

34519
345 19

```

The statements in Lines 1, 2, and 3 declare the variables **x**, **a**, and **y** and initialize these variables to 19, 345, and 76.384, respectively. The statement in Line 4 sets the output of floating-point numbers in a fixed decimal format with a decimal point and trailing zeros. The output of the statement in Line 5 shows the column positions when the specific values are printed; it is the first line of output.

The statement in Line 6 outputs the value of **x** in five columns. Because **x** has only two digits, only two columns are needed to output its value. Therefore, the first three columns are left blank in the second line of output. The statement in Line 7 outputs the value of **a** in the first five columns, the string "Hi" in the next five columns, and then the value of **x** in the following five columns. Because the string "Hi" contains only two characters and five columns are set to output these two characters, the first three columns are left blank. See

the third line of output. The fourth line of output is blank because the manipulator `endl` appears twice in the statement in Line 7.

The statement in Line 8 sets the output of floating-point numbers to two decimal places. The statement in Line 9 outputs the values of `a` in the first six columns, `y` in the next six columns, and `x` in the following six columns, creating the fifth line of output. The output of the statement in Line 10 (which is the sixth line of output) is similar to the output of the statement in Line 9. Notice how the numbers are nicely aligned in the outputs of the statements in Lines 9 and 10. The seventh line of output is blank because the manipulator `endl` appears twice in the statement in Line 10.

The statement in Line 11 outputs first the value of `a` in five columns and then the value of `x`. Note that the manipulator `setw` in the statement in Line 11 controls only the output of `a`. Thus, after the value of `a` is printed, the value of `x` is printed at the current cursor position (see the eighth line of output).

In the `cout` statement in Line 12, only two columns are assigned to output the value of `a`. However, the value of `a` has three digits, so the output is expanded to three columns. The value of `x` is then printed in four columns. Because the value of `x` contains only two digits, only two columns are required to output the value of `x`. Therefore, because four columns are allocated to output the value of `x`, the first two columns are left blank (see the ninth line of output).

Additional Output Formatting Tools

In the previous section, you learned how to use the manipulators `setprecision`, `fixed`, and `showpoint` to control the output of floating-point numbers and how to use the manipulator `setw` to display the output in specific columns. Even though these manipulators are adequate to produce an elegant report, in some situations, you may want to do more. In this section, you will learn additional formatting tools that give you more control over your output.

`setfill` Manipulator

Recall that in the manipulator `setw`, if the number of columns specified exceeds the number of columns required by the expression, the output of the expression is right-justified and the unused columns to the left are filled with spaces. The output stream variables can use the manipulator `setfill` to fill the unused columns with a character other than a space.

The syntax to use the manipulator `setfill` is:

```
ostreamVar << setfill(ch);
```

where `ostreamVar` is an output stream variable and `ch` is a character. For example, the statement:

```
cout << setfill('#');
```

sets the fill character to '#' on the standard output device.

To use the manipulator `setfill`, the program must include the header file `iomanip`.

The program in Example 3-13 illustrates the effect of using `setfill` in a program.

EXAMPLE 3-13

```
//Example: setfill

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int x = 15;           //Line 1
    int y = 7634;         //Line 2

    cout << "12345678901234567890" << endl; //Line 3
    cout << setw(5) << x << setw(7) << y
        << setw(8) << "Warm" << endl;      //Line 4

    cout << setfill('*'); //Line 5
    cout << setw(5) << x << setw(7) << y
        << setw(8) << "Warm" << endl;      //Line 6

    cout << setw(5) << x << setw(7) << setfill('#')
        << y << setw(8) << "Warm" << endl;  //Line 7

    cout << setw(5) << setfill('@') << x
        << setw(7) << setfill('#') << y
        << setw(8) << setfill('^') << "Warm"
        << endl;                          //Line 8
    cout << setfill(' '); //Line 9
    cout << setw(5) << x << setw(7) << y
        << setw(8) << "Warm" << endl;      //Line 10

    return 0;
}
```

Sample Run:

```
12345678901234567890
  15   7634   Warm
***15***7634***Warm
***15###7634###Warm
@@@15###7634^^^Warm
  15   7634   Warm
```

The statements in Lines 1 and 2 declare and initialize the variables `x` and `y` to 15 and 7634, respectively. The output of the statement in Line 3—the first line of output—shows the

column position when the subsequent statements output the values of the variables. The statement in Line 4 outputs the value of **x** in five columns, the value of **y** in seven columns, and the string "Warm" in eight columns. In this statement, the filling character is the blank character, as shown in the second line of output.

The statement in Line 5 sets the filling character to *. The statement in Line 6 outputs the value of **x** in five columns, the value of **y** in seven columns, and the string "Warm" in eight columns. Because **x** is a two-digit number and five columns are assigned to output its value, the first three columns are unused by **x** and are, therefore, filled by the filling character *. To print the value of **y**, seven columns are assigned; **y** is a four-digit number, however, so the filling character fills the first three columns. Similarly, to print the value of the string "Warm", eight columns are assigned; the string "Warm" has only four characters, so the filling character fills the first four columns. See the third line of output.

The output of the statement in Line 7—the fourth line of output—is similar to the output of the statement in Line 6, except that the filling character for **y** and the string "Warm" is #. In the output of the statement in Line 8 (the fifth line of output), the filling character for **x** is @, the filling character for **y** is #, and the filling character for the string "Warm" is ^. The manipulator `setfill` sets these filling characters.

The statement in Line 9 sets the filling character to blank. The statement in Line 10 outputs the values of **x**, **y**, and the string "Warm" using the filling character blank, as shown in the sixth line of output.

left and right Manipulators

Recall that if the number of columns specified in the `setw` manipulator exceeds the number of columns required by the next expression, the default output is right-justified. Sometimes, you might want the output to be left-justified. To left-justify the output, you use the manipulator `left`.

The syntax to set the manipulator `left` is:

```
ostreamVar << left;
```

where `ostreamVar` is an output stream variable. For example, the following statement sets the output to be left-justified on the standard output device:

```
cout << left;
```

You can disable the manipulator `left` by using the stream function `unsetf`. The syntax to disable the manipulator `left` is:

```
ostreamVar.unsetf(ios::left);
```

where `ostreamVar` is an output stream variable. Disabling the manipulator `left` returns the output to the settings of the default output format. For example, the following statement disables the manipulator `left` on the standard output device:

```
cout.unsetf(ios::left);
```

The syntax to set the manipulator `right` is:

```
ostreamVar << right;
```

where `ostreamVar` is an output stream variable. For example, the following statement sets the output to be right-justified on the standard output device:

```
cout << right;
```

NOTE

On some compilers, the statements `cin >> left;` and `cin >> right;` might not work. In this case, you can use `cin.setf(ios::left);` in place of `cin >> left;` and `cin.setf(ios::right);` in place of `cin >> right;`.

The program in Example 3-14 illustrates the effect of the manipulators `left` and `right`.

EXAMPLE 3-14

//Example: left justification

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int x = 15;                                //Line 1
    int y = 7634;                              //Line 2

    cout << left;                                //Line 3

    cout << "12345678901234567890" << endl;      //Line 4
    cout << setw(5) << x << setw(7) << y
        << setw(8) << "Warm" << endl;          //Line 5

    cout << setfill('*');                        //Line 6

    cout << setw(5) << x << setw(7) << y
        << setw(8) << "Warm" << endl;          //Line 7

    cout << setw(5) << x << setw(7) << setfill('#')
        << y << setw(8) << "Warm" << endl;      //Line 8

    cout << setw(5) << setfill('@') << x
        << setw(7) << setfill('#') << y
```

```

        << setw(8) << setfill('^') << "Warm"
        << endl;                                     //Line 9

    cout << right;                                     //Line 10
    cout << setfill(' ');                             //Line 11

    cout << setw(5) << x << setw(7) << y
        << setw(8) << "Warm" << endl;               //Line 12

    return 0;
}

```

Sample Run:

```

12345678901234567890
15   7634   Warm
15***7634***Warm***
15***7634###Warm###
15@@@7634###Warm^^^^
    15   7634   Warm

```

The output of this program is the same as the output of Example 3-11. The only difference here is that for the statements in Lines 4 through 9, the output is left-justified. You are encouraged to do a walk-through of this program.

NOTE

This chapter discusses several stream functions and stream manipulators. To use stream functions such as `get`, `ignore`, `fill`, and `clear` in a program, the program must include the header file `iostream`.

There are two types of manipulators: those with parameters and those without parameters. Manipulators with parameters are called **parameterized stream manipulators**. For example, manipulators such as `setprecision`, `setw`, and `setfill` are parameterized. On the other hand, manipulators such as `endl`, `fixed`, `scientific`, `showpoint`, and `left` do not have parameters.

To use a parameterized stream manipulator in a program, you must include the header file `iomanip`. Manipulators without parameters are part of the `iostream` header file and, therefore, do not require inclusion of the header file `iomanip`.

Input/Output and the `string` Type

You can use an input stream variable, such as `cin`, and the extraction operator `>>` to read a string into a variable of the data type `string`. For example, if the input is the string `"Shelly"`, the following code stores this input into the `string` variable `name`:

```

string name;    //variable declaration
cin >> name;    //input statement

```

Recall that the extraction operator skips any leading whitespace characters and that reading stops at a whitespace character. As a consequence, you cannot use the extraction operator to read strings that contain blanks. For example, suppose that the variable `name` is defined as noted above. If the input is:

Alice Wonderland

then after the statement:

```
cin >> name;
```

executes, the value of the variable `name` is "Alice".

To read a string containing blanks, you can use the function `getline`.

The syntax to use the function `getline` is:

```
getline(istreamVar, strVar);
```

where `istreamVar` is an input stream variable and `strVar` is a `string` variable. The reading is delimited by the newline character `'\n'`.

The function `getline` reads until it reaches the end of the current line. The newline character is also read but not stored in the `string` variable.

Consider the following statement:

```
string myString;
```

If the input is 29 characters:

```
bbbbHello there. How are you?
```

where `b` represents a blank, after the statement:

```
getline(cin, myString);
```

the value of `myString` is:

```
myString = "    Hello there. How are you?"
```

All 29 characters, including the first four blanks, are stored into `myString`.

Similarly, you can use an output stream variable, such as `cout`, and the insertion operator `<<` to output the contents of a variable of the data type `string`.

Debugging: Understanding Logic Errors and Debugging with `cout` Statements

In the debugging section of Chapter 2, we illustrated how to understand and correct syntax errors. As we have seen, syntax errors are reported by the compiler, and the compiler not only reports syntax errors, but also gives some explanation about the errors. On the other hand, logic errors are typically not caught by the compiler except for the trivial ones such as using a variable without properly initializing it. In this section, we illustrate how to spot and

correct logic errors using `cout` statements. Suppose that we want to write a program that takes as input the temperature in Fahrenheit and outputs the equivalent temperature in Celsius. The formula to convert the temperature is: $Celsius = 5 / 9 * (Fahrenheit - 32)$. So consider the following program:

```
#include <iostream>                                //Line 1

using namespace std;                                //Line 2

int main()                                          //Line 3
{                                                  //Line 4
    int fahrenheit;                                //Line 5
    int celsius;                                   //Line 6

    cout << "Enter temperature in Fahrenheit: "; //Line 7
    cin >> fahrenheit;                             //Line 8
    cout << endl;                                   //Line 9

    celsius = 5 / 9 * (fahrenheit - 32);           //Line 10

    cout << fahrenheit << " degree F = "           //Line 11
         << celsius << " degree C. " << endl;
    return 0;                                       //Line 12
}                                                  //Line 13
```

Sample Run 1: In this sample run, the user input is shaded.

Enter temperature in Fahrenheit: 32

32 degree F = 0 degree C.

Sample Run 2: In this sample run, the user input is shaded.

Enter temperature in Fahrenheit: 110

110 degree F = 0 degree C.

The result shown in the first calculation looks correct. However, the result in the second calculation is clearly not correct even though the same formula is used, because 110 degree F = 43 degree C. It means the value of `celsius` calculated in Line 10 is incorrect. Now, the value of `celsius` is given by the expression $5 / 9 * (fahrenheit - 32)$. So we should look at this expression closely. To see the effect of this expression, we can separately print the values of the two expression $5 / 9$ and $fahrenheit - 32$. This can be accomplished by temporarily inserting an output statement as shown in the following program:

```
#include <iostream>                                //Line 1

using namespace std;                                //Line 2

int main()                                          //Line 3
{                                                  //Line 4
    int fahrenheit;                                //Line 5
    int celsius;                                   //Line 6
```



```

cout << "Enter temperature in Fahrenheit: "; //Line 7
cin >> fahrenheit;                          //Line 8
cout << endl;                              //Line 9

cout << "5 / 9 = " << 5 / 9
    << "; fahrenheit - 32 = "
    << fahrenheit - 32 << endl;             //Line 9a

celsius = 5 / 9 * (fahrenheit - 32);        //Line 10

cout << fahrenheit << " degree F = "
    << celsius << " degree C. " << endl;    //Line 11

return 0;                                  //Line 12
}                                           //Line 13

```

Sample Run: In this sample run, the user input is shaded.

Enter temperature in Fahrenheit: **110**

```

5 / 9 = 0; fahrenheit - 32 = 78
110 degree F = 0 degree C.

```

Let us look at the sample run. We see that the value of $5 / 9 = 0$ and the value of $\text{fahrenheit} - 32 = 78$. Because `fahrenheit` = 110, the value of the expression $\text{fahrenheit} - 32$ is correct. Now let us look at the expression $5 / 9$. The value of this expression is 0. Because both of the operands, 5 and 9, of the operator `/` are integers, using integer division, the value of the expression is 0. That is, the value of the expression $5 / 9 = 0$ is also calculated correctly. So by the precedence of the operators, the value of the expression $5 / 9 * (\text{fahrenheit} - 32)$ will always be 0 regardless of the value of `fahrenheit`. So the problem is in the integer division. We can replace the expression $5 / 9$ with $5.0 / 9$. In this case, the value of the expression $5.0 / 9 * (\text{fahrenheit} - 32)$ will be a decimal number. Because `fahrenheit` and `celsius` are `int` variables, we can use the cast operators to convert this value to an integer, that is, we use the following expression:

```
celsius = static_cast<int>(5.0 / 9 * (fahrenheit - 32) + 0.5);
```

(Note that in the preceding expression, we added 0.5 to round the number to the nearest integer.)

The revised program is:

```

#include <iostream>                          //Line 1

using namespace std;                        //Line 2

int main()                                 //Line 3
{                                           //Line 4
    int fahrenheit;                         //Line 5
    int celsius;                           //Line 6

```

```

    cout << "Enter temperature in Fahrenheit: ";           //Line 7
    cin >> fahrenheit;                                     //Line 8
    cout << endl;                                           //Line 9

    celsius = static_cast<int>
               (5.0 / 9 * (fahrenheit - 32) + 0.5);         //Line 10

    cout << fahrenheit << " degree F = "
         << celsius << " degree C. " << endl;             //Line 11

    return 0;                                              //Line 12
}                                                         //Line 13

```

Sample Run: In this sample run, the user input is shaded.

Enter temperature in Fahrenheit: 110

110 degree F = 43 degree C.

As we can see, using temporary `cout` statements, we were able to find the problem. After correcting the problem, the temporary `cout` statements are removed.

The temperature conversion program contained logic errors, not syntax errors. Using `cout` statements to print the values of expressions and/or variables to see the results of a calculation is an effective way to find and correct logic errors.

File Input/Output

The previous sections discussed in some detail how to get input from the keyboard (standard input device) and send output to the screen (standard output device). However, getting input from the keyboard and sending output to the screen have several limitations. Inputting data in a program from the keyboard is comfortable as long as the amount of input is very small. Sending output to the screen works well if the amount of data is small (no larger than the size of the screen) and you do not want to distribute the output in a printed format to others.

If the amount of input data is large, however, it is inefficient to type it at the keyboard each time you run a program. In addition to the inconvenience of typing large amounts of data, typing can generate errors, and unintentional typos cause erroneous results. You must have some way to get data into the program from other sources. By using alternative sources of data, you can prepare the data before running a program, and the program can access the data each time it runs.

Suppose you want to present the output of a program in a meeting. Distributing printed copies of the program output is a better approach than showing the output on a screen. For example, you might give a printed report to each member of a committee before an important meeting. Furthermore, output must sometimes be saved so that the output produced by one program can be used as an input to other programs.

This section discusses how to obtain data from other input devices, such as a disk (that is, secondary storage), and how to save the output to a disk. C++ allows a program to get

data directly from and save output directly to secondary storage. A program can use the file I/O and read data from or write data to a file. Formally, a file is defined as follows:

File: An area in secondary storage used to hold information.

The standard I/O header file, `iostream`, contains data types and variables that are used only for input from the standard input device and output to the standard output device. In addition, C++ provides a header file called `fstream`, which is used for file I/O. Among other things, the `fstream` header file contains the definitions of two data types: `ifstream`, which means input file stream and is similar to `istream`, and `ofstream`, which means output file stream and is similar to `ostream`.

The variables `cin` and `cout` are already defined and associated with the standard input/output devices. In addition, `>>`, `get`, `ignore`, `putback`, `peek`, and so on can be used with `cin`, whereas `<<`, `setfill`, and so on can be used with `cout`. These same operators and functions are also available for file I/O, but the header file `fstream` does not declare variables to use them. You must declare variables called **file stream variables**, which include `ifstream` variables for input and `ofstream` variables for output. You then use these variables together with `>>`, `<<`, or other functions for I/O. Remember that C++ does not automatically initialize user-defined variables. Once you declare the `fstream` variables, you must associate these file variables with the input/output sources.

File I/O is a five-step process:

1. Include the header file `fstream` in the program.
2. Declare file stream variables.
3. Associate the file stream variables with the input/output sources.
4. Use the file stream variables with `>>`, `<<`, or other input/output functions.
5. Close the files.

We will now describe these five steps in detail. A skeleton program then shows how the steps might appear in a program.

Step 1 requires that the header file `fstream` be included in the program. The following statement accomplishes this task:

```
#include <fstream>
```

Step 2 requires you to declare file stream variables. Consider the following statements:

```
ifstream inData;
ofstream outData;
```

The first statement declares `inData` to be an input file stream variable. The second statement declares `outData` to be an output file stream variable.

Step 3 requires you to associate file stream variables with the input/output sources. This step is called **opening the files**. The stream member function `open` is used to open files. The syntax for opening a file is:

```
fileStreamVariable.open(sourceName);
```

Here, `fileStreamVariable` is a file stream variable, and `sourceName` is the name of the input/output file.

Suppose you include the declaration from Step 2 in a program. Further suppose that the input data is stored in a file called `prog.dat`. The following statements associate `inData` with `prog.dat` and `outData` with `prog.out`. That is, the file `prog.dat` is opened for inputting data, and the file `prog.out` is opened for outputting data.

```
inData.open("prog.dat"); //open the input file; Line 1
outData.open("prog.out"); //open the output file; Line 2
```

NOTE

IDEs such as Visual Studio .Net manage programs in the form of projects. That is, first you create a project, and then you add source files to the project. The statement in Line 1 assumes that the file `prog.dat` is in the same directory (subdirectory) as your project. However, if this is in a different directory (subdirectory), then you must specify the path where the file is located, along with the name of the file. For example, suppose that the file `prog.dat` is on a flash memory in drive H. Then the statement in Line 1 should be modified as follows:

```
inData.open("h:\\prog.dat");
```

Note that there are two `\` after `h:`. Recall from Chapter 2 that in C++, `\` is the escape character. Therefore, to produce a `\` within a string, you need `\\`. (To be absolutely sure about specifying the source where the input file is stored, such as the drive `h:\\`, check your system's documentation.)

Similar conventions for the statement in Line 2.

NOTE

Suppose that a program reads data from a file. Because different computers have drives labeled differently, for simplicity, throughout the book, we assume that the file containing the data and the program reading data from the file are in the same directory (subdirectory).

NOTE

We typically use `.dat`, `.out`, or `.txt` as an extension for the input and output files and use Notepad, Wordpad, or TextPad to create and open these files. You can also use your IDE's editor, if any, to create `.txt` (text) files. (To be absolutely sure about it, check you IDE's documentation.)

Step 4 typically works as follows. You use the file stream variables with `>>`, `<<`, or other input/output functions. The syntax for using `>>` or `<<` with file stream variables is exactly the same as the syntax for using `cin` and `cout`. Instead of using `cin` and `cout`, however, you use the file stream variable names that were declared. For example, the statement:

```
inData >> payRate;
```

reads the data from the file `prog.dat` and stores it in the variable `payRate`. The statement:

```
outData << "The paycheck is: $" << pay << endl;
```

stores the output—**The paycheck is: \$565.78**—in the file `prog.out`. This statement assumes that the pay was calculated as `565.78`.

Once the I/O is complete, Step 5 requires closing the files. Closing a file means that the file stream variables are disassociated from the storage area and are freed. Once these variables are freed, they can be reused for other file I/O. Moreover, closing an output file ensures that the entire output is sent to the file; that is, the buffer is emptied. You close files by using the stream function `close`. For example, assuming the program includes the declarations listed in Steps 2 and 3, the statements for closing the files are:

```
inData.close();
outData.close();
```

NOTE

On some systems, it is not necessary to close the files. When the program terminates, the files are closed automatically. Nevertheless, it is a good practice to close the files yourself. Also, if you want to use the same file stream variable to open another file, you must close the first file opened with that file stream variable.

In skeleton form, a program that uses file I/O usually takes the following form:

```
#include <fstream>

//Add additional header files you use

using namespace std;

int main()
{
    //Declare file stream variables such as the following
    ifstream inData;
    ofstream outData;
    .
    .
    .

    //Open the files
    inData.open("prog.dat"); //open the input file
    outData.open("prog.out"); //open the output file

    //Code for data manipulation

    //Close files
    inData.close();
    outData.close();

    return 0;
}
```

Recall that Step 3 requires the file to be opened for file I/O. Opening a file associates a file stream variable declared in the program with a physical file at the source, such as a disk. In the case of an input file, the file must exist before the **open** statement executes. If the file does not exist, the **open** statement fails and the input stream enters the fail state. An output file does not have to exist before it is opened; if the output file does not exist, the computer prepares an empty file for output. If the designated output file already exists, by default, the old contents are erased when the file is opened.

NOTE

To add the output at the end of an existing file, you can use the option `ios::app` as follows. Suppose that `outData` is declared as before and you want to add the output at the end of the existing file, say, `firstProg.out`. The statement to open this file is:

```
outData.open("firstProg.out", ios::app);
```

If the file `firstProg.out` does not exist, then the system creates an empty file.

NOTE

Appendix E discusses binary and random access files.

PROGRAMMING EXAMPLE:

Movie Tickets Sale and Donation to Charity

A movie in a local theater is in great demand. To help a local charity, the theater owner has decided to donate to the charity a portion of the gross amount generated from the movie. This example designs and implements a program that prompts the user to input the movie name, adult ticket price, child ticket price, number of adult tickets sold, number of child tickets sold, and percentage of the gross amount to be donated to the charity. The output of the program is as follows.

```
-_*-*_*-_*-*_**_-*_--*_---*_--*_-----*_-*_*-*_*-_*-*_*-*_*-*_*-_*
Movie Name: ..... Journey to Mars
Number of Tickets Sold: .....      2650
Gross Amount: ..... $ 9150.00
Percentage of Gross Amount Donated:    10.00%
Amount Donated: ..... $   915.00
Net Sale: ..... $ 8235.00
```

Note that the strings, such as **"Movie Name:"** , in the first column are left-justified, the numbers in the right column are right-justified, and the decimal numbers are output with two decimal places.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Input The input to the program consists of the movie name, adult ticket price, child ticket price, number of adult tickets sold, number of child tickets sold, and percentage of the gross amount to be donated to the charity.

Output The output is as shown above.

To calculate the amount donated to the local charity and the net sale, you first need to determine the gross amount. To calculate the gross amount, you multiply the number of adult tickets sold by the price of an adult ticket, multiply the number of child tickets sold by the price of a child ticket, and then add these two numbers. That is:

```
grossAmount = adultTicketPrice * noOfAdultTicketsSold
              + childTicketPrice * noOfChildTicketsSold;
```

Next, you determine the percentage of the amount donated to the charity and then calculate the net sale amount by subtracting the amount donated from the gross amount. The formulas to calculate the amount donated and the net sale amount are given below. This analysis leads to the following algorithm:

1. Get the movie name.
2. Get the price of an adult ticket.
3. Get the price of a child ticket.
4. Get the number of adult tickets sold.
5. Get the number of child tickets sold.
6. Get the percentage of the gross amount donated to the charity.
7. Calculate the gross amount using the following formula:

```
grossAmount = adultTicketPrice * noOfAdultTicketsSold
              + childTicketPrice * noOfChildTicketsSold;
```
8. Calculate the amount donated to the charity using the following formula:

```
amountDonated = grossAmount * percentDonation / 100;
```
9. Calculate the net sale amount using the following formula:

```
netSaleAmount = grossAmount - amountDonated;
```

Variables From the preceding discussion, it follows that you need variables to store the movie name, adult ticket price, child ticket price, number of adult tickets sold, number of child tickets sold, percentage of the gross amount donated to the charity, gross amount, amount donated, and net sale amount. Therefore, the following variables are needed:

```
string movieName;
double adultTicketPrice;
double childTicketPrice;
int noOfAdultTicketsSold;
int noOfChildTicketsSold;
```


1. Declare the variables.
2. Set the output of the floating-point numbers to two decimal places in a fixed decimal format with a decimal point and trailing zeros. Include the header file `iomanip`.
3. Prompt the user to enter a movie name.
4. Input (read) the movie name. Because the name of a movie might contain more than one word (and, therefore, might contain blanks), the program uses the function `getline` to input the movie name.
5. Prompt the user to enter the price of an adult ticket.
6. Input (read) the price of an adult ticket.
7. Prompt the user to enter the price of a child ticket.
8. Input (read) the price of a child ticket.
9. Prompt the user to enter the number of adult tickets sold.
10. Input (read) the number of adult tickets sold.
11. Prompt the user to enter the number of child tickets sold.
12. Input (read) the number of child tickets sold.
13. Prompt the user to enter the percentage of the gross amount donated.
14. Input (read) the percentage of the gross amount donated.
15. Calculate the gross amount.
16. Calculate the amount donated.
17. Calculate the net sale amount.
18. Output the results.

COMPLETE PROGRAM LISTING

```
//*****
// Author: D.S. Malik
//
// Program: Movie Tickets Sale
// This program determines the money to be donated to a
// charity. It prompts the user to input the movie name, adult
// ticket price, child ticket price, number of adult tickets
// sold, number of child tickets sold, and percentage of the
// gross amount to be donated to the charity.
//*****

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;
```

```

int main()
{
    //Step 1
    string movieName;
    double adultTicketPrice;
    double childTicketPrice;
    int noOfAdultTicketsSold;
    int noOfChildTicketsSold;
    double percentDonation;
    double grossAmount;
    double amountDonated;
    double netSaleAmount;

    cout << fixed << showpoint << setprecision(2); //Step 2

    cout << "Enter the movie name: "; //Step 3
    getline(cin, movieName); //Step 4
    cout << endl;

    cout << "Enter the price of an adult ticket: "; //Step 5
    cin >> adultTicketPrice; //Step 6
    cout << endl;

    cout << "Enter the price of a child ticket: "; //Step 7
    cin >> childTicketPrice; //Step 8
    cout << endl;
    cout << "Enter the number of adult tickets "
        << "sold: "; //Step 9
    cin >> noOfAdultTicketsSold; //Step 10
    cout << endl;

    cout << "Enter the number of child tickets "
        << "sold: "; //Step 11
    cin >> noOfChildTicketsSold; //Step 12
    cout << endl;

    cout << "Enter the percentage of donation: "; //Step 13
    cin >> percentDonation; //Step 14
    cout << endl << endl;

    //Step 15
    grossAmount = adultTicketPrice * noOfAdultTicketsSold +
        childTicketPrice * noOfChildTicketsSold;

    //Step 16
    amountDonated = grossAmount * percentDonation / 100;

    netSaleAmount = grossAmount - amountDonated; //Step 17

    //Step 18: Output results
    cout << "_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*"
        << "_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*" << endl;
}

```

```

cout << setfill('.') << left << setw(35) << "Movie Name: "
    << right << " " << movieName << endl;
cout << left << setw(35) << "Number of Tickets Sold: "
    << setfill(' ') << right << setw(10)
    << noOfAdultTicketsSold + noOfChildTicketsSold
    << endl;
cout << setfill('.') << left << setw(35)
    << "Gross Amount: "
    << setfill(' ') << right << " $"
    << setw(8) << grossAmount << endl;
cout << setfill('.') << left << setw(35)
    << "Percentage of Gross Amount Donated: "
    << setfill(' ') << right
    << setw(9) << percentDonation << '%' << endl;
cout << setfill('.') << left << setw(35)
    << "Amount Donated: "
    << setfill(' ') << right << " $"
    << setw(8) << amountDonated << endl;
cout << setfill('.') << left << setw(35) << "Net Sale: "
    << setfill(' ') << right << " $"
    << setw(8) << netSaleAmount << endl;

return 0;
}

```

Sample Run: In this sample run, the user input is shaded.

Enter movie name: **Journey to Mars**

Enter the price of an adult ticket: **4.50**

Enter the price of a child ticket: **3.00**

Enter number of adult tickets sold: **800**

Enter number of child tickets sold: **1850**

Enter the percentage of donation: **10**

```

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Movie Name: ..... Journey to Mars
Number of Tickets Sold: ..... 2650
Gross Amount: ..... $ 9150.00
Percentage of Gross Amount Donated: 10.00%
Amount Donated: ..... $ 915.00
Net Sale: ..... $ 8235.00

```

Note that the first six lines of output get the necessary data to generate the last six lines of the output as required.

PROGRAMMING EXAMPLE: Student Grade

Write a program that reads a student name followed by five test scores. The program should output the student name, the five test scores, and the average test score. Output the average test score with two decimal places.

The data to be read is stored in a file called `test.txt`. The output should be stored in a file called `testavg.out`.

Input A file containing the student name and the five test scores. A sample input is:

 Andrew Miller 87.50 89 65.75 37 98.50

Output The student name, the five test scores, and the average of the five test scores, saved to a file.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

To find the average of the five test scores, you add the five test scores and divide the sum by 5. The input data is in the following form: the student name followed by the five test scores. Therefore, you must read the student name first and then read the five test scores. This problem analysis translates into the following algorithm:

1. Read the student name and the five test scores.
2. Output the student name and the five test scores.
3. Calculate the average.
4. Output the average.

You output the average test score in the fixed decimal format with two decimal places.

Variables The program needs to read a student's first and last name and five test scores. Therefore, you need two variables to store the student name and five variables to store the five test scores.

To find the average, you must add the five test scores and then divide the sum by 5. Thus, you need a variable to store the average test score. Furthermore, because the input data is in a file, you need an `ifstream` variable to open the input file. Because the program output will be stored in a file, you need an `ofstream` variable to open the output file. The program, therefore, needs at least the following variables:

```
ifstream inFile;    //input file stream variable
ofstream outFile;   //output file stream variable

double test1, test2, test3, test4, test5; //variables to
                                           //read the five test scores
double average;     //variable to store the average test score
string firstName;    //variable to store the first name
string lastName;     //variable to store the last name
```

MAIN ALGORITHM

In the preceding sections, we analyzed the problem and determined the formulas to perform the calculations. We also determined the necessary variables and named

constants. We can now expand the previous algorithm to solve the problem given at the beginning of this programming example:

1. Declare the variables.
2. Open the input file.
3. Open the output file.
4. To output the floating-point numbers in a fixed decimal format with a decimal point and trailing zeros, set the manipulators **fixed** and **showpoint**. Also, to output the floating-point numbers with two decimal places, set the precision to two decimal places.
5. Read the student name.
6. Output the student name.
7. Read the five test scores.
8. Output the five test scores.
9. Find the average test score.
10. Output the average test score.
11. Close the input and output files.

Because this program reads data from a file and outputs data to a file, it must include the header file **fstream**. Because the program outputs the average test score to two decimal places, you need to set the precision to two decimal places. Therefore, the program uses the manipulator **setprecision**, which requires you to include the header file **iomanip**. Because **firstName** and **lastName** are **string** variables, we must include the header file **string**. The program also includes the header file **iostream** to print a message on the screen so that you will not stare at a blank screen while the program executes.

COMPLETE PROGRAM LISTING

```
//*****
// Author: D.S. Malik
//
// Program to calculate the average test score.
// Given a student's name and five test scores, this program
// calculates the average test score. The student's name, the
// five test scores, and the average test score are stored in
// the file testavg.out. The data is input from the file
// test.txt.
//*****

#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>

using namespace std;
```

```

int main()
{
    //Declare variables; Step 1
    ifstream inFile;
    ofstream outFile;

    double test1, test2, test3, test4, test5;
    double average;

    string firstName;
    string lastName;

    inFile.open("test.txt"); //Step 2
    outFile.open("testavg.out"); //Step 3

    outFile << fixed << showpoint; //Step 4
    outFile << setprecision(2); //Step 4

    cout << "Processing data" << endl;

    inFile >> firstName >> lastName; //Step 5
    outFile << "Student name: " << firstName
        << " " << lastName << endl; //Step 6

    inFile >> test1 >> test2 >> test3
        >> test4 >> test5; //Step 7
    outFile << "Test scores: " << setw(6) << test1
        << setw(6) << test2 << setw(6) << test3
        << setw(6) << test4 << setw(6) << test5
        << endl; //Step 8

    average = (test1 + test2 + test3 + test4
        + test5) / 5.0; //Step 9

    outFile << "Average test score: " << setw(6)
        << average << endl; //Step 10

    inFile.close(); //Step 11
    outFile.close(); //Step 11

    return 0;
}

```

Sample Run:**Input File** (contents of the file test.txt):

Andrew Miller 87.50 89 65.75 37 98.50

Output File (contents of the file testavg.out):

Student name: Andrew Miller
 Test scores: 87.50 89.00 65.75 37.00 98.50
 Average test score: 75.55

NOTE

The preceding program uses five variables—`test1`, `test2`, `test3`, `test4`, and `test5`—to read the five test scores and then find the average test score. The Web site accompanying this book contains a modified version of this program that uses only one variable, `testScore`, to read the test scores and another variable, `sum`, to find the sum of the test scores. The program is named `Ch3_AverageTestScoreVersion2.cpp`.

QUICK REVIEW

1. A stream in C++ is an infinite sequence of characters from a source to a destination.
2. An input stream is a stream from a source to a computer.
3. An output stream is a stream from a computer to a destination.
4. `cin`, which stands for common input, is an input stream object, typically initialized to the standard input device, which is the keyboard.
5. `cout`, which stands for common output, is an output stream object, typically initialized to the standard output device, which is the screen.
6. When the binary operator `>>` is used with an input stream object, such as `cin`, it is called the stream extraction operator. The left-side operand of `>>` must be an input stream variable, such as `cin`; the right-side operand must be a variable.
7. When the binary operator `<<` is used with an output stream object, such as `cout`, it is called the stream insertion operator. The left-side operand of `<<` must be an output stream variable, such as `cout`; the right-side operand of `<<` must be an expression or a manipulator.
8. When inputting data into a variable, the operator `>>` skips all leading whitespace characters.
9. To use `cin` and `cout`, the program must include the header file `iostream`.
10. The function `get` is used to read data on a character-by-character basis and does not skip any whitespace characters.
11. The function `ignore` is used to skip data in a line.
12. The function `putback` puts the last character retrieved by the function `get` back into the input stream.
13. The function `peek` returns the next character from the input stream but does not remove the character from the input stream.
14. Attempting to read invalid data into a variable causes the input stream to enter the fail state.
15. Once an input failure has occurred, you use the function `clear` to restore the input stream to a working state.

16. The manipulator `setprecision` formats the output of floating-point numbers to a specified number of decimal places.
17. The manipulator `fixed` outputs floating-point numbers in the fixed decimal format.
18. The manipulator `showpoint` outputs floating-point numbers with a decimal point and trailing zeros.
19. The manipulator `setw` formats the output of an expression in a specific number of columns; the default output is right-justified.
20. If the number of columns specified in the argument of `setw` is less than the number of columns needed to print the value of the expression, the output is not truncated and the output of the expression expands to the required number of columns.
21. The manipulator `setfill` is used to fill the unused columns on an output device with a character other than a space.
22. If the number of columns specified in the `setw` manipulator exceeds the number of columns required by the next expression, the output is right-justified. To left-justify the output, you use the manipulator `left`.
23. To use the stream functions `get`, `ignore`, `putback`, `peek`, `clear`, and `unsetf` for standard I/O, the program must include the header file `iostream`.
24. To use the manipulators `setprecision`, `setw`, and `setfill`, the program must include the header file `iomanip`.
25. The header file `fstream` contains the definitions of `ifstream` and `ofstream`.
26. For file I/O, you must use the statement `#include <fstream>` to include the header file `fstream` in the program. You must also do the following: declare variables of type `ifstream` for file input and of type `ofstream` for file output and use open statements to open input and output files. You can use `<<`, `>>`, `get`, `ignore`, `peek`, `putback`, or `clear` with file stream variables.
27. To close a file as indicated by the `ifstream` variable `inFile`, you use the statement `inFile.close();`. To close a file as indicated by the `ofstream` variable `outFile`, you use the statement `outFile.close();`.

EXERCISES

1. Mark the following statements as true or false.
 - a. The extraction operator `>>` skips all leading whitespace characters when searching for the next data in the input stream.
 - b. In the statement `cin >> x;`, `x` must be a variable.
 - c. The statement `cin >> x >> y;` requires the input values for `x` and `y` to appear on the same line.
 - d. The statement `cin >> num;` is equivalent to the statement `num >> cin;`.

- e. You generate the newline character by pressing the Enter (return) key on the keyboard.
 - f. The function `ignore` is used to skip certain input in a line.
2. Suppose `x` and `y` are `int` variables and `ch` is a `char` variable. Consider the following input:

5 28 36

What value (if any) is assigned to `x`, `y`, and `ch` after each of the following statements executes? (Use the same input for each statement.)

- a. `cin >> x >> y >> ch;`
 - b. `cin >> ch >> x >> y;`
 - c. `cin >> x >> ch >> y;`
 - d. `cin >> x >> y;`
`cin.get(ch);`
3. Suppose `x` and `y` are `int` variables and `z` is a `double` variable. Assume the following input data:

37 86.56 32

What value (if any) is assigned to `x`, `y`, and `z` after each of the following statements executes? (Use the same input for each statement.)

- a. `cin >> x >> y >> z;`
 - b. `cin >> x >> z >> y;`
 - c. `cin >> z >> x >> y;`
4. Suppose `x` and `y` are `int` variables and `ch` is a `char` variable. Assume the following input data:

13 28 D
14 E 98
A B 56

What value (if any) is assigned to `x`, `y`, and `ch` after each of the following statements executes? (Use the same input for each statement.)

- a. `cin >> x >> y;`
`cin.ignore(50, '\n');`
`cin >> ch;`
- b. `cin >> x;`
`cin.ignore(50, '\n');`
`cin >> y;`
`cin.ignore(50, '\n');`
`cin.get(ch);`
- c. `cin >> y;`
`cin.ignore(50, '\n');`
`cin >> x >> ch;`

```
d.  cin.get(ch);
    cin.ignore(50, '\n');
    cin >> x;
    cin.ignore(50, 'E');
    cin >> y;
```

5. Given the input:

```
46 A 49
```

and the C++ code:

```
int x = 10, y = 18;
char z = '*';
cin >> x >> y >> z;
cout << x << " " << y << " " << z << endl;
```

What is the output?

6. Suppose that **x** and **y** are **int** variables, **z** is a **double** variable, and **ch** is a **char** variable. Suppose the input statement is:

```
cin >> x >> y >> ch >> z;
```

What values, if any, are stored in **x**, **y**, **z**, and **ch** if the input is:

- a. 35 62.78
 - b. 86 32A 92.6
 - c. 12 .45A 32
7. Which header file must be included to use the function **steprecision**?
8. Which header file must be included to use the function **pow**?
9. Suppose that **name** is a variable of type **string**. Write the input statement to read and store the input **Brenda Clinton** in **name**. (Assume that the input is from the standard input device.)
10. Write a C++ statement that uses the manipulator **setfill** to output a line containing 35 stars, as in the following line:
- ```

```
11. Suppose that **age** is an **int** variable and **name** is a **string** variable. What are the values of **age** and **name** after the following input statements execute:
- ```
cin >> age;
getline(cin, name);
```
- if the input is:
- a. 23 Lance Grant
 - b. 23
Lance Grant
12. Suppose that **age** is an **int** variable, **ch** is a **char** variable, and **name** is a **string** variable. What are the values of **age** and **name** after the following input statements execute:

```
cin >> age;
cin.get(ch);
getline(cin, name);
```

if the input is:

- a. 23 Lance Grant
- b. 23
Lance Grant

13. The following program is supposed to read two numbers from a file named `input.dat` and write the sum of the numbers to a file named `output.dat`. However, it fails to do so. Rewrite the program so that it accomplishes what it is intended to do. (Also, include statements to close the files.)

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    int num1, num2;
    ifstream infile;

    outfile.open("output.dat");
    infile >> num1 >> num2;
    outfile << "Sum = " << num1 + num2 << endl;
    return 0;
}
```

14. What may cause an input stream to enter the fail state? What happens when an input stream enters the fail state?
15. Which header file needs to be included in a program that uses the data types `ifstream` and `ofstream`?
16. Suppose that `infile` is an `ifstream` variable and `employee.dat` is a file that contains employees' information. Write the C++ statement that opens this file using the variable `infile`.
17. A program reads data from a file called `inputFile.dat` and, after doing some calculations, writes the results to a file called `outFile.dat`. Answer the following questions:
- a. After the program executes, what are the contents of the file `inputFile.dat`?
 - b. After the program executes, what are the contents of the file `outFile.dat` if this file was empty before the program executed?
 - c. After the program executes, what are the contents of the file `outFile.dat` if this file contained 100 numbers before the program executed?
 - d. What would happen if the file `outFile.dat` did not exist before the program executed?

18. Suppose that `infile` is an `ifstream` variable and it is associated with the file that contains the following data: 27306 savings 7503.35. Write the C++ statement(s) that reads and stores the first input in the `int` variable `acctNumber`, the second input in the `string` variable `accountType`, and the third input in the `double` variable `balance`.
19. Suppose that you have the following statements:
- ```
ofstream outfile;
double distance = 375;
double speed = 58;
double travelTime;
```

Write C++ statements to do the following:

- Open the file `travel.dat` using the variable `outfile`.
- Write the statement to format your output to two decimal places in fixed form.
- Write the values of the variables `day`, `distance`, and `speed` in the file `travel.dat`.
- Calculate and write the `travelTime` in the file `travel.dat`.
- Which header files are needed to process the information in (a) to (d)?

## PROGRAMMING EXERCISES

---

1. Consider the following incomplete C++ program:

```
#include <iostream>

int main()
{
 ...
}
```

- Write a statement that includes the header files `fstream`, `string`, and `iomanip` in this program.
- Write statements that declare `infile` to be an `ifstream` variable and `outfile` to be an `ofstream` variable.
- The program will read data from the file `inData.txt` and write output to the file `outData.txt`. Write statements to open both of these files, associate `infile` with `inData.txt`, and associate `outfile` with `outData.txt`.
- Suppose that the file `inData.txt` contains the following data:

```
10.20 5.35
15.6
Randy Gill 31
18500 3.5
A
```

The numbers in the first line represent the length and width, respectively, of a rectangle. The number in the second line represents the radius of a circle. The third line contains the first name, last name, and the age of a person. The first number in the fourth line is the savings account balance at the beginning of the month, and the second number is the interest rate per year. (Assume that  $\pi = 3.1416$ .) The fifth line contains an uppercase letter between **A** and **Y** (inclusive). Write statements so that after the program executes, the contents of the file `outData.txt` are as shown below. If necessary, declare additional variables. Your statements should be general enough so that if the content of the input file changes and the program is run again (without editing and recompiling), it outputs the appropriate results.

Rectangle:

Length = 10.20, width = 5.35, area = 54.57, parameter = 31.10

Circle:

Radius = 15.60, area = 764.54, circumference = 98.02

Name: Randy Gill, age: 31

Beginning balance = \$18500.00, interest rate = 3.50

Balance at the end of the month = \$18553.96

The character that comes after **A** in the ASCII set is **B**

- e. Write statements that close the input and output files.
  - f. Write a C++ program that tests the statements in parts a through e.
2. Consider the following program in which the statements are in the incorrect order. Rearrange the statements so that the program prompts the user to input the height and the radius of the base of a cylinder and outputs the volume and surface area of the cylinder. Format the output to two decimal places.

```
#include <iomanip>
#include <cmath>

int main()
{

 double height;

 cout << "Volume of the cylinder = "
 << PI * pow(radius, 2.0)* height << endl;

 cout << "Enter the height of the cylinder: ";
 cin >> radius;
 cout << endl;

 return 0;

 double radius;
```

```

 cout << "Surface area: "
 << 2 * radius * + 2 * PI * pow(radius, 2.0) << endl;
 cout << fixed << showpoint << setprecision(2);

 cout << "Enter the radius of the base of the cylinder: ";
 cin >> height;
 cout << endl;

```

```

#include <iostream>
const double PI = 3.14159;

```

```
using namespace std;
```

3. Write a program that prompts the user to enter the weight of a person in kilograms and outputs the equivalent weight in pounds. Output both the weights rounded to two decimal places. (Note that 1 kilogram = 2.2 pounds.) Format your output with two decimal places.
4. During each summer, John and Jessica grow vegetables in their back yard and buy seeds and fertilizer from a local nursery. The nursery carries different types of vegetable fertilizers in various bag sizes. When buying a particular fertilizer, they want to know the price of the fertilizer per pound and the cost of fertilizing per square foot. The following program prompts the user to enter the size of the fertilizer bag, in pounds, the cost of the bag, and the area, in square feet, that can be covered by the bag. The program should output the desired result. However, the program contains logic errors. Find and correct the logic errors so that the program works properly.

```
//Logic errors.
```

```

#include <iostream>
#include <iomanip>

```

```
using namespace std;
```

```

int main()
{

```

```

 double cost;
 double area;

```

```
 double bagSize;
```

```
 cout << fixed << showpoint << setprecision(2);
```

```

 cout << "Enter the amount of fertilizer, in pounds, "
 << "in one bag: ";
 cin >> bagSize;
 cout << endl;

```

```

 cout << "Enter the cost of the " << bagSize
 << " pound fertilizer bag: ";

```

```

 cin >> cost;
 cout << endl;

 cout << "Enter the area, in square feet, that can be "
 << "fertilized by one bag: ";
 cin >> area;
 cout << endl;

 cout << "The cost of the fertilizer per pound is: $"
 << bagSize / cost << endl;
 cout << "The cost of fertilizing per square foot is: $"
 << area / cost << endl;

 return 0;
}

```

5. The manager of a football stadium wants you to write a program that calculates the total ticket sales after each game. There are four types of tickets—box, sideline, premium, and general admission. After each game, data is stored in a file in the following form:

```

ticketPrice numberOfTicketsSold
...

```

Sample data are shown below:

```

250 5750
100 28000
 50 35750
 25 18750

```

The first line indicates that the ticket price is \$250 and that 5750 tickets were sold at that price. Output the number of tickets sold and the total sale amount. Format your output with two decimal places.

6. Write a program that calculates and prints the monthly paycheck for an employee. The net pay is calculated after taking the following deductions:

```

Federal Income Tax: 15%
State Tax: 3.5%
Social Security Tax: 5.75%
Medicare/Medicaid Tax: 2.75%
Pension Plan: 5%
Health Insurance: $75.00

```

Your program should prompt the user to input the gross amount and the employee name. The output will be stored in a file. Format your output to have two decimal places. A sample output follows:

```

Bill Robinson
Gross Amount: $3575.00
Federal Tax: $ 536.25
State Tax: $ 125.13

```

```

Social Security Tax: $ 205.56
Medicare/Medicaid Tax: ... $ 98.31
Pension Plan: $ 178.75
Health Insurance: $ 75.00
Net Pay: $2356.00

```

Note that the first column is left-justified, and the right column is right-justified.

7. Redo Programming Exercise 21, in Chapter 2, so that each string can store a line of text.
8. Three employees in a company are up for a special pay increase. You are given a file, say `Ch3_Ex8Data.txt`, with the following data:

```

Miller Andrew 65789.87 5
Green Sheila 75892.56 6
Sethi Amit 74900.50 6.1

```

Each input line consists of an employee's last name, first name, current salary, and percent pay increase. For example, in the first input line, the last name of the employee is **Miller**, the first name is **Andrew**, the current salary is **65789.87**, and pay increase is **5%**. Write a program that reads data from the specified file and stores the output in the file `Ch3_Ex8Output.dat`. For each employee, the data must be output in the following form: **firstName lastName updatedSalary**. Format the output of decimal numbers to two decimal places.

9. Write a program that accepts as input the mass, in grams, and density, in grams per cubic centimeters, and outputs the volume of the object using the formula:  $density = mass / volume$ . Format your output to two decimal places.
10. Interest on a credit card's unpaid balance is calculated using the average daily balance. Suppose that *netBalance* is the balance shown in the bill, *payment* is the payment made, *d1* is the number of days in the billing cycle, and *d2* is the number of days payment is made before billing cycle. Then, the average daily balance is:

$$averageDailyBalance = (netBalance * d1 - payment * d2) / d1$$

If the interest rate per month is, say, 0.0152, then the interest on the unpaid balance is:

$$interest = averageDailyBalance * 0.0152$$

Write a program that accepts as input *netBalance*, *payment*, *d1*, *d2*, and interest rate per month. The program outputs the interest. Format your output to two decimal places.





# CHAPTER 4

## CONTROL STRUCTURES I (SELECTION)

IN THIS CHAPTER, YOU WILL:

- Learn about control structures
- Examine relational and logical operators
- Explore how to form and evaluate logical (Boolean) expressions
- Discover how to use the selection control structures `if`, `if...else`, and `switch` in a program
- Learn how to avoid bugs by avoiding partially understood concepts
- Learn to use the `assert` function to terminate a program

Chapter 2 defined a program as a sequence of statements whose objective is to accomplish some task. The programs you have examined so far were simple and straightforward. To process a program, the computer begins at the first executable statement and executes the statements in order until it comes to the end. In this chapter and Chapter 5, you will learn how to tell a computer that it does not have to follow a simple sequential order of statements; it can also make decisions and repeat certain statements over and over until certain conditions are met.

## Control Structures

A computer can process a program in one of the following ways: in sequence; selectively, by making a choice, which is also called a branch; repetitively, by executing a statement over and over, using a structure called a loop; or by calling a function. Figure 4-1 illustrates the first three types of program flow. (In Chapter 7, we will show how function calls work.) The programming examples in Chapters 2 and 3 included simple sequential programs. With such a program, the computer starts at the beginning and follows the statements in order. No choices are made; there is no repetition. Control structures provide alternatives to sequential program execution and are used to alter the sequential flow of execution. The two most common control structures are selection and repetition. In *selection*, the program executes particular statements depending on some condition(s). In *repetition*, the program repeats particular statements a certain number of times based on some condition(s).

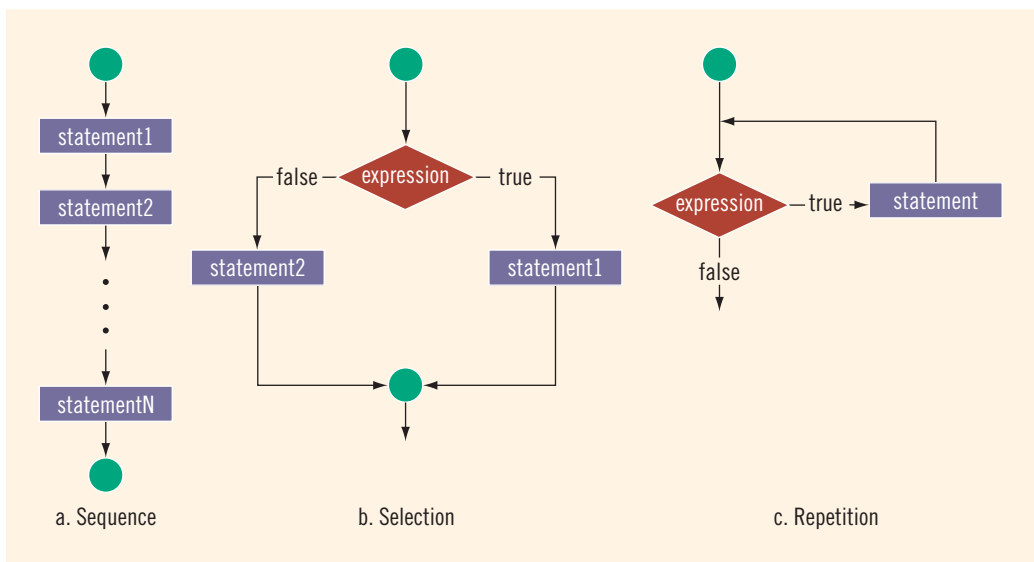


FIGURE 4-1 Flow of execution

Before you can learn about selection and repetition, you must understand the nature of conditional statements and how to use them. Consider the following three statements:

1. `if (score is greater than or equal to 90)`  
    `grade is A`
2. `if (hours worked are less than or equal to 40)`  
    `wages = rate * hours`  
    `otherwise`  
        `wages = (rate * 40) + 1.5 * (rate * (hours - 40))`
3. `if (temperature is greater than 70 degrees and it is not raining)`  
    `Go golfing!`

These statements are examples of conditional statements. You can see that certain statements are to be executed only if certain conditions are met. A condition is met if it evaluates to `true`. For example, in statement 1:

`score is greater than or equal to 90`

is `true` if the value of `score` is greater than or equal to 90; it is `false` otherwise. For example, if the value of `score` is 95, the statement evaluates to `true`. Similarly, if the value of `score` is 86, the statement evaluates to `false`.

It would be useful if the computer could recognize these types of statements to be true for appropriate values. Furthermore, in certain situations, the truth or falsity of a statement could depend on more than one condition. For example, in statement 3, both `temperature is greater than 70 degrees` and `it is not raining` must be true to recommend golfing.

As you can see, for the computer to make decisions and repeat statements, it must be able to react to conditions that exist when the program executes. The next few sections discuss how to represent and evaluate conditional statements in C++.

## Relational Operators

To make decisions, you must be able to express conditions and make comparisons. For example, the interest rate and service charges on a checking account might depend on the balance at the end of the month. If the balance is less than some minimum balance, not only is the interest rate lower, but there is also usually a service charge. Therefore, to determine the interest rate, you must be able to state the minimum balance and compare the account balance with the minimum balance (a condition). The premium on an insurance policy is also determined by stating conditions and making comparisons. For example, to determine an insurance premium, you must be able to check the smoking status of the policyholder. Nonsmokers (the condition) receive lower premiums than smokers. Both of these examples involve comparing items. Certain items are compared

for equality against a particular condition; others are compared for inequality (greater than or less than) against a particular condition.

In C++, a condition is represented by a logical (Boolean) expression. An expression that has a value of either `true` or `false` is called a **logical (Boolean) expression**. Moreover, `true` and `false` are **logical (Boolean) values**. Suppose `i` and `j` are integers. Consider the expression:

`i > j`

If this expression is a logical expression, it will have the value `true` if the value of `i` is greater than the value of `j`; otherwise, it will have the value `false`. The symbol `>` is called a relational operator. A **relational operator** allows you to make comparisons in a program.

C++ includes six relational operators that allow you to state conditions and make comparisons. Table 4-1 lists the relational operators.

**TABLE 4-1** Relational Operators in C++

| Operator           | Description              |
|--------------------|--------------------------|
| <code>==</code>    | equal to                 |
| <code>!=</code>    | not equal to             |
| <code>&lt;</code>  | less than                |
| <code>&lt;=</code> | less than or equal to    |
| <code>&gt;</code>  | greater than             |
| <code>&gt;=</code> | greater than or equal to |

**NOTE** In C++, the symbol `==`, which consists of two equal signs, is called the equality operator. Recall that the symbol `=` is called the assignment operator. Remember that the equality operator, `==`, determines whether two expressions are equal, whereas the assignment operator, `=`, assigns the value of an expression to a variable.

Each of the relational operators is a binary operator; that is, it requires two operands. Because the result of a comparison is `true` or `false`, expressions using these operators evaluate to `true` or `false`.

## Relational Operators and Simple Data Types

You can use the relational operators with all three simple data types. In the following example, the expressions use both integers and real numbers:

EXAMPLE 4-1

| Expression | Meaning                          | Value |
|------------|----------------------------------|-------|
| 8 < 15     | 8 is less than 15                | true  |
| 6 != 6     | 6 is not equal to 6              | false |
| 2.5 > 5.8  | 2.5 is greater than 5.8          | false |
| 5.9 <= 7.5 | 5.9 is less than or equal to 7.5 | true  |

Comparing Characters

For `char` values, whether an expression using relational operators evaluates to `true` or `false` depends on a machine's collating sequence. The collating sequence of some of the characters is:

| ASCII Value | Char | ASCII Value | Char | ASCII Value | Char | ASCII Value | Char |
|-------------|------|-------------|------|-------------|------|-------------|------|
| 32          | ' '  | 61          | =    | 81          | Q    | 105         | i    |
| 33          | !    | 62          | >    | 82          | R    | 106         | j    |
| 34          | "    | 65          | A    | 83          | S    | 107         | k    |
| 42          | *    | 66          | B    | 84          | T    | 108         | l    |
| 43          | +    | 67          | C    | 85          | U    | 109         | m    |
| 45          | -    | 68          | D    | 86          | V    | 110         | n    |
| 47          | /    | 69          | E    | 87          | W    | 111         | o    |
| 48          | 0    | 70          | F    | 88          | X    | 112         | p    |
| 49          | 1    | 71          | G    | 89          | Y    | 113         | q    |
| 50          | 2    | 72          | H    | 90          | Z    | 114         | r    |
| 51          | 3    | 73          | I    | 97          | a    | 115         | s    |
| 52          | 4    | 74          | J    | 98          | b    | 116         | t    |
| 53          | 5    | 75          | K    | 99          | c    | 117         | u    |
| 54          | 6    | 76          | L    | 100         | d    | 118         | v    |
| 55          | 7    | 77          | M    | 101         | e    | 119         | w    |
| 56          | 8    | 78          | N    | 102         | f    | 120         | x    |
| 57          | 9    | 79          | O    | 103         | g    | 121         | y    |
| 60          | <    | 80          | P    | 104         | h    | 122         | z    |

The ASCII character set is described in Appendix C.

Now, because 32 < 97, and the ASCII value of ' ' is 32 and the ASCII value of 'a' is 97, it follows that ' ' < 'a' is `true`. Similarly, using the previous ASCII values:

'R' > 'T' is `false`

'+' < '\*' is `false`

'A' <= 'a' is `true`

note that comparing values of different data types may produce unpredictable results. For example, the following expression compares an integer and a character:

```
8 < '5'
```

In this expression, on a particular machine, `8` would be compared with the collating sequence of `'5'`, which is 53. That is, `8` is compared with 53, which makes this particular expression evaluate to `true`.

Expressions such as `4 < 6` and `'R' > 'T'` are examples of **logical (Boolean) expressions**. When C++ evaluates a logical expression, it returns an integer value of 1 if the logical expression evaluates to `true`; it returns an integer value of 0 otherwise. In C++, any nonzero value is treated as `true`.

**NOTE** Chapter 2 introduced the data type `bool`. Recall that the data type `bool` has two values, `true` and `false`. In C++, `true` and `false` are reserved words. The identifier `true` is set to 1, and the identifier `false` is set to 0. For readability, whenever logical expressions are used, the identifiers `true` and `false` will be used here as the value of the logical expression.

## Relational Operators and the `string` Type

The relational operators can be applied to variables of type `string`. Variables of type `string` are compared character by character, starting with the first character and using the ASCII collating sequence. The character-by-character comparison continues until either a mismatch is found or the last characters have been compared and are equal. The following example shows how variables of type `string` are compared.

### EXAMPLE 4-2

Suppose that you have the statements:

```
string str1 = "Hello";
string str2 = "Hi";
string str3 = "Air";
string str4 = "Bill";
string str5 = "Big";
```

The following expressions show how string relational expressions evaluate.

| Expression                  | Value /Explanation                                                                                                                                                                                                                                                                                                                                                   |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str1 &lt; str2</code> | <code>true</code><br><code>str1 = "Hello"</code> and <code>str2 = "Hi"</code> . The first characters of <code>str1</code> and <code>str2</code> are the same, but the second character <code>'e'</code> of <code>str1</code> is less than the second character <code>'i'</code> of <code>str2</code> . Therefore, <code>str1 &lt; str2</code> is <code>true</code> . |

|                              |                                                                                                                                                                                                                                                                                                                                                         |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str1 &gt; "Hen"</code> | <p><b>false</b></p> <p><code>str1 = "Hello"</code>. The first two characters of <code>str1</code> and <code>"Hen"</code> are the same, but the third character <code>'l'</code> of <code>str1</code> is less than the third character <code>'n'</code> of <code>"Hen"</code>. Therefore, <code>str1 &gt; "Hen"</code> is <b>false</b>.</p>              |
| <code>str3 &lt; "An"</code>  | <p><b>true</b></p> <p><code>str3 = "Air"</code>. The first characters of <code>str3</code> and <code>"An"</code> are the same, but the second character <code>'i'</code> of <code>"Air"</code> is less than the second character <code>'n'</code> of <code>"An"</code>. Therefore, <code>str3 &lt; "An"</code> is <b>true</b>.</p>                      |
| <code>str1 == "hello"</code> | <p><b>false</b></p> <p><code>str1 = "Hello"</code>. The first character <code>'H'</code> of <code>str1</code> is less than the first character <code>'h'</code> of <code>"hello"</code> because the ASCII value of <code>'H'</code> is 72, and the ASCII value of <code>'h'</code> is 104. Therefore, <code>str1 == "hello"</code> is <b>false</b>.</p> |
| <code>str3 &lt;= str4</code> | <p><b>true</b></p> <p><code>str3 = "Air"</code> and <code>str4 = "Bill"</code>. The first character <code>'A'</code> of <code>str3</code> is less than the first character <code>'B'</code> of <code>str4</code>. Therefore, <code>str3 &lt;= str4</code> is <b>true</b>.</p>                                                                           |
| <code>str2 &gt; str4</code>  | <p><b>true</b></p> <p><code>str2 = "Hi"</code> and <code>str4 = "Bill"</code>. The first character <code>'H'</code> of <code>str2</code> is greater than the first character <code>'B'</code> of <code>str4</code>. Therefore, <code>str2 &gt; str4</code> is <b>true</b>.</p>                                                                          |

If two strings of different lengths are compared and the character-by-character comparison is equal until it reaches the last character of the shorter string, the shorter string is evaluated as less than the larger string, as shown next.

| Expression                       | Value/Explanation                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str4 &gt;= "Billy"</code>  | <p><b>false</b></p> <p><code>str4 = "Bill"</code>. It has four characters, and <code>"Billy"</code> has five characters. Therefore, <code>str4</code> is the shorter string. All four characters of <code>str4</code> are the same as the corresponding first four characters of <code>"Billy"</code>, and <code>"Billy"</code> is the larger string. Therefore, <code>str4 &gt;= "Billy"</code> is <b>false</b>.</p>    |
| <code>str5 &lt;= "Bigger"</code> | <p><b>true</b></p> <p><code>str5 = "Big"</code>. It has three characters, and <code>"Bigger"</code> has six characters. Therefore, <code>str5</code> is the shorter string. All three characters of <code>str5</code> are the same as the corresponding first three characters of <code>"Bigger"</code>, and <code>"Bigger"</code> is the larger string. Therefore, <code>str5 &lt;= "Bigger"</code> is <b>true</b>.</p> |

The program `Chapter4_StringComparisons.cpp` at the Web site accompanying this book shows the results of the previous expressions.

## Logical (Boolean) Operators and Logical Expressions

This section describes how to form and evaluate logical expressions that are combinations of other logical expressions. **Logical (Boolean) operators** enable you to combine logical expressions. C++ has three logical (Boolean) operators, as shown in Table 4-2.

**TABLE 4-2** Logical (Boolean) Operators in C++

| Operator                | Description |
|-------------------------|-------------|
| <code>!</code>          | not         |
| <code>&amp;&amp;</code> | and         |
| <code>  </code>         | or          |

Logical operators take only logical values as operands and yield only logical values as results. The operator `!` is unary, so it has only one operand. The operators `&&` and `||` are binary operators. Tables 4-3, 4-4, and 4-5 define these operators.

Table 4-3 defines the operator `!` (not). When you use the `!` operator, `!true` is `false` and `!false` is `true`. Putting `!` in front of a logical expression reverses the value of that logical expression.

**TABLE 4-3** The `!` (Not) Operator

| Expression                  | <code>!(Expression)</code> |
|-----------------------------|----------------------------|
| <code>true</code> (nonzero) | <code>false</code> (0)     |
| <code>false</code> (0)      | <code>true</code> (1)      |

### EXAMPLE 4-3

| Expression                   | Value              | Explanation                                                                                                   |
|------------------------------|--------------------|---------------------------------------------------------------------------------------------------------------|
| <code>!('A' &gt; 'B')</code> | <code>true</code>  | Because <code>'A' &gt; 'B'</code> is <code>false</code> , <code>!('A' &gt; 'B')</code> is <code>true</code> . |
| <code>!(6 &lt;= 7)</code>    | <code>false</code> | Because <code>6 &lt;= 7</code> is <code>true</code> , <code>!(6 &lt;= 7)</code> is <code>false</code> .       |



Table 4-4 defines the operator `&&` (and). From this table, it follows that `Expression1 && Expression2` is **true** if and only if both `Expression1` and `Expression2` are **true**; otherwise, `Expression1 && Expression2` evaluates to **false**.

**TABLE 4-4** The `&&` (And) Operator

| Expression1           | Expression2           | Expression1 && Expression2 |
|-----------------------|-----------------------|----------------------------|
| <b>true</b> (nonzero) | <b>true</b> (nonzero) | <b>true</b> (1)            |
| <b>true</b> (nonzero) | <b>false</b> (0)      | <b>false</b> (0)           |
| <b>false</b> (0)      | <b>true</b> (nonzero) | <b>false</b> (0)           |
| <b>false</b> (0)      | <b>false</b> (0)      | <b>false</b> (0)           |

#### EXAMPLE 4-4

| Expression                                           | Value        | Explanation                                                                                                                                                                                    |
|------------------------------------------------------|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>(14 &gt;= 5) &amp;&amp; ('A' &lt; 'B')</code>  | <b>true</b>  | Because <code>(14 &gt;= 5)</code> is <b>true</b> , <code>('A' &lt; 'B')</code> is <b>true</b> , and <b>true</b> && <b>true</b> is <b>true</b> , the expression evaluates to <b>true</b> .      |
| <code>(24 &gt;= 35) &amp;&amp; ('A' &lt; 'B')</code> | <b>false</b> | Because <code>(24 &gt;= 35)</code> is <b>false</b> , <code>('A' &lt; 'B')</code> is <b>true</b> , and <b>false</b> && <b>true</b> is <b>false</b> , the expression evaluates to <b>false</b> . |

Table 4-5 defines the operator `||` (or). From this table, it follows that `Expression1 || Expression2` is **true** if and only if at least one of the expressions, `Expression1` or `Expression2`, is **true**; otherwise, `Expression1 || Expression2` evaluates to **false**.

**TABLE 4-5** The `||` (Or) Operator

| Expression1           | Expression2           | Expression1    Expression2 |
|-----------------------|-----------------------|----------------------------|
| <b>true</b> (nonzero) | <b>true</b> (nonzero) | <b>true</b> (1)            |
| <b>true</b> (nonzero) | <b>false</b> (0)      | <b>true</b> (1)            |
| <b>false</b> (0)      | <b>true</b> (nonzero) | <b>true</b> (1)            |
| <b>false</b> (0)      | <b>false</b> (0)      | <b>false</b> (0)           |

| EXAMPLE 4-5               |       |                                                                                                                    |
|---------------------------|-------|--------------------------------------------------------------------------------------------------------------------|
| Expression                | Value | Explanation                                                                                                        |
| (14 >= 5)    ('A' > 'B')  | true  | Because (14 >= 5) is true, ('A' > 'B') is false, and true    false is true, the expression evaluates to true.      |
| (24 >= 35)    ('A' > 'B') | false | Because (24 >= 35) is false, ('A' > 'B') is false, and false    false is false, the expression evaluates to false. |
| ('A' <= 'a')    (7 != 7)  | true  | Because ('A' <= 'a') is true, (7 != 7) is false, and true    false is true, the expression evaluates to true.      |

### Order of Precedence

Complex logical expressions can be difficult to evaluate. Consider the following logical expression:

11 > 5 || 6 < 15 && 7 >= 8

This logical expression yields different results, depending on whether || or && is evaluated first. If || is evaluated first, the expression evaluates to false. If && is evaluated first, the expression evaluates to true.

An expression might contain arithmetic, relational, and logical operators, as in the expression:

5 + 3 <= 9 && 2 > 3

To work with complex logical expressions, there must be some priority scheme for evaluating operators. Table 4-6 shows the order of precedence of some C++ operators,

TABLE 4-6 Precedence of Operators

| Operators                 | Precedence |
|---------------------------|------------|
| !, +, - (unary operators) | first      |
| *, /, %                   | second     |
| +, -                      | third      |
| <, <=, >=, >              | fourth     |
| ==, !=                    | fifth      |
| &&                        | sixth      |
|                           | seventh    |
| = (assignment operator)   | last       |

including the arithmetic, relational, and logical operators. (See Appendix B for the precedence of all C++ operators.)

# NOTE

In C++, `&` and `|` are also operators. The meaning of these operators is different from the meaning of `&&` and `||`. Using `&` in place of `&&` or `|` in place of `||`—as might result from a typographical error—would produce very strange results.

Using the precedence rules in an expression, relational and logical operators are evaluated from left to right. Because relational and logical operators are evaluated from left to right, the **associativity** of these operators is said to be from left to right.

Example 4-6 illustrates how logical expressions consisting of variables are evaluated.

## EXAMPLE 4-6

Suppose you have the following declarations:

```
bool found = true;
int age = 20;
double hours = 45.30;
double overTime = 15.00;
int count = 20;
char ch = 'B';
```

Consider the following expressions:

| Expression                                      | Value / Explanation                                                                                                                                                                                                                                                                   |
|-------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>!found</code>                             | <code>false</code><br>Because <code>found</code> is <code>true</code> , <code>!found</code> is <code>false</code> .                                                                                                                                                                   |
| <code>hours &gt; 40.00</code>                   | <code>true</code><br>Because <code>hours</code> is <code>45.30</code> and <code>45.30 &gt; 40.00</code> is <code>true</code> , the expression <code>hours &gt; 40.00</code> evaluates to <code>true</code> .                                                                          |
| <code>!age</code>                               | <code>false</code><br><code>age</code> is <code>20</code> , which is nonzero, so <code>age</code> is <code>true</code> . Therefore, <code>!age</code> is <code>false</code> .                                                                                                         |
| <code>!found &amp;&amp; (age &gt;= 18)</code>   | <code>false</code><br><code>!found</code> is <code>false</code> ; <code>age &gt; 18</code> is <code>20 &gt; 18</code> is <code>true</code> . Therefore, <code>!found &amp;&amp; (age &gt;= 18)</code> is <code>false &amp;&amp; true</code> , which evaluates to <code>false</code> . |
| <code>!(found &amp;&amp; (age &gt;= 18))</code> | <code>false</code><br>Now, <code>found &amp;&amp; (age &gt;= 18)</code> is <code>true &amp;&amp; true</code> , which evaluates to <code>true</code> . Therefore, <code>!(found &amp;&amp; (age &gt;= 18))</code> is <code>!true</code> , which evaluates to <code>false</code> .      |

| Expression                                                        | Value / Explanation                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>hours + overTime &lt;= 75.00</code>                         | <b>true</b><br>Because <code>hours + overTime</code> is <code>45.30 + 15.00 = 60.30</code> and <code>60.30 &lt;= 75.00</code> is <b>true</b> , it follows that <code>hours + overTime &lt;= 75.00</code> evaluates to <b>true</b> .                                                                                                                                                                   |
| <code>(count &gt;= 0) &amp;&amp;<br/>    (count &lt;= 100)</code> | <b>true</b><br>Now, <code>count</code> is 20. Because <code>20 &gt;= 0</code> is <b>true</b> , <code>count &gt;= 0</code> is <b>true</b> . Also, <code>20 &lt;= 100</code> is <b>true</b> , so <code>count &lt;= 100</code> is <b>true</b> . Therefore, <code>(count &gt;= 0) &amp;&amp; (count &lt;= 100)</code> is <b>true &amp;&amp; true</b> , which evaluates to <b>true</b> .                   |
| <code>('A' &lt;= ch &amp;&amp; ch &lt;= 'Z')</code>               | <b>true</b><br>Here, <code>ch</code> is 'B'. Because <code>'A' &lt;= 'B'</code> is <b>true</b> , <code>'A' &lt;= ch</code> evaluates to <b>true</b> . Also, because <code>'B' &lt;= 'Z'</code> is <b>true</b> , <code>ch &lt;= 'Z'</code> evaluates to <b>true</b> . Therefore, <code>('A' &lt;= ch &amp;&amp; ch &lt;= 'Z')</code> is <b>true &amp;&amp; true</b> , which evaluates to <b>true</b> . |

The following program evaluates and outputs the values of these logical expressions. Note that if a logical expression evaluates to **true**, the corresponding output is 1; if the logical expression evaluates to **false**, the corresponding output is 0, as shown in the output at the end of the program. (Recall that if the value of a logical expression is **true**, it evaluates to 1, and if the value of the logical expression is **false**, it evaluates to 0.)

#### //Chapter 4 Logical operators

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
 bool found = true;
 int age = 20;
 double hours = 45.30;
 double overTime = 15.00;
 int count = 20;
 char ch = 'B';

 cout << fixed << showpoint << setprecision(2);
 cout << "found = " << found << ", age = " << age
 << ", hours = " << hours << ", overTime = " << overTime
 << ", " << endl << "count = " << count
 << ", ch = " << ch << endl << endl;

 cout << "!found evaluates to " << !found << endl;
 cout << "hours > 40.00 evaluates to " << (hours > 40.00) << endl;
 cout << "!age evaluates to " << !age << endl;
 cout << "!found && (hours >= 0) evaluates to "
 << (!found && (hours >= 0)) << endl;
```

```

cout << "!(found && (hours >= 0)) evaluates to "
 << (!(found && (hours >= 0))) << endl;
cout << "hours + overTime <= 75.00 evaluates to "
 << (hours + overTime <= 75.00) << endl;
cout << "(count >= 0) && (count <= 100) evaluates to "
 << ((count >= 0) && (count <= 100)) << endl;
cout << "('A' <= ch && ch <= 'Z') evaluates to "
 << ('A' <= ch && ch <= 'Z') << endl;

return 0;
}

```

### Sample Run:

```

found = 1, age = 20, hours = 45.30, overTime = 15.00,
count = 20, ch = B

```

```

!found evaluates to 0
hours > 40.00 evaluates to 1
!age evaluates to 0
!found && (hours >= 0) evaluates to 0
!(found && (hours >= 0)) evaluates to 0
hours + overTime <= 75.00 evaluates to 1
(count >= 0) && (count <= 100) evaluates to 1
('A' <= ch && ch <= 'Z') evaluates to 1

```

You can insert parentheses into an expression to clarify its meaning. You can also use parentheses to override the precedence of operators. Using the standard order of precedence, the expression:

```
11 > 5 || 6 < 15 && 7 >= 8
```

is equivalent to:

```
11 > 5 || (6 < 15 && 7 >= 8)
```

In this expression, `11 > 5` is **true**, `6 < 15` is **true**, and `7 >= 8` is **false**. Substitute these values in the expression `11 > 5 || (6 < 15 && 7 >= 8)` to get **true || (true && false) = true || false = true**. Therefore, the expression `11 > 5 || (6 < 15 && 7 >= 8)` evaluates to **true**.

In C++, logical (Boolean) expressions can be manipulated or processed in either of two ways: by using **int** variables or by using **bool** variables. The following sections describe these methods.

## int Data Type and Logical (Boolean) Expressions

Earlier versions of C++ did not provide built-in data types that had logical (or Boolean) values **true** and **false**. Because logical expressions evaluate to either 1 or 0, the value of a logical expression was stored in a variable of the data type **int**. Therefore, you can use the **int** data type to manipulate logical (Boolean) expressions.

Recall that nonzero values are treated as **true**. Now, consider the declarations:

```
int legalAge;
int age;
```

and the assignment statement:

```
legalAge = 21;
```

If you regard **legalAge** as a logical variable, the value of **legalAge** assigned by this statement is **true**.

The assignment statement:

```
legalAge = (age >= 21);
```

assigns the value 1 to **legalAge** if the value of **age** is greater than or equal to 21. The statement assigns the value 0 if the value of **age** is less than 21.

## bool Data Type and Logical (Boolean) Expressions

More recent versions of C++ contain a built-in data type, **bool**, that has the logical (Boolean) values **true** and **false**. Therefore, you can manipulate logical (Boolean) expressions using the **bool** data type. Recall that in C++, **bool**, **true**, and **false** are reserved words. In addition, the identifier **true** has the value 1, and the identifier **false** has the value 0. Now, consider the following declaration:

```
bool legalAge;
int age;
```

The statement:

```
legalAge = true;
```

sets the value of the variable **legalAge** to **true**. The statement:

```
legalAge = (age >= 21);
```

assigns the value **true** to **legalAge** if the value of **age** is greater than or equal to 21. This statement assigns the value **false** to **legalAge** if the value of **age** is less than 21. For example, if the value of **age** is 25, the value assigned to **legalAge** is **true**—that is, 1. Similarly, if the value of **age** is 16, the value assigned to **legalAge** is **false**—that is, 0.

### NOTE

You can use either an **int** variable or a **bool** variable to store the value of a logical expression. For the purpose of clarity, this book uses **bool** variables to store the values of logical expressions.

## Selection: **if** and **if...else**

Although there are only two logical values, **true** and **false**, they turn out to be extremely useful because they permit programs to incorporate decision making that alters the processing flow. The remainder of this chapter discusses ways to incorporate decisions

into a program. In C++, there are two selections, or branch control structures: `if` statements and the `switch` structure. This section discusses how `if` and `if...else` statements can be used to create one-way selection, two-way selection, and multiple selections. The `switch` structure is discussed later in this chapter.

## One-Way Selection

A bank would like to send a notice to a customer if her or his checking account balance falls below the required minimum balance. That is, if the account balance is below the required minimum balance, it should send a notice to the customer; otherwise, it should do nothing. Similarly, if the policyholder of an insurance policy is a nonsmoker, the company would like to apply a 10% discount to the policy premium. Both of these examples involve one-way selection. In C++, one-way selections are incorporated using the `if` statement. The syntax of one-way selection is:

```
if (expression)
 statement
```

Note the elements of this syntax. It begins with the reserved word `if`, followed by an **expression** contained within parentheses, followed by a **statement**. Note that the parentheses around the **expression** are part of the syntax. The **expression** is sometimes called a **decision maker** because it decides whether to execute the **statement** that follows it. The **expression** is usually a logical expression. If the value of the **expression** is `true`, the **statement** executes. If the value is `false`, the **statement** does not execute and the computer goes on to the next statement in the program. The **statement** following the **expression** is sometimes called the **action statement**. Figure 4-2 shows the flow of execution of the `if` statement (one-way selection).

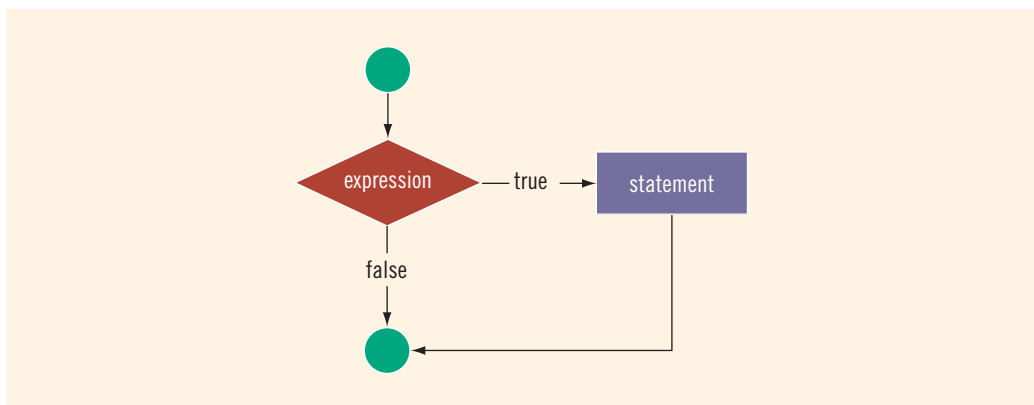


FIGURE 4-2 One-way selection

**EXAMPLE 4-7**

```
if (score >= 60)
 grade = 'P';
```

In this code, if the expression (`score >= 60`) evaluates to **true**, the assignment statement, `grade = 'P';`, executes. If the expression evaluates to **false**, the statements (if any) following the **if** structure execute. For example, if the value of `score` is 65, the value assigned to the variable `grade` is 'P'.

**EXAMPLE 4-8**

The following C++ program finds the absolute value of an integer.

```
//Program: Absolute value of an integer

#include <iostream>

using namespace std;

int main()
{
 int number, temp;

 cout << "Line 1: Enter an integer: "; //Line 1
 cin >> number; //Line 2
 cout << endl; //Line 3

 temp = number; //Line 4

 if (number < 0) //Line 5
 number = -number; //Line 6

 cout << "Line 7: The absolute value of "
 << temp << " is " << number << endl; //Line 7

 return 0;
}
```

**Sample Run:** In this sample run, the user input is shaded.

```
Line 1: Enter an integer: -6734
Line 7: The absolute value of -6734 is 6734
```

The statement in Line 1 prompts the user to enter an integer; the statement in Line 2 inputs the number into the variable `number`. The statement in Line 4 copies the value of `number` into `temp`, and the statement in Line 5 checks whether `number` is negative. If `number` is negative, the statement in Line 6 changes `number` to a positive number. The statement in Line 7 outputs the number and its absolute value. Note that because we want



to output both `number` and its absolute value, and if `number` is negative, the `if` statement changes `number` to positive, we copied the value of `number` into `temp` in Line 4.

#### EXAMPLE 4-9

Consider the following statement:

```
if score >= 60 //syntax error
 grade = 'P';
```

This statement illustrates an incorrect version of an `if` statement. The parentheses around the logical expression are missing, which is a syntax error.

Putting a semicolon after the parentheses following the `expression` in an `if` statement (that is, before the `statement`) is a semantic error. If the semicolon immediately follows the closing parenthesis, the `if` statement will operate on the empty statement.

#### EXAMPLE 4-10

Consider the following C++ statements:

```
if (score >= 60); //Line 1
 grade = 'P'; //Line 2
```

Because there is a semicolon at the end of the expression (see Line 1), the `if` statement in Line 1 terminates. The action of this `if` statement is null, and the statement in Line 2 is not part of the `if` statement in Line 1. Hence, the statement in Line 2 executes regardless of how the `if` statement evaluates.

## Two-Way Selection

There are many programming situations in which you must choose between two alternatives. For example, if a part-time employee works overtime, the paycheck is calculated using the overtime payment formula; otherwise, the paycheck is calculated using the regular formula. This is an example of two-way selection. To choose between two alternatives—that is, to implement two-way selections—C++ provides the `if...else` statement. Two-way selection uses the following syntax:

```
if (expression)
 statement1
else
 statement2
```

Take a moment to examine this syntax. It begins with the reserved word **if**, followed by a logical expression contained within parentheses, followed by a statement, followed by the reserved word **else**, followed by a second statement. Statements 1 and 2 are any valid C++ statements. In a two-way selection, if the value of the **expression** is **true**, **statement1** executes. If the value of the **expression** is **false**, **statement2** executes. Figure 4-3 shows the flow of execution of the **if...else** statement (two-way selection).

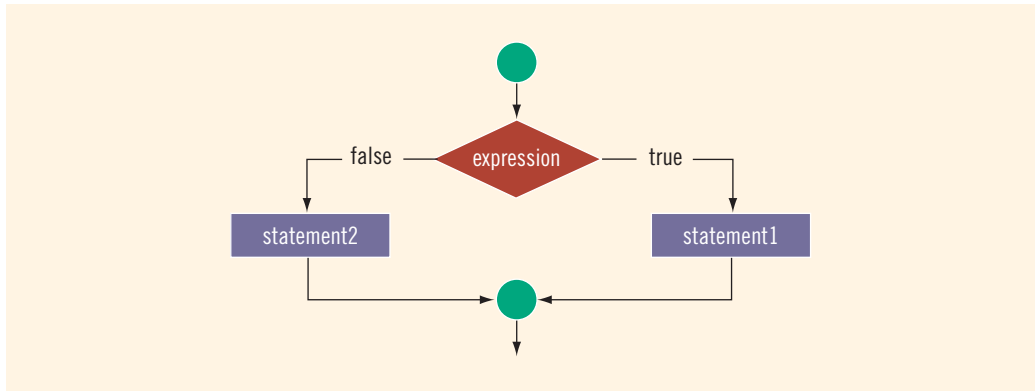


FIGURE 4-3 Two-way selection

### EXAMPLE 4-11

Consider the following statements:

```

if (hours > 40.0) //Line 1
 wages = 40.0 * rate +
 1.5 * rate * (hours - 40.0); //Line 2
else //Line 3
 wages = hours * rate; //Line 4

```

If the value of the variable **hours** is greater than 40.0, the **wages** include overtime payment. Suppose that **hours** is 50. The expression in the **if** statement, in Line 1, evaluates to **true**, so the statement in Line 2 executes. On the other hand, if **hours** is 30 or any number less than or equal to 40, the expression in the **if** statement, in Line 1, evaluates to **false**. In this case, the program skips the statement in Line 2 and executes the statement in Line 4—that is, the statement following the reserved word **else** executes.

In a two-way selection statement, putting a semicolon after the **expression** and before **statement1** creates a syntax error. If the **if** statement ends with a semicolon, **statement1** is no longer part of the **if** statement, and the **else** part of the **if...else** statement stands all by itself. There is no stand-alone **else** statement in C++. That is, it cannot be separated from the **if** statement.

**EXAMPLE 4-12**

The following statements show an example of a syntax error.

```

if (hours > 40.0); //Line 1
 wages = 40.0 * rate +
 1.5 * rate * (hours - 40.0); //Line 2
else //Line 3
 wages = hours * rate; //Line 4

```

The semicolon at the end of the **if** statement (see Line 1) ends the **if** statement, so the statement in Line 2 separates the **else** clause from the **if** statement. That is, **else** is all by itself. Because there is no stand-alone **else** statement in C++, this code generates a syntax error. As shown in Example 4-10, in a one-way selection, the semicolon at the end of an **if** statement is a logical error, whereas as shown in this example, in a two-way selection, it is a syntax error.

**4****EXAMPLE 4-13**

The following program determines an employee's weekly wages. If the hours worked exceed 40, wages include overtime payment.

```

//Program: Weekly wages

#include <iostream>
#include <iomanip>

using namespace std;

int main()

 double wages, rate, hours;

 cout << fixed << showpoint << setprecision(2); //Line 1
 cout << "Line 2: Enter working hours and rate: "; //Line 2
 cin >> hours >> rate; //Line 3

 if (hours > 40.0) //Line 4
 wages = 40.0 * rate +
 1.5 * rate * (hours - 40.0); //Line 5
 else //Line 6
 wages = hours * rate; //Line 7

 cout << endl; //Line 8
 cout << "Line 9: The wages are $" << wages
 << endl; //Line 9

 return 0;

}

```

**Sample Run:** In this sample run, the user input is shaded.

Line 2: Enter working hours and rate: 56.45 12.50

Line 9: The wages are \$808.44

The statement in Line 1 sets the output of the floating-point numbers in a fixed decimal format, with a decimal point, trailing zeros, and two decimal places. The statement in Line 2 prompts the user to input the number of hours worked and the pay rate. The statement in Line 3 inputs these values into the variables `hours` and `rate`, respectively. The statement in Line 4 checks whether the value of the variable `hours` is greater than 40.0. If `hours` is greater than 40.0, then the wages are calculated by the statement in Line 5, which includes overtime payment. Otherwise, the wages are calculated by the statement in Line 7. The statement in Line 9 outputs the wages.

Let us now consider another example of an `if` statement and examine some of the semantic errors that can occur.

#### EXAMPLE 4-14

Consider the following statements:

```
if (score >= 60) //Line 1
 cout << "Passing" << endl; //Line 2
 cout << "Failing" << endl; //Line 3
```

If the expression `(score >= 60)` evaluates to `false`, the output statement in Line 2 does not execute. So the output would be `Failing`. That is, this set of statements performs the same action as an `if...else` statement. It will execute the output statement in Line 3 rather than the output statement in Line 2. For example, if the value of `score` is 50, these statements will output the following line:

`Failing`

However, if the expression `(score >= 60)` evaluates to `true`, the program will execute both of the output statements, giving a very unsatisfactory result. For example, if the value of `score` is 70, these statements will output the following lines:

`Passing`  
`Failing`

The `if` statement controls the execution of only the statement in Line 2. The statement in Line 3 always executes.

The correct code to print `Passing` or `Failing`, depending on the value of `score`, is:

```
if (score >= 60)
 cout << "Passing" << endl;
else
 cout << "Failing" << endl;
```

## Compound (Block of) Statements

The `if` and `if...else` structures control only one statement at a time. Suppose, however, that you want to execute more than one statement if the **expression** in an `if` or `if...else` statement evaluates to `true`. To permit more complex statements, C++ provides a structure called a **compound statement** or a **block of statements**. A compound statement takes the following form:

```
{
 statement_1
 statement_2
 .
 .
 .
 statement_n
}
```

That is, a compound statement consists of a sequence of statements enclosed in curly braces, { and }. In an `if` or `if...else` structure, a compound statement functions as if it was a single statement. Thus, instead of having a simple two-way selection similar to the following code:

```
if (age >= 18)
 cout << "Eligible to vote." << endl;
else
 cout << "Not eligible to vote." << endl;
```

you could include compound statements, similar to the following code:

```
if (age >= 18)
{
 cout << "Eligible to vote." << endl;
 cout << "No longer a minor." << endl;
}
else
{
 cout << "Not eligible to vote." << endl;
 cout << "Still a minor." << endl;
}
```

The compound statement is very useful and will be used in most of the structured statements in this chapter.

## Multiple Selections: Nested `if`

In the previous sections, you learned how to implement one-way and two-way selections in a program. Some problems require the implementation of more than two alternatives. For example, suppose that if the checking account balance is more than \$50,000, the interest rate is 7%; if the balance is between \$25,000 and \$49,999.99, the interest rate is 5%; if the balance is between \$1,000 and \$24,999.99, the interest rate is 3%; otherwise,

the interest rate is 0%. This particular problem has four alternatives—that is, multiple selection paths. You can include multiple selection paths in a program by using an **if...else** structure if the action statement itself is an **if** or **if...else** statement. When one control statement is located within another, it is said to be **nested**.

Example 4-15 illustrates how to incorporate multiple selections using a nested **if...else** structure.

### EXAMPLE 4-15

Suppose that **balance** and **interestRate** are variables of type **double**. The following statements determine the **interestRate** depending on the value of the **balance**.

```

if (balance > 50000.00) //Line 1
 interestRate = 0.07; //Line 2
else //Line 3
 if (balance >= 25000.00) //Line 4
 interestRate = 0.05; //Line 5
 else //Line 6
 if (balance >= 1000.00) //Line 7
 interestRate = 0.03; //Line 8
 else //Line 9
 interestRate = 0.00; //Line 10

```

A nested **if...else** structure demands the answer to an important question: How do you know which **else** is paired with which **if**? Recall that in C++, there is no stand-alone **else** statement. Every **else** must be paired with an **if**. The rule to pair an **else** with an **if** is as follows:

**Pairing an **else** with an **if**:** In a nested **if** statement, C++ associates an **else** with the most recent incomplete **if**—that is, the most recent **if** that has not been paired with an **else**.

Using this rule, in Example 4-15, the **else** in Line 3 is paired with the **if** in Line 1. The **else** in Line 6 is paired with the **if** in Line 4, and the **else** in Line 9 is paired with the **if** in Line 7.

To avoid excessive indentation, the code in Example 4-15 can be rewritten as follows:

```

if (balance > 50000.00) //Line 1
 interestRate = 0.07; //Line 2
else if (balance >= 25000.00) //Line 3
 interestRate = 0.05; //Line 4
else if (balance >= 1000.00) //Line 5
 interestRate = 0.03; //Line 6
else //Line 7
 interestRate = 0.00; //Line 8

```

The following examples will help you to see the various ways in which you can use nested **if** structures to implement multiple selection.

**EXAMPLE 4-16**

Assume that `score` is a variable of type `int`. Based on the value of `score`, the following code outputs the grade.

```
if (score >= 90)
 cout << "The grade is A." << endl;
else if (score >= 80)
 cout << "The grade is B." << endl;
else if (score >= 70)
 cout << "The grade is C." << endl;
else if (score >= 60)
 cout << "The grade is D." << endl;
else
 cout << "The grade is F." << endl;
```

4

**EXAMPLE 4-17**

Assume that all variables are properly declared, and consider the following statements:

```
if (temperature >= 50) //Line 1
 if (temperature >= 80) //Line 2
 cout << "Good day for swimming." << endl; //Line 3
 else //Line 4
 cout << "Good day for golfing." << endl; //Line 5
else //Line 6
 cout << "Good day to play tennis." << endl; //Line 7
```

In this C++ code, the `else` in Line 4 is paired with the `if` in Line 2, and the `else` in Line 6 is paired with the `if` in Line 1. Note that the `else` in Line 4 cannot be paired with the `if` in Line 1. If you pair the `else` in Line 4 with the `if` in Line 1, the `if` in Line 2 becomes the action statement part of the `if` in Line 1, leaving the `else` in Line 6 dangling. Also, the statements in Lines 2 through 5 form the statement part of the `if` in Line 1. The indentation does not determine the pairing, but should be used to communicate the pairing.

**EXAMPLE 4-18**

Assume that all variables are properly declared, and consider the following statements:

```
if (temperature >= 70) //Line 1
 if (temperature >= 80) //Line 2
 cout << "Good day for swimming." << endl; //Line 3
 else //Line 4
 cout << "Good day for golfing." << endl; //Line 5
```

In this code, the **else** in Line 4 is paired with the **if** in Line 2. Note that for the **else** in Line 4, the most recent incomplete **if** is in Line 2. In this code, the **if** in Line 1 has no **else** and is a one-way selection. Once again, the indentation does not determine the pairing, but it communicates the pairing.

### EXAMPLE 4-19

Assume that all variables are properly declared, and consider the following statements:

```

if (gender == 'M') //Line 1
 if (age < 21) //Line 2
 policyRate = 0.05; //Line 3
 else //Line 4
 policyRate = 0.035; //Line 5
else if (gender == 'F') //Line 6
 if (age < 21) //Line 7
 policyRate = 0.04; //Line 8
 else //Line 9
 policyRate = 0.03; //Line 10

```

In this code, the **else** in Line 4 is paired with the **if** in Line 2. Note that for the **else** in Line 4, the most recent incomplete **if** is the **if** in Line 2. The **else** in Line 6 is paired with the **if** in Line 1. The **else** in Line 9 is paired with the **if** in Line 7. Once again, the indentation does not determine the pairing, but it communicates the pairing.

## Comparing **if...else** Statements with a Series of **if** Statements

Consider the following C++ program segments, all of which accomplish the same task.

```

a. if (month == 1) //Line 1
 cout << "January" << endl; //Line 2
 else if (month == 2) //Line 3
 cout << "February" << endl; //Line 4
 else if (month == 3) //Line 5
 cout << "March" << endl; //Line 6
 else if (month == 4) //Line 7
 cout << "April" << endl; //Line 8
 else if (month == 5) //Line 9
 cout << "May" << endl; //Line 10
 else if (month == 6) //Line 11
 cout << "June" << endl; //Line 12

b. if (month == 1)
 cout << "January" << endl;
 if (month == 2)
 cout << "February" << endl;
 if (month == 3)
 cout << "March" << endl;

```



```

if (month == 4)
 cout << "April" << endl;
if (month == 5)
 cout << "May" << endl;
if (month == 6)
 cout << "June" << endl;

```

Program segment (a) is written as a sequence of **if...else** statements; program segment (b) is written as a series of **if** statements. Both program segments accomplish the same thing. If **month** is 3, then both program segments output **March**. If **month** is 1, then in program segment (a), the expression in the **if** statement in Line 1 evaluates to **true**. The statement (in Line 2) associated with this **if** then executes; the rest of the structure, which is the **else** of this **if** statement, is skipped; and the remaining **if** statements are not evaluated. In program segment (b), the computer has to evaluate the expression in each **if** statement because there is no **else** statement. As a consequence, program segment (b) executes more slowly than does program segment (a).

## Short-Circuit Evaluation

Logical expressions in C++ are evaluated using a highly efficient algorithm. This algorithm is illustrated with the help of the following statements:

```

(x > y) || (x == 5) //Line 1
(a == b) && (x >= 7) //Line 2

```

In the statement in Line 1, the two operands of the operator **||** are the expressions **(x > y)** and **(x == 5)**. This expression evaluates to **true** if either the operand **(x > y)** is **true** or the operand **(x == 5)** is **true**. With short-circuit evaluation, the computer evaluates the logical expression from left to right. As soon as the value of the entire logical expression is known, the evaluation stops. For example, in statement 1, if the operand **(x > y)** evaluates to **true**, then the entire expression evaluates to **true** because **true || true** is **true** and **true || false** is **true**. Therefore, the value of the operand **(x == 5)** has no bearing on the final outcome.

Similarly, in the statement in Line 2, the two operands of the operator **&&** are **(a == b)** and **(x >= 7)**. If the operand **(a == b)** evaluates to **false**, then the entire expression evaluates to **false** because **false && true** is **false** and **false && false** is **false**.

**Short-circuit evaluation** (of a logical expression): A process in which the computer evaluates a logical expression from left to right and stops as soon as the value of the expression is known.

### EXAMPLE 4-20

Consider the following expressions:

```

(age >= 21) || (x == 5) //Line 1
(grade == 'A') && (x >= 7) //Line 2

```

For the expression in Line 1, suppose that the value of `age` is 25. Because  $(25 \geq 21)$  is `true` and the logical operator used in the expression is `||`, the expression evaluates to `true`. Due to short-circuit evaluation, the computer does not evaluate the expression  $(x == 5)$ . Similarly, for the expression in Line 2, suppose that the value of `grade` is 'B'. Because  $('B' == 'A')$  is `false` and the logical operator used in the expression is `&&`, the expression evaluates to `false`. The computer does not evaluate  $(x \geq 7)$ .

---

## Comparing Floating-Point Numbers for Equality: A Precaution

Comparison of floating-point numbers for equality may not behave as you would expect. For example, consider the following program:

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

int main()
{
 double x = 1.0;
 double y = 3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0;

 cout << fixed << showpoint << setprecision(17);

 cout << "3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 = "
 << 3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 << endl;

 cout << "x = " << x << endl;
 cout << "y = " << y << endl;

 if (x == y)
 cout << "x and y are the same." << endl;
 else
 cout << "x and y are not the same." << endl;

 if (fabs(x - y) < 0.000001)
 cout << "x and y are the same within the tolerance "
 << "0.000001." << endl;
 else
 cout << " x and y are not the same within the "
 << "tolerance 0.000001." << endl;

 return 0;
}
```

**Sample Run:**

```

3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 = 0.999999999999999989
x = 1.000000000000000000
y = 0.999999999999999989
x and y are not the same.
x and y are the same within the tolerance 0.000001.

```

In this program, `x` is initialized to 1.0 and `y` is initialized to  $3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0$ . Now, due to rounding, as shown by the output, this expression evaluates to 0.999999999999999989. Therefore, the expression `(x == y)` evaluates to `false`. However, if you evaluate the expression  $3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0$  by hand using a paper and a pencil, you will get  $3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 = (3.0 + 2.0 + 2.0) / 7.0 = 7.0 / 7.0 = 1.0$ . That is, the value of `y` should be set to 1.0.

The preceding program and its output show that you should be careful when comparing floating-point numbers for equality. One way to check whether two floating-point numbers are equal is to check whether the absolute value of their difference is less than a certain tolerance. For example, suppose the tolerance is 0.000001. Then, `x` and `y` are equal if the absolute value of `(x - y)` is less than 0.000001. To find the absolute value, you can use the function `fabs` of the header file `cmath`, as shown in the program. Therefore, the expression `fabs(x - y) < 0.000001` determines whether the absolute value of `(x - y)` is less than 0.000001.

## Associativity of Relational Operators: A Precaution

Sometimes logical expressions do not behave as you might expect, as shown by the following program, which determines if a number is between 0 and 10 (inclusive).

```

#include <iostream>

using namespace std;

int main()
{
 int num;

 cout << "Enter an integer: ";
 cin >> num;
 cout << endl;

 if (0 <= num <= 10)
 cout << num << " is within 0 and 10." << endl;
 else
 cout << num << " is not within 0 and 10." << endl;

 return 0;
}

```

**Sample Runs:** In these sample runs, the user input is shaded.

**Sample Run 1:**

Enter an integer: **5**

5 is within 0 and 10.

**Sample Run 2:**

Enter an integer: **20**

20 is within 0 and 10.

**Sample Run 3:**

Enter an integer: **-10**

-10 is within 0 and 10.

Clearly, Sample Run 1 is correct and Sample Runs 2 and 3 are incorrect. Because the **if** statement determines whether an integer is between 0 and 10, the problem is in the expression in the if statement. So, let us look at this expression, which is:

`0 <= num <= 10`

Although this statement is a legal C++ expression, you do not get the desired result. Let us evaluate this expression for certain values of `num`. Suppose that the value of `num` is 5. Then:

|                                   |                                     |                                                                                         |
|-----------------------------------|-------------------------------------|-----------------------------------------------------------------------------------------|
| <code>0 &lt;= num &lt;= 10</code> | <code>= 0 &lt;= 5 &lt;= 10</code>   |                                                                                         |
|                                   | <code>= (0 &lt;= 5) &lt;= 10</code> | (Because relational operators are evaluated from left to right)                         |
|                                   | <code>= 1 &lt;= 10</code>           | (Because <code>0 &lt;= 5</code> is <b>true</b> , <code>0 &lt;= 5</code> evaluates to 1) |
|                                   | <code>= 1   (<b>true</b>)</code>    |                                                                                         |

Now, suppose that `num = 20`. Then:

|                                   |                                      |                                                                                           |
|-----------------------------------|--------------------------------------|-------------------------------------------------------------------------------------------|
| <code>0 &lt;= num &lt;= 10</code> | <code>= 0 &lt;= 20 &lt;= 10</code>   |                                                                                           |
|                                   | <code>= (0 &lt;= 20) &lt;= 10</code> | (Because relational operators are evaluated from left to right)                           |
|                                   | <code>= 1 &lt;= 10</code>            | (Because <code>0 &lt;= 20</code> is <b>true</b> , <code>0 &lt;= 20</code> evaluates to 1) |
|                                   | <code>= 1   (<b>true</b>)</code>     |                                                                                           |

Now, you can see why the expression evaluates to **true** when `num` is 20. Similarly, if `num` is `-10`, the expression `0 <= num <= 10` evaluates to **true**. In fact, this expression will always evaluate to **true**, no matter what `num` is. This is due to the fact that the expression `0 <= num` evaluates to either 0 or 1, and `0 <= 10` is **true** and `1 <= 10` is **true**. So what is

wrong with the expression `0 <= num <= 10`? It is missing the logical operator `&&`. A correct way to write this expression in C++ is:

```
0 <= num && num <= 10
```

You must take care when formulating logical expressions. When creating a complex logical expression, you must use the proper logical operators.

## Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques

## 4

The debugging sections in Chapters 2 and 3 illustrated how to understand and fix syntax and logic errors. In this section, we illustrate how to avoid bugs by avoiding partially understood concepts and techniques.

The programs that you have written until now should have illustrated that a small error such as omission of a semicolon at the end of a variable declaration or using a variable without properly declaring it can prevent a program from successfully compiling. Similarly, using a variable without properly initializing it can prevent a program from running correctly. Recall that the condition associated with an `if` statement must be enclosed in parentheses. Therefore, the following expression will result in a syntax error.

```
if score >= 90
```

Example 4-12 illustrates that an unintended semicolon following the condition of the following `if` statement:

```
if (hours > 40.0);
```

can prevent successful compilation or correct execution.

The approach that you take to solve a problem must use concepts and techniques correctly; otherwise, your solution will be either incorrect or deficient. If you do not understand a concept or technique completely, don't use it until your understanding is complete. The problem of using partially understood concepts and techniques can be illustrated by the following program.

Suppose that we want to write a program that analyzes students' GPA. If the GPA is greater than or equal to 3.9, the student makes the dean's honor list. If the GPA is less than 2.00, the student is sent a warning letter indicating that the GPA is below the graduation requirement. So, consider the following program:

```
//GPA program with bugs.
```

```
#include <iostream>
```

```
//Line 1
```

```
using namespace std;
```

```
//Line 2
```

```
int main()
```

```
//Line 3
```

```

{ //Line 4
 double gpa; //Line 5

 cout << "Enter the GPA: "; //Line 6
 cin >> gpa; //Line 7
 cout << endl; //Line 8

 if (gpa >= 2.0) //Line 9
 if (gpa >= 3.9) //Line 10
 cout << "Dean\'s Honor List." << endl; //Line 11
 else //Line 12
 cout << "The GPA is below the graduation "
 << "requirement. \nSee your "
 << "academic advisor." << endl; //Line 13

 return 0; //Line 14
} //Line 15

```

**Sample Runs:** In these sample runs, the user input is shaded.

**Sample Run 1:**

Enter the GPA: 3.91

Dean's Honor List.

**Sample Run 2:**

Enter the GPA: 3.8

The GPA is below the graduation requirement.  
See your academic advisor.

**Sample Run 3:**

Enter the GPA: 1.95

Let us look at these sample runs. Clearly, the output in Sample Run 1 is correct. In Sample Run 2, the input is 3.8 and the output indicates that this GPA is below the graduation requirement. However, a student with a GPA of 3.8 would graduate with some type of honor. So, the output in Sample Run 2 is incorrect. In Sample Run 3, the input is 1.95, and the output does not show any warning message. Therefore, the output in Sample Run 3 is also incorrect. It means that the `if...else` statement in Lines 9 to 13 is incorrect. Let us look at these statements, that is:

```

if (gpa >= 2.0) //Line 9
 if (gpa >= 3.9) //Line 10
 cout << "Dean\'s Honor List." << endl; //Line 11
 else //Line 12
 cout << "The GPA is below the graduation "
 << "requirement. \nSee your "
 << "academic advisor." << endl; //Line 13

```

Following the rule of pairing an `else` with an `if`, the `else` in Line 12 is paired with the `if` in Line 10. In other words, using the correct indentation, the code is:

```
if (gpa >= 2.0) //Line 9
 if (gpa >= 3.9) //Line 10
 cout << "Dean's Honor List." << endl; //Line 11
 else //Line 12
 cout << "The GPA is below the graduation "
 << "requirement. \nSee your "
 << "academic advisor." << endl; //Line 13
```

Now, we can see that the `if` statement in Line 9 is a one-way selection. Therefore, if the input number is less than 2.0, no action will take place, that is, no warning message will be printed. Now, suppose the input is 3.8. Then, the expression in Line 9 evaluates to `true`, so the expression in Line 10 is evaluated, which evaluates to `false`. This means the output statement in Line 13 executes, resulting in an unsatisfactory result.

In fact, the program should print the warning message only if the GPA is less than 2.0, and it should print the message:

Dean's Honor List.

if the GPA is greater than or equal to 3.9.

To achieve that result, the `else` in Line 12 needs to be paired with the `if` in Line 9. To pair the `else` in Line 12 with the `if` in Line 9, you need to use a compound statement, as follows:

```
if (gpa >= 2.0) //Line 9
{
 if (gpa >= 3.9) //Line 10
 cout << "Dean's Honor List." << endl; //Line 11
}
else //Line 12
 cout << "The GPA is below the graduation "
 << "requirement. \nSee your "
 << "academic advisor." << endl; //Line 13
```

The correct program is as follows:

//Correct GPA program.

```
#include <iostream> //Line 1

using namespace std; //Line 2

int main() //Line 3
{
 double gpa; //Line 4
 //Line 5

 cout << "Enter the GPA: "; //Line 6
 cin >> gpa; //Line 7
 cout << endl; //Line 8
```

```

 if (gpa >= 2.0) //Line 9
 { //Line 10
 if (gpa >= 3.9) //Line 11
 cout << "Dean\'s Honor List." << endl; //Line 12
 } //Line 13
 else //Line 14
 cout << "The GPA is below the graduation "
 << "requirement. \nSee your "
 << "academic advisor." << endl; //Line 15

 return 0; //Line 16
} //Line 17

```

**Sample Runs:** In these sample runs, the user input is shaded.

**Sample Run 1:**

Enter the GPA: 3.91

Dean's Honor List.

**Sample Run 2:**

Enter the GPA: 3.8

**Sample Run 3:**

Enter the GPA: 1.95

The GPA is below the graduation requirement.  
See your academic advisor.

In cases such as this one, the general rule is that you cannot look inside of a block (that is, inside the braces) to pair an **else** with an **if**. The **else** in Line 14 cannot be paired with the **if** in Line 11 because the **if** statement in Line 11 is enclosed within braces, and the **else** in Line 14 cannot look inside those braces. Therefore, the **else** in Line 14 is paired with the **if** in Line 9.

In this book, the C++ programming concepts and techniques are presented in a logical order. When these concepts and techniques are learned one at a time in a logical order, they are simple enough to be understood completely. Understanding a concept or technique completely before using it will save you an enormous amount of debugging time.

## Input Failure and the **if** Statement

In Chapter 3, you saw that an attempt to read invalid data causes the input stream to enter a fail state. Once an input stream enters a fail state, all subsequent input statements associated with that input stream are ignored, and the computer continues to execute the program, which produces erroneous results. You can use **if** statements to check the status of an input stream variable and, if the input stream enters the fail state, include instructions that stop program execution.



In addition to reading invalid data, other events can cause an input stream to enter the fail state. Two additional common causes of input failure are the following:

- Attempting to open an input file that does not exist
- Attempting to read beyond the end of an input file

One way to address these causes of input failure is to check the status of the input stream variable. You can check the status by using the input stream variable as the logical expression in an `if` statement. If the last input succeeded, the input stream variable evaluates to `true`; if the last input failed, it evaluates to `false`.

The statement:

```
if (cin)
 cout << "Input is OK." << endl;
```

prints:

Input is OK.

if the last input from the standard input device succeeded. Similarly, if `infile` is an `ifstream` variable, the statement:

```
if (!infile)
 cout << "Input failed." << endl;
```

prints:

Input failed.

if the last input associated with the stream variable `infile` failed.

Suppose an input stream variable tries to open a file for inputting data into a program. If the input file does not exist, you can use the value of the input stream variable, in conjunction with the `return` statement, to terminate the program.

Recall that the last statement included in the function `main` is:

```
return 0;
```

This statement returns a value of `0` to the operating system when the program terminates. A value of `0` indicates that the program terminated normally and that no error occurred during program execution. Values of type `int` other than `0` can also be returned to the operating system via the `return` statement. The return of any value other than `0`, however, indicates that something went wrong during program execution.

The `return` statement can appear anywhere in the program. Whenever a `return` statement executes, it immediately exits the function in which it appears. In the case of the function `main`, the program terminates when the `return` statement executes. You can use these properties of the `return` statement to terminate the function `main` whenever the input stream fails. This technique is especially useful when a program tries to open an input file. Consider the following statements:

```

ifstream infile;

infile.open("inputdat.dat"); //open inputdat.dat

if (!infile)
{
 cout << "Cannot open the input file. "
 << "The program terminates." << endl;
 return 1;
}

```

Suppose that the file `inputdat.dat` does not exist. The operation to open this file fails, causing the input stream to enter the fail state. As a logical expression, the file stream variable `infile` then evaluates to `false`. Because `infile` evaluates to `false`, the expression `!infile` (in the `if` statement) evaluates to `true`, and the body of the `if` statement executes. The message:

Cannot open the input file. The program terminates.

is printed on the screen, and the `return` statement terminates the program by returning a value of 1 to the operating system.

Let's now use the code that responds to input failure by including these features in the Programming Example: Student Grade from Chapter 3. Recall that this program calculates the average test score based on data from an input file and then outputs the results to another file. The following programming code is the same as the code from Chapter 3, except that it includes statements to exit the program if the input file does not exist.

```

//Program to calculate the average test score.

#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;

int main()
{
 ifstream inFile; //input file stream variable
 ofstream outFile; //output file stream variable

 double test1, test2, test3, test4, test5;
 double average;

 string firstName;
 string lastName;

 inFile.open("test.txt"); //open the input file

 if (!inFile)

```

```

{
 cout << "Cannot open the input file. "
 << "The program terminates." << endl;
 return 1;
}

outFile.open("testavg.out"); //open the output file

outFile << fixed << showpoint;
outFile << setprecision(2);

cout << "Processing data" << endl;

inFile >> firstName >> lastName;
outFile << "Student name: " << firstName
 << " " << lastName << endl;

inFile >> test1 >> test2 >> test3
 >> test4 >> test5;
outFile << "Test scores: " << setw(4) << test1
 << setw(4) << test2 << setw(4) << test3
 << setw(4) << test4 << setw(4) << test5
 << endl;

average = (test1 + test2 + test3 + test4 + test5) / 5.0;

outFile << "Average test score: " << setw(6)
 << average << endl;

inFile.close();
outFile.close();

return 0;
}

```

## Confusion between the Equality Operator (==) and the Assignment Operator (=)

Recall that if the decision-making expression in the **if** statement evaluates to **true**, the **statement** part of the **if** statement executes. In addition, the **expression** is usually a logical expression. However, C++ allows you to use *any* expression that can be evaluated to either **true** or **false** as an **expression** in the **if** statement. Consider the following statement:

```

if (x = 5)
 cout << "The value is five." << endl;

```

The **expression**—that is, the decision maker—in the **if** statement is **x = 5**. The expression **x = 5** is called an assignment expression because the operator **=** appears in the expression and there is no semicolon at the end.

This expression is evaluated as follows. First, the right side of the operator **=** is evaluated, which evaluates to 5. The value 5 is then assigned to **x**. Moreover, the value 5—that is, the

new value of **x**—also becomes the value of the expression in the **if** statement—that is, the value of the assignment expression. Because 5 is nonzero, the expression in the **if** statement evaluates to **true**, so the statement part of the **if** statement outputs: **The value is five.**

No matter how experienced a programmer is, almost everyone makes the mistake of using **=** in place of **==** at one time or another. One reason why these two operators are often confused is that most programming languages use **=** as an equality operator. Thus, experience with other programming languages can create confusion. Sometimes the error is merely typographical, another reason to be careful when typing code.

Despite the fact that an assignment expression can be used as an expression, using the assignment operator in place of the equality operator can cause serious problems in a program. For example, suppose that the discount on a car insurance policy is based on the insured's driving record. A driving record of 1 means that the driver is accident-free and receives a 25% discount on the policy. The statement:

```
if (drivingCode == 1)
 cout << "The discount on the policy is 25%." << endl;
```

outputs:

**The discount on the policy is 25%.**

only if the value of **drivingCode** is 1. However, the statement:

```
if (drivingCode = 1)
 cout << "The discount on the policy is 25%." << endl;
```

always outputs:

**The discount on the policy is 25%.**

because the right side of the assignment expression evaluates to 1, which is nonzero and so evaluates to **true**. Therefore, the expression in the **if** statement evaluates to **true**, outputting the following line of text: **The discount on the policy is 25%.** Also, the value 1 is assigned to the variable **drivingCode**. Suppose that before the **if** statement executes, the value of the variable **drivingCode** is 4. After the **if** statement executes, not only is the output wrong, but the new value also replaces the old driving code.

The appearance of **=** in place of **==** resembles a *silent killer*. It is not a syntax error, so the compiler does not warn you of an error. Rather, it is a logical error.

#### NOTE

Using **=** in place of **==** can cause serious problems, especially if it happens in a looping statement. Chapter 5 discusses looping structures.

The appearance of the equality operator in place of the assignment operator can also cause errors in a program. For example, suppose **x**, **y**, and **z** are **int** variables. The statement:

```
x = y + z;
```

assigns the value of the expression `y + z` to `x`. The statement:

```
x == y + z;
```

compares the value of the expression `y + z` with the value of `x`; the value of `x` remains the same, however. If somewhere else in the program you are counting on the value of `x` being `y + z`, a logic error will occur, the program output will be incorrect, and you will receive no warning of this situation from the compiler. The compiler provides feedback only about syntax errors, not logic errors. For this reason, you must use extra care when working with the equality operator and the assignment operator.

## Conditional Operator (`?:`)

### NOTE

The reader can skip this section without any discontinuation.

Certain `if...else` statements can be written in a more concise way by using C++'s conditional operator. The **conditional operator**, written as `?:`, is a **ternary operator**, which means that it takes three arguments. The syntax for using the conditional operator is:

```
expression1 ? expression2 : expression3
```

This type of statement is called a **conditional expression**. The conditional expression is evaluated as follows: If `expression1` evaluates to a nonzero integer (that is, to `true`), the result of the conditional expression is `expression2`. Otherwise, the result of the conditional expression is `expression3`.

Consider the following statements:

```
if (a >= b)
 max = a;
else
 max = b;
```

You can use the conditional operator to simplify the writing of this `if...else` statement as follows:

```
max = (a >= b) ? a : b;
```

## Program Style and Form (Revisited): Indentation

In the section “Program Style and Form” of Chapter 2, we specified some guidelines to write programs. Now that we have started discussing control structures, in this section, we give some general guidelines to properly indent your program.

As you write programs, typos and errors are unavoidable. If your program is properly indented, you can spot and fix errors quickly, as shown by several examples in this

chapter. Typically, the IDE that you use will automatically indent your program. If for some reason your IDE does not indent your program, you can indent your program yourself.

Proper indentation can show the natural grouping of statements. You should insert a blank line between statements that are naturally separate. In this book, the statements inside braces, the statements of a selection structure, and an `if` statement within an `if` statement are all indented four spaces to the right. Throughout the book, we use four spaces to indent statements, especially to show the levels of control structures within other control structures. You can also use four spaces for indentation.

There are two commonly used styles for placing braces. In this book, we place braces on a line by themselves. Also, matching left and right braces are in the same column, that is, they are the same number of spaces away from the left side of the program. This style of placing braces easily shows the grouping of the statements and also matches left and right braces. You can also follow this style to place and indent braces.

In the second style of placing braces, the left brace need not be on a line by itself. Typically, for control structures, the left brace is placed after the last right parenthesis of the (logical) expression, and the right brace is on a line by itself. This style might save some space. However, sometimes this style might not immediately show the grouping or the block of the statements.

No matter what style of indentation you use, you should be consistent within your programs, and the indentation should show the structure of the program.

## Using Pseudocode to Develop, Test, and Debug a Program

---

There are several ways to develop a program. One method involves using an informal mixture of C++ and ordinary language, called **pseudocode** or just **pseudo**. Sometimes pseudo provides a useful means to outline and refine a program before putting it into formal C++ code. When you are constructing programs that involve complex nested control structures, pseudo can help you quickly develop the correct structure of the program and avoid making common errors.

One useful program segment determines the larger of two integers. If `x` and `y` are integers, using pseudo, you can quickly write the following:

- a.   `if (x > y) then`  
      `x is larger`
- b.   `if (y > x) then`  
      `y is larger`

If the statement in (a) is **true**, then **x** is larger. If the statement in (b) is **true**, then **y** is larger. However, for this code to work in concert to determine the larger of two integers, the computer needs to evaluate both expressions:

**(x > y)      and      (y > x)**

even if the first statement is **true**. Evaluating both expressions is a waste of computer time.

Let's rewrite this pseudo as follows:

```
if (x > y) then
 x is larger
else
 y is larger
```

Here, only one condition needs to be evaluated. This code looks okay, so let's put it into C++.

```
#include <iostream>

using namespace std;

int main()
{
 if (x > y)
```

Wait...once you begin translating the pseudo into a C++ program, you should immediately notice that there is no place to store the value of **x** or **y**. The variables were not declared, which is a very common oversight, especially for new programmers. If you examine the pseudo, you will see that the program needs three variables, and you might as well make them self-documenting. Let's start the program code again:

```
#include <iostream>

using namespace std;

int main()
{
 int num1, num2, larger; //Line 1

 if (num1 > num2); //Line 2; error
 larger = num1; //Line 3
 else //Line 4
 larger = num2; //Line 5

 return 0;
}
```

Compiling this program will result in the identification of a common syntax error (in Line 2). Recall that a semicolon cannot appear after the **expression** in the

`if...else` statement. However, even if you corrected this syntax error, the program still would not give satisfactory results because it tries to use identifiers that have no values. The variables have not been initialized, which is another common error. In addition, because there are no output statements, you would not be able to see the results of the program.

Because there are so many mistakes in the program, you should try a walk-through to see whether it works at all. You should always use a wide range of values in a walk-through to evaluate the program under as many different circumstances as possible. For example, does this program work if one number is zero, if one number is negative and the other number is positive, if both numbers are negative, or if both numbers are the same? Examining the program, you can see that it does not check whether the two numbers are equal. Taking all of these points into account, you can rewrite the program as follows:

```
//Program: Compare Numbers
//This program compares two integers and outputs the largest.

#include <iostream>

using namespace std;

int main()

 int num1, num2;

 cout << "Enter any two integers: ";
 cin >> num1 >> num2;
 cout << endl;

 cout << "The two integers entered are " << num1
 << " and " << num2 << endl;

 if (num1 > num2)
 cout << "The larger number is " << num1 << endl;
 else if (num2 > num1)
 cout << "The larger number is " << num2 << endl;
 else
 cout << "Both numbers are equal." << endl;

 return 0;
}
```

**Sample Run:** In this sample run, the user input is shaded.

```
Enter any two integers: 78 90
The two integers entered are 78 and 90
The larger number is 90
```

One thing you can learn from the preceding program is that you must first develop a program using paper and pencil. Although a program that is first written on a piece of



paper is not guaranteed to run successfully on the first try, this step is still a good starting point. On paper, it is easier to spot errors and improve the program, especially with large programs.

## switch Structures

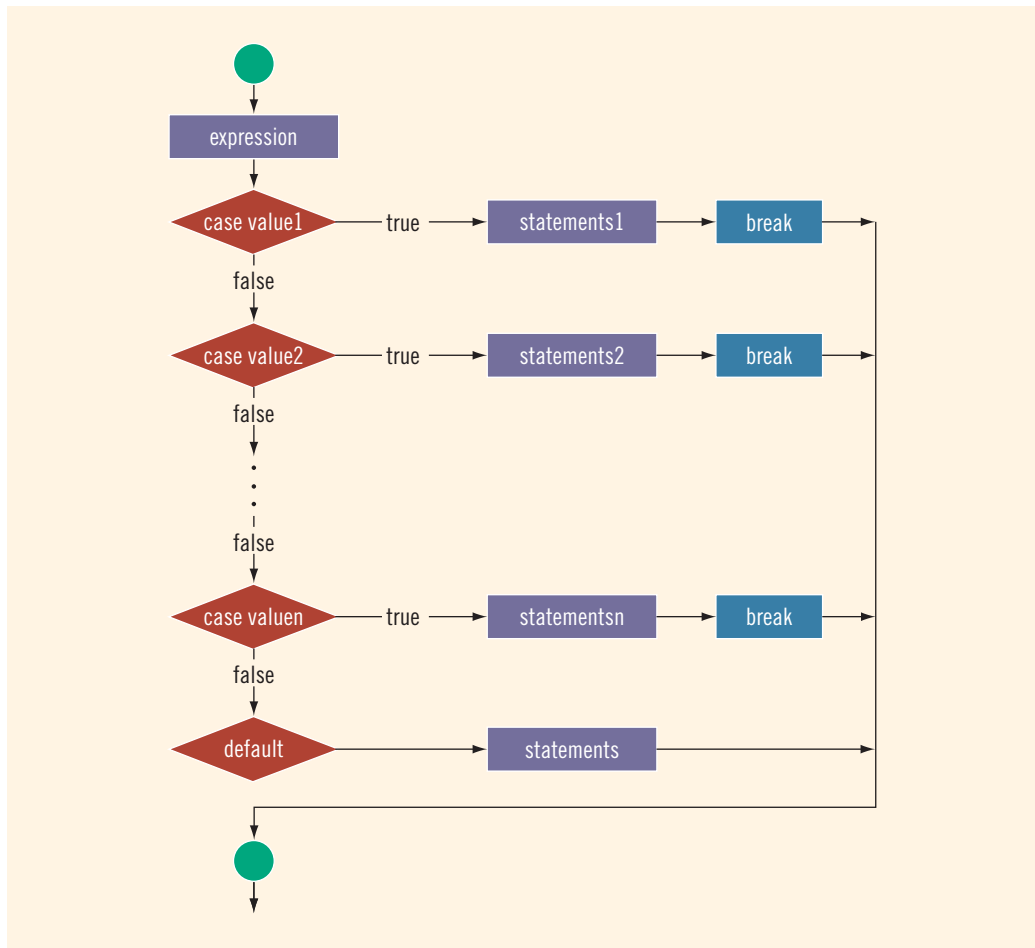
Recall that there are two selection, or branch, structures in C++. The first selection structure, which is implemented with `if` and `if...else` statements, usually requires the evaluation of a (logical) expression. The second selection structure, which does not require the evaluation of a logical expression, is called the **switch structure**. C++'s `switch` structure gives the computer the power to choose from among many alternatives.

A general syntax of the `switch` statement is:

```
switch (expression)
{
 case value1:
 statements1
 break;
 case value2:
 statements2
 break;
 .
 .
 .
 case valuen:
 statementsn
 break;
 default:
 statements
}
```

In C++, `switch`, `case`, `break`, and `default` are reserved words. In a `switch` structure, first the **expression** is evaluated. The value of the **expression** is then used to perform the actions specified in the statements that follow the reserved word `case`. Recall that in a syntax, shading indicates an optional part of the definition.

Although it need not be, the **expression** is usually an identifier. Whether it is an identifier or an expression, the value can be only integral. The **expression** is sometimes called the **selector**. Its value determines which statement is selected for execution. A particular `case` value should appear only once. One or more statements may follow a `case` label, so you do not need to use braces to turn multiple statements into a single compound statement. The `break` statement may or may not appear after each statement. Figure 4-4 shows the flow of execution of the `switch` statement.

FIGURE 4-4 `switch` statement

The `switch` statement executes according to the following rules:

1. When the value of the `expression` is matched against a `case` value (also called a label), the statements execute until either a `break` statement is found or the end of the `switch` structure is reached.
2. If the value of the `expression` does not match any of the `case` values, the statements following the `default` label execute. If the `switch` structure has no `default` label and if the value of the `expression` does not match any of the `case` values, the entire `switch` statement is skipped.
3. A `break` statement causes an immediate *exit* from the `switch` structure.

**EXAMPLE 4-21**

Consider the following statements, in which `grade` is a variable of type `char`.

```
switch (grade)
{
case 'A':
 cout << "The grade point is 4.0.";
 break;
case 'B':
 cout << "The grade point is 3.0.";
 break;
case 'C':
 cout << "The grade point is 2.0.";
 break;
case 'D':
 cout << "The grade point is 1.0.";
 break;
case 'F':
 cout << "The grade point is 0.0.";
 break;
default:
 cout << "The grade is invalid.";
}
```

In this example, the expression in the `switch` statement is a variable identifier. The variable `grade` is of type `char`, which is an integral type. The possible values of `grade` are 'A', 'B', 'C', 'D', and 'F'. Each `case` label specifies a different action to take, depending on the value of `grade`. If the value of `grade` is 'A', the output is:

The grade point is 4.0.

**EXAMPLE 4-22**

The following program illustrates the effect of the `break` statement. It asks the user to input a number between 0 and 10.

*//Program: Effect of break statements in a switch structure*

```
#include <iostream>

using namespace std;

int main()
{
 int num;

 cout << "Enter an integer between 0 and 7: "; //Line 1
 cin >> num; //Line 2
 cout << endl; //Line 3
```

```

 cout << "The number you entered is " << num
 << endl; //Line 4

 switch(num) //Line 5
 {
 case 0: //Line 6
 case 1: //Line 7
 cout << "Learning to use "; //Line 8
 case 2: //Line 9
 cout << "C++'s "; //Line 10
 case 3: //Line 11
 cout << "switch structure." << endl; //Line 12
 break; //Line 13
 case 4: //Line 14
 break; //Line 15
 case 5: //Line 16
 cout << "This program shows the effect "; //Line 17
 case 6: //Line 18
 case 7: //Line 19
 cout << "of the break statement." << endl; //Line 20
 break; //Line 21
 default: //Line 22
 cout << "The number is out of range." << endl; //Line 23
 }

 cout << "Out of the switch structure." << endl; //Line 24

 return 0; //Line 25
}

```

**Sample Runs:** These outputs were obtained by executing the preceding program several times. In each of these sample runs, the user input is shaded.

#### Sample Run 1:

```

Enter an integer between 0 and 7: 0
The number you entered is 0
Learning to use C++'s switch structure.
Out of the switch structure.

```

#### Sample Run 2:

```

Enter an integer between 0 and 7: 2
The number you entered is 2
C++'s switch structure.
Out of the switch structure.

```

#### Sample Run 3:

```

Enter an integer between 0 and 7: 4
The number you entered is 4
Out of the switch structure.

```

**Sample Run 4:**

Enter an integer between 0 and 7: **5**

The number you entered is 5

This program shows the effect of the break statement.

Out of the switch structure.

**Sample Run 5:**

Enter an integer between 0 and 7: **7**

The number you entered is 7

of the break statement.

Out of the switch structure.

**Sample Run 6:**

Enter an integer between 0 and 7: **8**

The number you entered is 8

The number is out of range.

Out of the switch structure.

A walk-through of this program, using certain values of the **switch** expression **num**, can help you understand how the **break** statement functions. If the value of **num** is 0, the value of the **switch** expression matches the **case** value 0. All statements following **case 0**: execute until a **break** statement appears.

The first **break** statement appears in Line 13, just before the **case** value of 4. Even though the value of the **switch** expression does not match any of the **case** values (that is, 1, 2, or 3), the statements following these values execute.

When the value of the **switch** expression matches a **case** value, *all* statements execute until a **break** is encountered, and the program skips all **case** labels in between. Similarly, if the value of **num** is 3, it matches the **case** value of 3, and the statements following this label execute until the **break** statement is encountered in Line 13. If the value of **num** is 4, it matches the **case** value of 4. In this situation, the action is empty because only the **break** statement, in Line 15, follows the **case** value of 4.

**EXAMPLE 4-23**

Although a **switch** structure's **case** values (labels) are limited, the **switch** statement expression can be as complex as necessary. For example, consider the following **switch** statement:

```
switch (score / 10)
{
case 0:
case 1:
case 2:
case 3:
```

```

case 4:
case 5:
 grade = 'F';
 break;
case 6:
 grade = 'D';
 break;
case 7:
 grade = 'C';
 break;
case 8:
 grade = 'B';
 break;
case 9:
case 10:
 grade = 'A';
 break;
default:
 cout << "Invalid test score." << endl;
}

```

Assume that `score` is an `int` variable with values between 0 and 100. If `score` is 75, `score / 10 = 75 / 10 = 7`, and the grade assigned is 'C'. If the value of `score` is between 0 and 59, the grade is 'F'. If `score` is between 0 and 59, then `score / 10` is 0, 1, 2, 3, 4, or 5. Each of these values corresponds to the grade 'F'.

Therefore, in this `switch` structure, the action statements of `case 0`, `case 1`, `case 2`, `case 3`, `case 4`, and `case 5` are all the same. Rather than write the statement `grade = 'F'`; followed by the `break` statement for each of the `case` values of 0, 1, 2, 3, 4, and 5, you can simplify the programming code by first specifying all of the case values (as shown in the preceding code) and then specifying the desired action statement. The `case` values of 9 and 10 follow similar conventions.

---

In addition to being a variable identifier or a complex expression, the `switch` expression can evaluate to a logical value. Consider the following statements:

```

switch (age >= 18)
{
case 1:
 cout << "Old enough to be drafted." << endl;
 cout << "Old enough to vote." << endl;
 break;
case 0:
 cout << "Not old enough to be drafted." << endl;
 cout << "Not old enough to vote." << endl;
}

```

If the value of `age` is 25, the expression `age >= 18` evaluates to 1—that is, `true`. If the `expression` evaluates to 1, the statements following the `case` label 1 execute. If the value of `age` is 14, the expression `age >= 18` evaluates to 0—that is, `false`—and the statements following the `case` label 0 execute.

You can use `true` and `false`, instead of 1 and 0, respectively, in the case labels, and rewrite the preceding `switch` statement as follows:

```
switch (age >= 18)
{
case true:
 cout << "Old enough to be drafted." << endl;
 cout << "Old enough to vote." << endl;
 break;
case false:
 cout << "Not old enough to be drafted." << endl;
 cout << "Not old enough to vote." << endl;
}
```

As you can see from the preceding examples, the `switch` statement is an elegant way to implement multiple selections. You will see the use of a `switch` statement in the programming example at the end of this chapter. Even though no fixed rules exist that can be applied to decide whether to use an `if...else` structure or a `switch` structure to implement multiple selections, the following considerations should be remembered. If multiple selections involve a range of values, you should use either an `if...else` structure or a `switch` structure, wherein you convert each range to a finite set of values.

For instance, in Example 4-23, the value of `grade` depends on the value of `score`. If `score` is between 0 and 59, `grade` is 'F'. Because `score` is an `int` variable, 60 values correspond to the grade of 'F'. If you list all 60 values as `case` values, the `switch` statement could be very long. However, dividing by 10 reduces these 60 values to only 6 values: 0, 1, 2, 3, 4, and 5.

If the range of values consists of infinitely many values and you cannot reduce them to a set containing a finite number of values, you must use the `if...else` structure. For example, if `score` happens to be a `double` variable, the number of values between 0 and 60 is infinite. However, you can use the expression `static_cast<int>(score) / 10` and still reduce this infinite number of values to just six values.

## Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques (Revisited)

Earlier in this chapter, we discussed how a partial understanding of a concept or technique can lead to errors in a program. In this section, we give another example to illustrate the problem of using partially understood concepts and techniques. In Example 4-23, we illustrate how to assign a grade based on a test score between 0 and 100. Next, consider the following program that assigns a grade based on a test score.

```
//Grade program with bugs.

#include <iostream> //Line 1

using namespace std; //Line 2
```

```

int main() //Line 3
{ //Line 4
 int testScore; //Line 5

 cout << "Enter the test score: "; //Line 6
 cin >> testScore; //Line 7
 cout << endl; //Line 8

 switch (testScore / 10) //Line 9
 { //Line 10
 case 0: //Line 11
 case 1: //Line 12
 case 2: //Line 13
 case 3: //Line 14
 case 4: //Line 15
 case 5: //Line 16
 cout << "The grade is F." << endl; //Line 17
 case 6: //Line 18
 cout << "The grade is D." << endl; //Line 19
 case 7: //Line 20
 cout << "The grade is C." << endl; //Line 21
 case 8: //Line 22
 cout << "The grade is B." << endl; //Line 23
 case 9: //Line 24
 case 10: //Line 25
 cout << "The grade is A." << endl; //Line 26
 default: //Line 27
 cout << "Invalid test score." << endl; //Line 28
 } //Line 29

 return 0; //Line 30
} //Line 31

```

**Sample Runs:** In these sample runs, the user input is shaded.

**Sample Run 1:**

Enter the test score: 110

Invalid test score.

**Sample Run 2:**

Enter the test score: -70

Invalid test score.

**Sample Run 3:**

Enter the test score: 75

The grade is C.  
The grade is B.  
The grade is A.  
Invalid test score.



From these sample runs, it follows that if the value of `testScore` is less than 0 or greater than 100, the program produces correct results, but if the value of `testScore` is between 0 and 100, say 75, the program produces incorrect results. Can you see why?

As in Sample Run 3, suppose that the value of `testScore` is 75. Then, `testScore % 10 = 7`, and this value matched the `case` label 7. So, as we indented, it should print **The grade is C.** However, the output is:

```
The grade is C.
The grade is B.
The grade is A.
Invalid test score.
```

But why? Clearly only at most one `cout` statement is associated with each `case` label. The problem is a result of having only a partial understanding of how the `switch` structure works. As we can see, the `switch` statement does not include any `break` statement. Therefore, after executing the statement(s) associated with the matching case label, execution continues with the statement(s) associated with the next case label, resulting in the printing of four unintended lines.

To output results correctly, the `switch` structure must include a `break` statement after each `cout` statement, except the last `cout` statement. We leave it as an exercise for you to modify this program so that it outputs correct results.

Once again, we can see that a partially understood concept can lead to serious errors in a program. Therefore, taking time to understand each concept and technique completely will save you hours of debugging time.

## Terminating a Program with the `assert` Function

Certain types of errors that are very difficult to catch can occur in a program. For example, division by zero can be difficult to catch using any of the programming techniques you have examined so far. C++ includes a predefined function, `assert`, that is useful in stopping program execution when certain elusive errors occur. In the case of division by zero, you can use the `assert` function to ensure that a program terminates with an appropriate error message indicating the type of error and the program location where the error occurred.

Consider the following statements:

```
int numerator;
int denominator;
int quotient;
double hours;
double rate;
double wages;
char ch;
```

1. `quotient = numerator / denominator;`
2. `if (hours > 0 && (0 < rate && rate <= 15.50))  
    wages = rate * hours;`
3. `if ('A' <= ch && ch <= 'Z')`

In the first statement, if the `denominator` is 0, logically you should not perform the division. During execution, however, the computer would try to perform the division. If the `denominator` is 0, the program would terminate with an error message stating that an illegal operation has occurred.

The second statement is designed to compute `wages` only if `hours` is greater than 0 and `rate` is positive and less than or equal to 15.50. The third statement is designed to execute certain statements only if `ch` is an uppercase letter.

For all of these statements (for that matter, in any situation in which certain conditions must be met), if conditions are not met, it would be useful to halt program execution with a message indicating where in the program an error occurred. You could handle these types of situations by including output and return statements in your program. However, C++ provides an effective method to halt a program if required conditions are not met through the `assert` function.

The syntax to use the `assert` function is:

```
assert(expression);
```

Here, `expression` is any logical expression. If `expression` evaluates to `true`, the next statement executes. If `expression` evaluates to `false`, the program terminates and indicates where in the program the error occurred.

The specification of the `assert` function is found in the header file `cassert`. Therefore, for a program to use the `assert` function, it must include the following statement:

```
#include <cassert>
```

A statement using the `assert` function is sometimes called an `assert` statement.

Returning to the preceding statements, you can rewrite statement 1 (`quotient = numerator / denominator;`) using the `assert` function. Because `quotient` should be calculated only if `denominator` is nonzero, you include an `assert` statement before the assignment statement as follows:

```
assert(denominator);
quotient = numerator / denominator;
```

Now, if `denominator` is 0, the `assert` statement halts the execution of the program with an error message similar to the following:

```
Assertion failed: denominator, file c:\temp\assert
function\assertfunction.cpp, line 20
```

This error message indicates that the assertion of `denominator` failed. The error message also gives the name of the file containing the source code and the line number where the assertion failed.

You can also rewrite statement 2 using an assertion statement as follows:

```
assert(hours > 0 && (0 < rate && rate <= 15.50));
if (hours > 0 && (0 < rate && rate <= 15.50))
 wages = rate * hours;
```

If the **expression** in the **assert** statement fails, the program terminates with an error message similar to the following:

```
Assertion failed: hours > 0 && (0 < rate && rate <= 15.50), file
c:\temp\assertfunction\assertfunction.cpp, line 26
```

During program development and testing, the **assert** statement is very useful for enforcing programming constraints. As you can see, the **assert** statement not only halts the program, but also identifies the expression where the assertion failed, the name of the file containing the source code, and the line number where the assertion failed.

Although **assert** statements are useful during program development, after a program has been developed and put into use, if an **assert** statement fails for some reason, an end user would have no idea what the error means. Therefore, after you have developed and tested a program, you might want to remove or disable the **assert** statements. In a very large program, it could be tedious, and perhaps impossible, to remove all of the **assert** statements that you used during development. In addition, if you plan to modify a program in the future, you might like to keep the **assert** statements. Therefore, the logical choice is to keep these statements but to disable them. You can disable **assert** statements by using the following preprocessor directive:

```
#define NDEBUG
```

This preprocessor directive **#define NDEBUG** must be placed *before* the directive **#include <cassert>**.

## PROGRAMMING EXAMPLE: Cable Company Billing

This programming example demonstrates a program that calculates a customer's bill for a local cable company. There are two types of customers: residential and business. There are two rates for calculating a cable bill: one for residential customers and one for business customers. For residential customers, the following rates apply:

- Bill processing fee: \$4.50
- Basic service fee: \$20.50
- Premium channels: \$7.50 per channel.

For business customers, the following rates apply:

- Bill processing fee: \$15.00
- Basic service fee: \$75.00 for first 10 connections, \$5.00 for each additional connection
- Premium channels: \$50.00 per channel for any number of connections

The program should ask the user for an account number (an integer) and a customer code. Assume that **R** or **r** stands for a residential customer, and **B** or **b** stands for a business customer

**Input**      The customer's account number, customer code, number of premium channels to which the user subscribes, and, in the case of business customers, number of basic service connections.

**Output**     Customer's account number and the billing amount.

**PROBLEM  
ANALYSIS  
AND  
ALGORITHM  
DESIGN**

The purpose of this program is to calculate and print the billing amount. To calculate the billing amount, you need to know the customer for whom the billing amount is calculated (whether the customer is residential or business) and the number of premium channels to which the customer subscribes. In the case of a business customer, you also need to know the number of basic service connections and the number of premium channels. Other data needed to calculate the bill, such as the bill processing fees and the cost of a premium channel, are known quantities. The program should print the billing amount to two decimal places, which is standard for monetary amounts. This problem analysis translates into the following algorithm:

1. Set the precision to two decimal places.
2. Prompt the user for the account number and customer type.
3. Based on the customer type, determine the number of premium channels and basic service connections, compute the bill, and print the bill:
  - a. If the customer type is **R** or **r**,
    - i. Prompt the user for the number of premium channels.
    - ii. Compute the bill.
    - iii. Print the bill.
  - b. If the customer type is **B** or **b**,
    - i. Prompt the user for the number of basic service connections and number of premium channels.
    - ii. Compute the bill.
    - iii. Print the bill.

**Variables** Because the program will ask the user to input the customer account number, customer code, number of premium channels, and number of basic service connections, you need variables to store all of this information. Also, because the program will calculate the billing amount, you need a variable to store the billing amount. Thus, the program needs at least the following variables to compute and print the bill:

```
int accountNumber; //variable to store the customer's
 //account number
char customerType; //variable to store the customer code
int numOfPremChannels; //variable to store the number
 //of premium channels to which the
 //customer subscribes
int numOfBasicServConn; //variable to store the
 //number of basic service connections
 //to which the customer subscribes
double amountDue; //variable to store the billing amount
```

**Named Constants** As you can see, the bill processing fees, the cost of a basic service connection, and the cost of a premium channel are fixed, and these values are needed to compute the bill. Although these values are constants in the program, the cable company can change them with little warning. To simplify the process of modifying the program later, instead of using these values directly in the program, you should declare them as named constants. Based on the problem analysis, you need to declare the following named constants:

```
//Named constants - residential customers
const double RES_BILL_PROC_FEES = 4.50;
const double RES_BASIC_SERV_COST = 20.50;
const double RES_COST_PREM_CHANNEL = 7.50;

//Named constants - business customers
const double BUS_BILL_PROC_FEES = 15.00;
const double BUS_BASIC_SERV_COST = 75.00;
const double BUS_BASIC_CONN_COST = 5.00;
const double BUS_COST_PREM_CHANNEL = 50.00;
```

**Formulas** The program uses a number of formulas to compute the billing amount. To compute the residential bill, you need to know only the number of premium channels to which the user subscribes. The following statement calculates the billing amount for a residential customer.

```
amountDue = RES_BILL_PROC_FEES + RES_BASIC_SERV_COST
 + numOfPremChannels * RES_COST_PREM_CHANNEL;
```

To compute the business bill, you need to know the number of basic service connections and the number of premium channels to which the user subscribes. If the number of basic service connections is less than or equal to 10, the cost of the

basic service connections is fixed. If the number of basic service connections exceeds 10, you must add the cost for each connection over 10. The following statement calculates the business billing amount.

```
if (numOfBasicServConn <= 10)
 amountDue = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST
 + numOfPremChannels * BUS_COST_PREM_CHANNEL;
else
 amountDue = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST
 + (numOfBasicServConn - 10)
 * BUS_BASIC_CONN_COST
 + numOfPremChannels * BUS_COST_PREM_CHANNEL;
```

#### MAIN ALGORITHM

Based on the preceding discussion, you can now write the main algorithm.

1. To output floating-point numbers in a fixed decimal format with a decimal point and trailing zeros, set the manipulators **fixed** and **showpoint**. Also, to output floating-point numbers with two decimal places, set the precision to two decimal places. Recall that to use these manipulators, the program must include the header file **iomanip**.
2. Prompt the user to enter the account number.
3. Get the customer account number.
4. Prompt the user to enter the customer code.
5. Get the customer code.
6. If the customer code is **r** or **R**,
  - a. Prompt the user to enter the number of premium channels.
  - b. Get the number of premium channels.
  - c. Calculate the billing amount.
  - d. Print the account number and the billing amount.
7. If the customer code is **b** or **B**,
  - a. Prompt the user to enter the number of basic service connections.
  - b. Get the number of basic service connections.
  - c. Prompt the user to enter the number of premium channels.
  - d. Get the number of premium channels.
  - e. Calculate the billing amount.
  - f. Print the account number and the billing amount.
8. If the customer code is something other than **r**, **R**, **b**, or **B**, output an error message.

For Steps 6 and 7, the program uses a **switch** statement to calculate the bill for the desired customer.

**COMPLETE PROGRAM LISTING**

```

//*****
// Author: D. S. Malik
//
// Program: Cable Company Billing
// This program calculates and prints a customer's bill for
// a local cable company. The program processes two types of
// customers: residential and business.
//*****

#include <iostream>
#include <iomanip>

using namespace std;

 //Named constants - residential customers
const double RES_BILL_PROC_FEES = 4.50;
const double RES_BASIC_SERV_COST = 20.50;
const double RES_COST_PREM_CHANNEL = 7.50;

 //Named constants - business customers
const double BUS_BILL_PROC_FEES = 15.00;
const double BUS_BASIC_SERV_COST = 75.00;
const double BUS_BASIC_CONN_COST = 5.00;
const double BUS_COST_PREM_CHANNEL = 50.00;

int main()
{
 //Variable declaration
 int accountNumber;
 char customerType;
 int numOfPremChannels;
 int numOfBasicServConn;
 double amountDue;

 cout << fixed << showpoint; //Step 1
 cout << setprecision(2); //Step 1

 cout << "This program computes a cable "
 << "bill." << endl;
 cout << "Enter account number (an integer): "; //Step 2
 cin >> accountNumber; //Step 3
 cout << endl;

 cout << "Enter customer type: "
 << "R or r (Residential), "
 << "B or b (Business): "; //Step 4
 cin >> customerType; //Step 5
 cout << endl;

```





```

 default:
 cout << "Invalid customer type." << endl; //Step 8
 }//end switch

 return 0;
}

```

**Sample Run:** In this sample run, the user input is shaded.

This program computes a cable bill.

Enter account number (an integer): 12345

Enter customer type: R or r (Residential), B or b (Business): b

Enter the number of basic service connections: 16

Enter the number of premium channels: 8

Account number: 12345

Amount due: \$520.00

## QUICK REVIEW

1. Control structures alter the normal flow of control.
2. The two most common control structures are selection and repetition.
3. Selection structures incorporate decisions in a program.
4. The relational operators are == (equality), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), and != (not equal to).
5. Including a space between the relational operators ==, <=, >=, and != creates a syntax error.
6. Characters are compared using a machine's collating sequence.
7. Logical expressions evaluate to 1 (or a nonzero value) or 0. The logical value 1 (or any nonzero value) is treated as **true**; the logical value 0 is treated as **false**.
8. In C++, **int** variables can be used to store the value of a logical expression.
9. In C++, **bool** variables can be used to store the value of a logical expression.
10. In C++, the logical operators are ! (not), && (and), and || (or).
11. There are two selection structures in C++.
12. One-way selection takes the following form:

```

if (expression)
 statement

```

If **expression** is **true**, the **statement** executes; otherwise, the computer executes the **statement** following the **if** statement.

13. Two-way selection takes the following form:

```
if (expression)
 statement1
else
 statement2
```

If **expression** is **true**, then **statement1** executes; otherwise, **statement2** executes.

14. The expression in an **if** or **if...else** structure is usually a logical expression.
15. Including a semicolon before the **statement** in a one-way selection creates a semantic error. In this case, the action of the **if** statement is empty.
16. Including a semicolon before **statement1** in a two-way selection creates a syntax error.
17. There is no stand-alone **else** statement in C++. Every **else** has a related **if**.
18. An **else** is paired with the most recent **if** that has not been paired with any other **else**.
19. A sequence of statements enclosed between curly braces, {and }, is called a compound statement or block of statements. A compound statement is treated as a single statement.
20. You can use the input stream variable in an **if** statement to determine the state of the input stream.
21. Using the assignment operator in place of the equality operator creates a semantic error. This can cause serious errors in the program.
22. The **switch** structure is used to handle multiway selection.
23. The execution of a **break** statement in a **switch** statement immediately exits the **switch** structure.
24. If certain conditions are not met in a program, the program can be terminated using the **assert** function.

## EXERCISES

---

1. Mark the following statements as true or false.
  - a. The result of a logical expression cannot be assigned to an **int** variable.
  - b. In a one-way selection, if a semicolon is placed after the expression in an **if** statement, the expression in the **if** statement is always **true**.
  - c. Every **if** statement must have a corresponding **else**.
  - d. The expression in the **if** statement:
 

```
if (score = 30)
 grade = 'A';
```

 always evaluates to **true**.

- e. The expression:  
`(ch >= 'A' && ch <= 'Z')`  
evaluates to **false** if either `ch < 'A'` or `ch >= 'Z'`.
- f. Suppose the input is 5. The output of the code:  

```
cin >> num;
if (num > 5)
 cout << num;
 num = 0;
else
 cout << "Num is zero" << endl;
is: Num is zero
```
- g. The expression in a **switch** statement should evaluate to a value of the simple data type.
- h. The expression `!(x > 0)` is **true** only if `x` is a negative number.
- i. In C++, both `!` and `!=` are logical operators.
- j. The order in which statements execute in a program is called the flow of control.

2. Circle the best answer.

- ```
a.  if (60 <= 12 * 5)
    cout << "Hello";
    cout << " There";
```

outputs the following:

- (i) Hello There (ii) Hello (iii) Hello (iv) There
There

- ```
b. if ('a' > 'b' || 66 > static_cast<int>('A'))
 cout << "#*#" << endl;
```

outputs the following:

- (i) # \* #      (ii) #      (iii) \*      (iv) none of these  
                          \*  
                          #

- ```
c. if (7 <= 7)
    cout << 6 - 9 * 2 / 6 << endl;
```

outputs the following:

- (i) -1 (ii) 3 (iii) 3.0 (iv) none of these

- ```
d. if (7 < 8)
{
 cout << "2 4 6 8" << endl;
 cout << "1 3 5 7" << endl;
}
```

outputs the following:

- (i) 2 4 6 8      (ii) 1 3 5 7      (iii) none of these  
1 3 5 7

```
e. if (5 < 3)
 cout << "*";
else if (7 == 8)
 cout << "&";
else
 cout << "$";
```

outputs the following:

- (i) \*      (ii) &      (iii) \$      (iv) none of these

3. Suppose that `x`, `y`, and `z` are `int` variables, and `x = 10`, `y = 15`, and `z = 20`. Determine whether the following expressions evaluate to `true` or `false`.

- a. `!(x > 10)`
- b. `x <= 5 || y < 15`
- c. `(x != 5) && (y != z)`
- d. `x >= z || (x + y >= z)`
- e. `(x <= y - 2) && (y >= z) || (z - 2 != 20)`

4. Suppose that `str1`, `str2`, and `str3` are `string` variables, and `str1 = "English"`, `str2 = "Computer Science"`, and `str3 = "Programming"`. Evaluate the following expressions.

- a. `str1 >= str2`
- b. `str1 != "english"`
- c. `str3 < str2`
- d. `str2 >= "Chemistry"`

5. Suppose that `x`, `y`, `z`, and `w` are `int` variables, and `x = 3`, `y = 4`, `z = 7`, and `w = 1`. What is the output of the following statements?

- a. `cout << "x == y: " << (x == y) << endl;`
- b. `cout << "x != z: " << (x != z) << endl;`
- c. `cout << "y == z - 3: " << (y == z - 3) << endl;`
- d. `cout << "!(z > w): " << !(z > w) << endl;`
- e. `cout << "x + y < z: " << (x + y < z) << endl;`

6. What is the output of the following C++ code?

```
x = 100;
y = 200;
if (x > 100 && y <= 200)
 cout << x << " " << y << " " << x + y << endl;
```

```
else
 cout << x << " " << y << " " << 2 * x - y << endl;
```

7. Correct the following code so that it prints the correct message.

```
if (score >= 60)
 cout << "You pass." << endl;
else;
 cout << "You fail." << endl;
```

8. Write C++ statements that output **Male** if the **gender** is 'M', **Female** if the **gender** is 'F', and **invalid gender** otherwise.

9. What is the output of the following program?

```
#include <iostream>

using namespace std;

int main()
{
 int myNum = 10;
 int yourNum = 30;

 if (yourNum % myNum == 3)
 {
 yourNum = 3;
 myNum = 1;
 }
 else if (yourNum % myNum == 2)
 {
 yourNum = 2;
 myNum = 2;
 }
 else
 {
 yourNum = 1;
 myNum = 3;
 }

 cout << myNum << " " << yourNum << endl;

 return 0;
}
```

10. a. What is the output of the program in Exercise 9, if **myNum** = 5 and **yourNum** = 12?  
 b. What is the output of the program in Exercise 9, if **myNum** = 30 and **yourNum** = 33?
11. Suppose that **sale** and **bonus** are **double** variables. Write an **if...else** statement that assigns a value to **bonus** as follows: If **sale** is greater than \$20,000, the value assigned to **bonus** is 0.10; If **sale** is greater than

\$10,000 and less than or equal to \$20,000, the value assigned to `bonus` is 0.05; otherwise, the value assigned to `bonus` is 0.

12. Suppose that `overSpeed` and `fine` are `double` variables. Assign the value to `fine` as follows: If  $0 < \text{overSpeed} \leq 5$ , the value assigned to `fine` is \$20.00; if  $5 < \text{overSpeed} \leq 10$ , the value assigned to `fine` is \$75.00; if  $10 < \text{overSpeed} \leq 15$ , the value assigned to `fine` is \$150.00; if  $\text{overSpeed} > 15$ , the value assigned to `fine` is \$150.00 plus \$20.00 per mile over 15.
13. Suppose that `score` is an `int` variable. Consider the following `if` statements:
 

```
if (score >= 90);
 cout << "Discount = 10%" << endl;
```

  - a. What is the output if the value of `score` is 95? Justify your answer.
  - b. What is the output if the value of `score` is 85? Justify your answer.
14. Suppose that `score` is an `int` variable. Consider the following `if` statements:
  - i. 

```
if (score == 70)
 cout << "Grade is C." << endl;
```
  - ii. 

```
if (score = 70)
 cout << "Grade is C." << endl;
```

Answer the following questions:

- a. What is the output in (i) and (ii) if the value of `score` is 70? What is the value of `score` after the `if` statement executes?
  - b. What is the output in (i) and (ii) if the value of `score` is 80? What is the value of `score` after the `if` statement executes?
15. Rewrite the following expressions using the conditional operator. (Assume that all variables are declared properly.)
  - a. 

```
if (x >= y)
 z = x - y;
else
 z = y - x;
```
  - b. 

```
if (hours >= 40.0)
 wages = 40 * 7.50 + 1.5 * 7.5 * (hours - 40);
else
 wages = hours * 7.50;
```
  - c. 

```
if (score >= 60)
 str = "Pass";
else
 str = "Fail";
```
16. Rewrite the following expressions using an `if...else` statement. (Assume that all variables are declared properly.)

- a. `(x < 5) ? y = 10 : y = 20;`
  - b. `(fuel >= 10) ? drive = 150 : drive = 30;`
  - c. `(booksBought >= 3) ? discount = 0.15 : discount = 0.0;`
17. Suppose that you have the following conditional expression. (Assume that all the variables are properly declared.)
- ```
(0 < backyard && backyard <= 5000) ? fertilizingCharges = 40.00
    : fertilizingCharges = 40.00 + (backyard - 5000) * 0.01;
```
- a. What is the value of `fertilizingCharges` if the value of `backyard` is 3000?
 - b. What is the value of `fertilizingCharges` if the value of `backyard` is 5000?
 - c. What is the value of `fertilizingCharges` if the value of `backyard` is 6500?
18. State whether the following are valid `switch` statements. If not, explain why. Assume that `n` and `digit` are `int` variables.
- a.

```
switch (n <= 2)
{
    case 0:
        cout << "Draw." << endl;
        break;
    case 1:
        cout << "Win." << endl;
        break;
    case 2:
        cout << "Lose." << endl;
        break;
}
```
 - b.

```
switch (digit / 4)
{
    case 0,
    case 1:
        cout << "low." << endl;
        break;
    case 1,
    case 2:
        cout << "middle." << endl;
        break;
    case 3:
        cout << "high." << endl;
}
```
 - c.

```
switch (n % 6)
{
    case 1:
    case 2:
    case 3:
```

```

    case 4:
    case 5:
        cout << n;
        break;
    case 0:
        cout << endl;
        break;
}

```

d. `switch (n % 10)`

```

{
    case 2:
    case 4:
    case 6:
    case 8:
        cout << "Even";
        break;
    case 1:
    case 3:
    case 5:
    case 7:
        cout << "Odd";
        break;
}

```

19. Suppose the input is 5. What is the value of **alpha** after the following C++ code executes?

```

cin >> alpha;
switch (alpha)
{
    case 1:
    case 2:
        alpha = alpha + 2;
        break;
    case 4:
        alpha++;
    case 5:
        alpha = 2 * alpha;
    case 6:
        alpha = alpha + 5;
        break;
    default:
        alpha--;
}

```

20. Suppose the input is 3. What is the value of **beta** after the following C++ code executes?

```

cin >> beta;
switch (beta)
{
    case 3:
        beta = beta + 3;
}

```



```

case 1:
    beta++;
    break;
case 5:
    beta = beta + 5;
case 4:
    beta = beta + 4;
}

```

21. Suppose the input is 6. What is the value of **a** after the following C++ code executes?

```

cin >> a;
if (a > 0)
    switch (a)
    {
        case 1:
            a = a + 3;
        case 3:
            a++;
            break;
        case 6:
            a = a + 6;
        case 8:
            a = a * 8;
            break;
        default:
            a--;
    }
else
    a = a + 2;

```

22. In the following code, correct any errors that would prevent the program from compiling or running.

```

include <iostream>

main ()
{
    int a, b;
    bool found;
    cout << "Enter two integers: ";
    cin >> a >> b;

    if a > a*b && 10 < b
        found = 2 * a > b;
    else
    {
        found = 2 * a < b;
        if found
            a = 3;
            c = 15;
            if b

```

```

        {
            b = 0;
            a = 1;
        }
    }

```

23. The following program contains errors. Correct them so that the program will run and output `w = 21`.

```

#include <iostream>

using namespace std;

const int SECRET = 5

main ()
{
    int x, y, w, z;
    z = 9;

    if z > 10
        x = 12; y = 5, w = x + y + SECRET;
    else
        x = 12; y = 4, w = x + y + SECRET;

    cout << "w = " << w << endl;
}

```

24. Write the missing statements in the following program so that it prompts the user to input two numbers. If one of the numbers is 0, the program should output a message indicating that both numbers must be nonzero. If the first number is greater than the second number, it outputs the first number divided by the second number; if the first number is less than the second number, it outputs the second number divided by the first number; otherwise, it outputs the product of the numbers.

```

#include <iostream>
using namespace std;

int main()
{
    double firstNum, secondNum;

    cout << "Enter two nonzero numbers: ";
    cin >> firstNum >> secondNum;
    cout << endl;

    //Missing statements

    return 0;
}

```

PROGRAMMING EXERCISES

1. Write a program that prompts the user to input a number. The program should then output the number and a message saying whether the number is positive, negative, or zero.
2. Write a program that prompts the user to input three numbers. The program should then output the numbers in ascending order.
3. Write a program that prompts the user to input an integer between 0 and 35. If the number is less than or equal to 9, the program should output the number; otherwise, it should output A for 10, B for 11, C for 12... and Z for 35. (*Hint:* Use the cast operator, `static_cast<char>()`, for numbers ≥ 10 .)
4. The statements in the following program are in incorrect order. Rearrange the statements so that they prompt the user to input the shape type (`rectangle`, `circle`, or `cylinder`) and the appropriate dimension of the shape. The program then outputs the following information about the shape: For a rectangle, it outputs the area and perimeter; for a circle, it outputs the area and circumference; and for a cylinder, it outputs the volume and surface area. After rearranging the statements, your program should be properly indented.

```
using namespace std;

#include <iostream>

int main()
{
    string shape;
    double height;

    #include <string>

    cout << "Enter the shape type: (rectangle, circle, cylinder) ";
    cin >> shape;
    cout << endl;

    if (shape == "rectangle")
    {
        cout << "Area of the circle = "
             << PI * pow(radius, 2.0) << endl;

        cout << "Circumference of the circle: "
             << 2 * PI * pow(radius, 2.0) << endl;

        cout << "Enter the height of the cylinder: ";
        cin >> height;
        cout << endl;

        cout << "Enter the width of the rectangle: ";
        cin >> width;
        cout << endl;
    }
}
```

```

        cout << "Perimeter of the rectangle = "
              << 2 * (length + width) << endl;
        double width;
    }

    cout << "Surface area of the cylinder: "
          << 2 * radius * + 2 * PI * pow(radius, 2.0) << endl;
    }
    else if (shape == "circle")
    {
        cout << "Enter the radius of the circle: ";
        cin >> radius;
        cout << endl;

        cout << "Volume of the cylinder = "
              << PI * pow(radius, 2.0)* height << endl;
        double length;
    }
    return 0;
    else if (shape == "cylinder")
    {
        double radius;

        cout << "Enter the length of the rectangle: ";
        cin >> length;
        cout << endl;

        #include <iomanip>

        cout << "Enter the radius of the base of the cylinder: ";
        cin >> radius;
        cout << endl;

        const double PI = 3.1416;
        cout << "Area of the rectangle = "
              << length * width << endl;
    else
        cout << "The program does not handle " << shape << endl;
        cout << fixed << showpoint << setprecision(2);

        #include <cmath>
    }
}

```

5. Write a program to implement the algorithm you designed in Exercise 21 of Chapter 1.
6. In a right triangle, the square of the length of one side is equal to the sum of the squares of the lengths of the other two sides. Write a program that prompts the user to enter the lengths of three sides of a triangle and then outputs a message indicating whether the triangle is a right triangle.

7. A box of cookies can hold 24 cookies, and a container can hold 75 boxes of cookies. Write a program that prompts the user to enter the total number of cookies, the number of cookies in a box, and the number of cookie boxes in a container. The program then outputs the number of boxes and the number of containers to ship the cookies. Note that each box must contain the specified number of cookies, and each container must contain the specified number of boxes. If the last box of cookies contains less than the number of specified cookies, you can discard it and output the number of leftover cookies. Similarly, if the last container contains less than the number of specified boxes, you can discard it and output the number of leftover boxes.
8. The roots of the quadratic equation $ax^2 + bx + c = 0$, $a \neq 0$ are given by the following formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In this formula, the term $b^2 - 4ac$ is called the **discriminant**. If $b^2 - 4ac = 0$, then the equation has a single (repeated) root. If $b^2 - 4ac > 0$, the equation has two real roots. If $b^2 - 4ac < 0$, the equation has two complex roots. Write a program that prompts the user to input the value of a (the coefficient of x^2), b (the coefficient of x), and c (the constant term) and outputs the type of roots of the equation. Furthermore, if $b^2 - 4ac \geq 0$, the program should output the roots of the quadratic equation. (*Hint*: Use the function `pow` from the header file `cmath` to calculate the square root. Chapter 3 explains how the function `pow` is used.)

9. Write a program that mimics a calculator. The program should take as input two integers and the operation to be performed. It should then output the numbers, the operator, and the result. (For division, if the denominator is zero, output an appropriate message.) Some sample outputs follow:

```
3 + 4 = 7
13 * 5 = 65
```

10. Redo Exercise 9 to handle floating-point numbers. (Format your output to two decimal places.)
11. Redo Programming Exercise 19 of Chapter 2, taking into account that your parents buy additional savings bonds for you as follows:
 - a. If you do not spend any money to buy savings bonds, then because you had a summer job, your parents buy savings bonds for you in an amount equal to 1% of the money you save after paying taxes and buying clothes, other accessories, and school supplies.
 - b. If you spend up to 25% of your net income to buy savings bonds, your parents spend \$0.25 for each dollar you spend to buy savings bonds,

plus money equal to 1% of the money you save after paying taxes and buying clothes, other accessories, and school supplies.

- c. If you spend more than 25% of your net income to buy savings bonds, your parents spend \$0.40 for each dollar you spend to buy savings bonds, plus money equal to 2% of the money you save after paying taxes and buying clothes, other accessories, and school supplies.
12. A bank in your town updates its customers' accounts at the end of each month. The bank offers two types of accounts: savings and checking. Every customer must maintain a minimum balance. If a customer's balance falls below the minimum balance, there is a service charge of \$10.00 for savings accounts and \$25.00 for checking accounts. If the balance at the end of the month is at least the minimum balance, the account receives interest as follows:
- a. Savings accounts receive 4% interest.
 - b. Checking accounts with balances of up to \$5,000 more than the minimum balance receive 3% interest; otherwise, the interest is 5%.

Write a program that reads a customer's account number (`int` type), account type (`char`; `s` for savings, `c` for checking), minimum balance that the account should maintain, and current balance. The program should then output the account number, account type, current balance, and an appropriate message. Test your program by running it five times, using the following data:

```
46728 S 1000 2700
87324 C 1500 7689
79873 S 1000 800
89832 C 2000 3000
98322 C 1000 750
```

13. Write a program that implements the algorithm given in Example 1-3 (Chapter 1), which determines the monthly wages of a salesperson.
14. The number of lines that can be printed on a paper depends on the paper size, the point size of each character in a line, whether lines are double-spaced or single-spaced, the top and bottom margin, and the left and right margins of the paper. Assume that all characters are of the same point size, and all lines are either single-spaced or double-spaced. Note that 1 inch = 72 points. Moreover, assume that the lines are printed along the width of the paper. For example, if the length of the paper is 11 inches and width is 8.5 inches, then the maximum length of a line is 8.5 inches. Write a program that calculates the number of characters in a line and the number of lines that can be printed on a paper based on the following input from the user:
- a. The length and width, in inches, of the paper
 - b. The top, bottom, left, and right margins
 - c. The point size of a line
 - d. If the lines are double-spaced, then double the point size of each character

15. Write a program that calculates and prints the bill for a cellular telephone company. The company offers two types of service: regular and premium. Its rates vary, depending on the type of service. The rates are computed as follows:

Regular service: \$10.00 plus first 50 minutes are free. Charges for over 50 minutes are \$0.20 per minute.

Premium service: \$25.00 plus:

- a. For calls made from 6:00 a.m. to 6:00 p.m., the first 75 minutes are free; charges for more than 75 minutes are \$0.10 per minute.
- b. For calls made from 6:00 p.m. to 6:00 a.m., the first 100 minutes are free; charges for more than 100 minutes are \$0.05 per minute.

Your program should prompt the user to enter an account number, a service code (type `char`), and the number of minutes the service was used. A service code of `r` or `R` means regular service; a service code of `p` or `P` means premium service. Treat any other character as an error. Your program should output the account number, type of service, number of minutes the telephone service was used, and the amount due from the user.

For the premium service, the customer may be using the service during the day and the night. Therefore, to calculate the bill, you must ask the user to input the number of minutes the service was used during the day and the number of minutes the service was used during the night.

16. Write a program to implement the algorithm that you designed in Exercise 22 of Chapter 1. (Assume that the account balance is stored in the file `Ch4_Ex16_Data.txt`.) Your program should output account balance before and after withdrawal and service charges. Also save the account balance after withdrawal in the file `Ch4_Ex16_Output.txt`.
17. You have several pictures of different sizes that you would like to frame. A local picture-framing store offers two types of frames—regular and fancy. The frames are available in white and can be ordered in any color the customer desires. Suppose that each frame is 1 inch wide. The cost of coloring the frame is \$0.10 per inch. The cost of a regular frame is \$0.15 per inch, and the cost of a fancy frame is \$0.25 per inch. The cost of putting a cardboard paper behind the picture is \$0.02 per square inch, and the cost of putting glass on top of the picture is \$0.07 per square inch. The customer can also choose to put crowns on the corners, which costs \$0.35 per crown. Write a program that prompts the user to input the following information and then output the cost of framing the picture:
- a. The length and width, in inches, of the picture
 - b. The type of the frame
 - c. Customer's choice of color to color the frame
 - d. If the user wants to put the crowns, then the number of crowns

18. Samantha and Vikas are looking to buy a house in a new development. After looking at various models, the three models they like are colonial, split-entry, and single-story. The builder gave them the base price and the finished area in square feet of the three models. They want to know the model(s) with the least price per square foot. Write a program that accepts as input the base price and the finished area in square feet of the three models. The program outputs the model(s) with the least price per square foot.
19. One way to determine how healthy a person is by measuring the body fat of the person. The formulas to determine the body fat for female and male are as follows:

Body fat formula for women:

$$A1 = (\text{body weight} \times 0.732) + 8.987$$

$$A2 = \text{wrist measurement (at fullest point)} / 3.140$$

$$A3 = \text{waist measurement (at navel)} \times 0.157$$

$$A4 = \text{hip measurement (at fullest point)} \times 0.249$$

$$A5 = \text{forearm measurement (at fullest point)} \times 0.434$$

$$B = A1 + A2 - A3 - A4 + A5$$

$$\text{Body fat} = \text{body weight} - B$$

$$\text{Body fat percentage} = \text{body fat} \times 100 / \text{body weight}$$

Body fat formula for men:

$$A1 = (\text{body weight} \times 1.082) + 94.42$$

$$A2 = \text{wrist measurement} \times 4.15$$

$$B = A1 - A2$$

$$\text{Body fat} = \text{body weight} - B$$

$$\text{Body fat percentage} = \text{body fat} \times 100 / \text{body weight}$$

Write a program to calculate the body fat of a person.



5

CHAPTER

CONTROL STRUCTURES II (REPETITION)

IN THIS CHAPTER, YOU WILL:

- Learn about repetition (looping) control structures
- Explore how to construct and use counter-controlled, sentinel-controlled, flag-controlled, and EOF-controlled repetition structures
- Examine `break` and `continue` statements
- Discover how to form and use nested control structures
- Learn how to avoid bugs by avoiding patches
- Learn how to debug loops

In Chapter 4, you saw how decisions are incorporated in programs. In this chapter, you learn how repetitions are incorporated in programs.

Why Is Repetition Needed?

Suppose you want to add five numbers to find their average. From what you have learned so far, you could proceed as follows (assume that all variables are properly declared):

```
cin >> num1 >> num2 >> num3 >> num4 >> num5; //read five numbers
sum = num1 + num2 + num3 + num4 + num5;        //add the numbers
average = sum / 5;                             //find the average
```

But suppose you want to add and average 100, 1000, or more numbers. You would have to declare that many variables and list them again in `cin` statements and, perhaps, again in the output statements. This takes an exorbitant amount of space and time. Also, if you want to run this program again with different values or with a different number of values, you have to rewrite the program.

Suppose you want to add the following numbers:

5 3 7 9 4

Consider the following statements, in which `sum` and `num` are variables of type `int`:

1. `sum = 0;`
2. `cin >> num;`
3. `sum = sum + num;`

The first statement initializes `sum` to 0. Let us execute statements 2 and 3. Statement 2 stores 5 in `num`; statement 3 updates the value of `sum` by adding `num` to it. After statement 3, the value of `sum` is 5.

Let us repeat statements 2 and 3. After statement 2 (after the programming code reads the next number):

`num = 3`

After statement 3:

`sum = sum + num = 5 + 3 = 8`

At this point, `sum` contains the sum of the first two numbers. Let us again repeat statements 2 and 3 (a third time). After statement 2 (after the code reads the next number):

`num = 7`

After statement 3:

`sum = sum + num = 8 + 7 = 15`

Now, `sum` contains the sum of the first three numbers. If you repeat statements 2 and 3 two more times, `sum` will contain the sum of all five numbers.

If you want to add 10 numbers, you can repeat statements 2 and 3 ten times. And if you want to add 100 numbers, you can repeat statements 2 and 3 one hundred times. In either case, you do not have to declare any additional variables, as you did in the first code. You can use this C++ code to add any set of numbers, whereas the earlier code requires you to drastically change the code.

There are many other situations in which it is necessary to repeat a set of statements. For example, for each student in a class, the formula for determining the course grade is the same. C++ has three repetition, or looping, structures that let you repeat statements over and over until certain conditions are met. This chapter introduces all three looping (repetition) structures. The next section discusses the first repetition structure, called the **while** loop.

while Looping (Repetition) Structure

In the previous section, you saw that sometimes it is necessary to repeat a set of statements several times. One way to repeat a set of statements is to type the set of statements in the program over and over. For example, if you want to repeat a set of statements 100 times, you type the set of statements 100 times in the program. However, this solution of repeating a set of statements is impractical, if not impossible. Fortunately, there is a better way to repeat a set of statements. As noted earlier, C++ has three repetition, or looping, structures that allow you to repeat a set of statements until certain conditions are met. This section discusses the first looping structure, called a **while** loop.

The general form of the **while** statement is:

```
while (expression)
    statement
```

In C++, **while** is a reserved word. Of course, the **statement** can be either a simple or compound statement. The **expression** acts as a **decision maker** and is usually a logical expression. The **statement** is called the body of the loop. Note that the parentheses around the **expression** are part of the syntax. Figure 5-1 shows the flow of execution of a **while** loop.

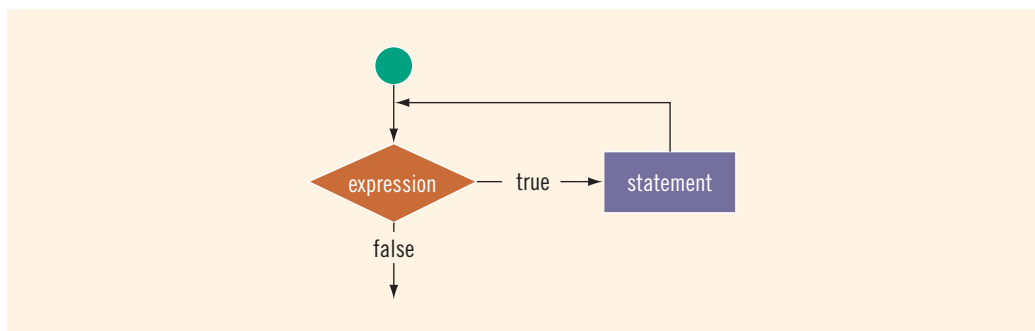


FIGURE 5-1 **while** loop

The **expression** provides an entry condition. If it initially evaluates to **true**, the **statement** executes. The loop condition—the **expression**—is then reevaluated. If it again evaluates to **true**, the **statement** executes again. The **statement** (body of the loop) continues to execute until the **expression** is no longer **true**. A loop that continues to execute endlessly is called an **infinite loop**. To avoid an infinite loop, make sure that the loop's body contains statement(s) that assure that the exit condition—the expression in the **while** statement—will eventually be **false**.

EXAMPLE 5-1

Consider the following C++ program segment: (Assume that **i** is an **int** variable.)

```
i = 0;                                //Line 1
while (i <= 20)                       //Line 2
{
    cout << i << " ";               //Line 3
    i = i + 5;                      //Line 4
}

cout << endl;
```

Sample Run:

```
0 5 10 15 20
```

In Line 1, the variable **i** is set to 0. The **expression** in the **while** statement (in Line 2), **i <= 20**, is evaluated. Because the expression **i <= 20** evaluates to **true**, the body of the **while** loop executes next. The body of the **while** loop consists of the statements in Lines 3 and 4. The statement in Line 3 outputs the value of **i**, which is 0. The statement in Line 4 changes the value of **i** to 5. After executing the statements in Lines 3 and 4, the **expression** in the **while** loop (Line 2) is evaluated again. Because **i** is 5, the expression **i <= 20** evaluates to **true** and the body of the **while** loop executes again. This process of evaluating the **expression** and executing the body of the **while** loop continues until the **expression**, **i <= 20** (in Line 2), no longer evaluates to **true**.

The variable **i** (in Line 2, Example 5-1) in the expression is called the **loop control variable**.

Note the following from Example 5-1:

- Within the loop, **i** becomes 25 but is not printed because the entry condition is **false**.
- If you omit the statement:

```
i = i + 5;
```

from the body of the loop, you will have an infinite loop, continually printing rows of zeros.

- c. You must initialize the loop control variable `i` before you execute the loop. If the statement:

```
i = 0;
```

(in Line 1) is omitted, the loop may not execute at all. (Recall that variables in C++ are not automatically initialized.)

- d. In Example 5-1, if the two statements in the body of the loop are interchanged, it may drastically alter the result. For example, consider the following statements:

```
i = 0;

while (i <= 20)
{
    i = i + 5;
    cout << i << " ";
}

cout << endl;
```

Here, the output is:

```
5 10 15 20 25
```

Typically, this would be a semantic error because you rarely want a condition to be true for `i <= 20` and yet produce results for `i > 20`.

- e. If you put a semicolon at the end of the `while` loop, (after the logical expression), then the action of the `while` loop is empty or null. For example, the action of the following `while` loop is empty.

```
i = 0;

while (i <= 20);
{
    i = i + 5;
    cout << i << " ";
}

cout << endl;
```

The statements within the braces do not form the body of the `while` loop.

Designing while Loops

As in Example 5-1, the body of a `while` executes only when the **expression**, in the `while` statement, evaluates to `true`. Typically, the **expression** checks whether a variable(s), called the **loop control variable (LCV)**, satisfies certain conditions. For example, in Example 5-1, the **expression** in the `while` statement checks whether `i <= 20`. The LCV must be properly initialized before the `while` loop, and it should

eventually make the **expression** evaluate to **false**. We do this by updating or reinitializing the LCV in the body of the **while** loop. Therefore, typically, **while** loops are written in the following form:

```
//initialize the loop control variable(s)

while (expression) //expression tests the LCV
{
    .
    .
    .
    //update the loop control variable(s)
    .
    .
    .
}
```

For instance, in Example 5-1, the statement in Line 1 initializes the LCV **i** to 0. The expression, **i <= 20**, in Line 2, checks whether **i** is less than or equal to 20, and the statement in Line 4 updates the value of **i**.

EXAMPLE 5-2

Consider the following C++ program segment:

```
i = 20; //Line 1
while (i < 20) //Line 2
{
    cout << i << " "; //Line 3
    i = i + 5; //Line 4
}
cout << endl; //Line 5
```

It is easy to overlook the difference between this example and Example 5-1. In this example, in Line 1, **i** is set to 20. Because **i** is 20, the expression **i < 20** in the **while** statement (Line 2) evaluates to **false**. Because initially the loop entry condition, **i < 20**, is **false**, the body of the **while** loop never executes. Hence, no values are output, and the value of **i** remains 20.

The next few sections describe the various forms of **while** loops.

Case 1: Counter-Controlled **while** Loops

Suppose you know exactly how many times certain statements need to be executed. For example, suppose you know exactly how many pieces of data (or entries) need to be read. In such cases, the **while** loop assumes the form of a **counter-controlled while loop**. Suppose that a set of statements needs to be executed **N** times. You can set up a **counter**

(initialized to 0 before the **while** statement) to track how many items have been read. Before executing the body of the **while** statement, the **counter** is compared with **N**. If **counter < N**, the body of the **while** statement executes. The body of the loop continues to execute until the value of **counter >= N**. Thus, inside the body of the **while** statement, the value of **counter** increments after it reads a new item. In this case, the **while** loop might look like the following:

```
counter = 0;           //initialize the loop control variable

while (counter < N) //test the loop control variable
{
    .
    .
    .
    counter++;        //update the loop control variable
    .
    .
    .
}
```

If **N** represents the number of data items in a file, then the value of **N** can be determined several ways. The program can prompt you to specify the number of items in the file; an input statement can read the value; or you can specify the first item in the file as the number of items in the file, so that you need not remember the number of input values (items). This is useful if someone other than the programmer enters the data. Consider Example 5-3.

EXAMPLE 5-3

Suppose the input is:

```
8 9 2 3 90 38 56 8 23 89 7 2
```

Suppose you want to add these numbers and find their average. Consider the following program:

//Program: Counter-Controlled Loop

```
#include <iostream>

using namespace std;

int main()
{
    int limit;    //store the number of data items
    int number;   //variable to store the number
    int sum;      //variable to store the sum
    int counter;  //loop control variable

    cout << "Line 1: Enter the number of "
         << "integers in the list: ";           //Line 1
    cin >> limit;                               //Line 2
    cout << endl;                               //Line 3
```

```

sum = 0; //Line 4
counter = 0; //Line 5

cout << "Line 6: Enter " << limit
    << " integers." << endl; //Line 6

while (counter < limit) //Line 7
{
    cin >> number; //Line 8
    sum = sum + number; //Line 9
    counter++; //Line 10
}

cout << "Line 11: The sum of the " << limit
    << " numbers = " << sum << endl; //Line 11

if (counter != 0) //Line 12
    cout << "Line 13: The average = "
        << sum / counter << endl; //Line 13
else //Line 14
    cout << "Line 15: No input." << endl; //Line 15

return 0; //Line 16
}

```

Sample Run: In this sample run, the user input is shaded.

Line 1: Enter the number of integers in the list: 12

Line 6: Enter 12 integers.

8 9 2 3 90 38 56 8 23 89 7 2

Line 11: The sum of the 12 numbers = 335

Line 13: The average = 27

This program works as follows. The statement in Line 1 prompts the user to input the number of data items. The statement in Line 2 reads the next input line and stores it in the variable `limit`. The value of `limit` indicates the number of items in the list. The statements in Lines 4 and 5 initialize the variables `sum` and `counter` to 0. (The variable `counter` is the loop control variable.) The statement in Line 6 prompts the user to input numbers. (In this sample run, the user is prompted to enter 12 integers.) The `while` statement in Line 7 checks the value of `counter` to determine how many items have been read. If `counter` is less than `limit`, the `while` loop proceeds for the next iteration. The statement in Line 8 reads the next number and stores it in the variable `number`. The statement in Line 9 updates the value of `sum` by adding the value of `number` to the previous value, and the statement in Line 10 increments the value of `counter` by 1. The statement in Line 11 outputs the sum of the numbers; the statements in Lines 12 through 15 output the average.

Note that `sum` is initialized to 0 in Line 4 in this program. In Line 9, after reading a number at Line 8, the program adds it to the sum of all the numbers scanned before the current number. The first number read will be added to zero (because `sum` is initialized to 0), giving the correct sum of the first number. To find the average, divide `sum` by `counter`. If `counter`

is 0, then dividing by zero will terminate the program and you get an error message. Therefore, before dividing `sum` by `counter`, you must check whether or not `counter` is 0.

Notice that in this program, the statement in Line 5 initializes the LCV `counter` to 0. The expression `counter < limit` in Line 7 evaluates whether `counter` is less than `limit`. The statement in Line 10 updates the value of `counter`.

Case 2: Sentinel-Controlled while Loops

You do not always know how many pieces of data (or entries) need to be read, but you may know that the last entry is a special value, called a **sentinel**. In this case, you read the first item before the `while` statement. If this item does not equal the sentinel, the body of the `while` statement executes. The `while` loop continues to execute as long as the program has not read the sentinel. Such a `while` loop is called a **sentinel-controlled while loop**. In this case, a `while` loop might look like the following:

```
cin >> variable;           //initialize the loop control variable

while (variable != sentinel) //test the loop control variable
{
    .
    .
    .
    cin >> variable;       //update the loop control variable
    .
    .
}
```

EXAMPLE 5-4

Suppose you want to read some positive integers and average them, but you do not have a preset number of data items in mind. Suppose the number -999 marks the end of the data. You can proceed as follows.

//Program: Sentinel-Controlled Loop

```
#include <iostream>

using namespace std;

const int SENTINEL = -999;

int main()
{
    int number;           //variable to store the number
    int sum = 0;          //variable to store the sum
    int count = 0;        //variable to store the total
                          //numbers read
```

```

cout << "Line 1: Enter integers ending with "
    << SENTINEL << endl;           //Line 1
cin >> number;                       //Line 2

while (number != SENTINEL)          //Line 3
{
    sum = sum + number;              //Line 4
    count++;                         //Line 5
    cin >> number;                   //Line 6
}

cout << "Line 7: The sum of the " << count
    << " numbers is " << sum << endl; //Line 7

if (count != 0)                     //Line 8
    cout << "Line 9: The average is "
        << sum / count << endl;     //Line 9
else                                 //Line 10
    cout << "Line 11: No input." << endl; //Line 11

return 0;
}

```

Sample Run: In this sample run, the user input is shaded.

```

Line 1: Enter integers ending with -999
34 23 9 45 78 0 77 8 3 5 -999
Line 7: The sum of the 10 numbers is 282
Line 9: The average is 28

```

This program works as follows. The statement in Line 1 prompts the user to enter numbers ending with -999. The statement in Line 2 reads the first number and stores it in `number`. The `while` statement in Line 3 checks whether `number` is not equal to `SENTINEL`. (The variable `number` is the loop control variable.) If `number` is not equal to `SENTINEL`, the body of the `while` loop executes. The statement in Line 4 updates the value of `sum` by adding `number` to it. The statement in Line 5 increments the value of `count` by 1; the statement in Line 6 reads and stores the next number into `number`. The statements in Lines 4 through 6 repeat until the program reads the `SENTINEL`. The statement in Line 7 outputs the sum of the numbers, and the statements in Lines 8 through 10 output the average of the numbers.

Notice that the statement in Line 2 initializes the LCV `number`. The expression `number != SENTINEL` in Line 3 checks whether the value of `number` is not equal to `SENTINEL`. The statement in Line 6 reinitializes the LCV `number`.

Next, consider another example of a sentinel-controlled `while` loop. In this example, the user is prompted to enter the value to be processed. If the user wants to stop the program, he or she can enter the sentinel.

EXAMPLE 5-5

Telephone Digits

The following program reads the letter codes A to Z and prints the corresponding telephone digit. This program uses a sentinel-controlled **while** loop. To stop the program, the user is prompted for the sentinel, which is #. This is also an example of a nested control structure, in which **if...else**, **switch**, and the **while** loop are nested.

```
//*****
// Program: Telephone Digits
// This is an example of a sentinel-controlled loop. This
// program converts uppercase letters to their corresponding
// telephone digits.
//*****

#include <iostream>

using namespace std;

int main()
{
    char letter;                                //Line 1

    cout << "Program to convert uppercase "
         << "letters to their corresponding "
         << "telephone digits." << endl;        //Line 2

    cout << "To stop the program enter #."
         << endl;                                //Line 3

    cout << "Enter a letter: ";                //Line 4
    cin >> letter;                             //Line 5
    cout << endl;                             //Line 6

    while (letter != '#')                      //Line 7
    {
        cout << "The letter you entered is: "
             << letter << endl;                //Line 8
        cout << "The corresponding telephone "
             << "digit is: ";                  //Line 9

        if (letter >= 'A' && letter <= 'Z')    //Line 10
            switch (letter)                   //Line 11
            {
                case 'A':
                case 'B':
                case 'C':
                    cout << 2 << endl;          //Line 12
                    break;                     //Line 13
            }
    }
}
```

```

        case 'D':
        case 'E':
        case 'F':
            cout << 3 << endl;           //Line 14
            break;                       //Line 15
        case 'G':
        case 'H':
        case 'I':
            cout << 4 << endl;           //Line 16
            break;                       //Line 17
        case 'J':
        case 'K':
        case 'L':
            cout << 5 << endl;           //Line 18
            break;                       //Line 19
        case 'M':
        case 'N':
        case 'O':
            cout << 6 << endl;           //Line 20
            break;                       //Line 21
        case 'P':
        case 'Q':
        case 'R':
        case 'S':
            cout << 7 << endl;           //Line 22
            break;                       //Line 23
        case 'T':
        case 'U':
        case 'V':
            cout << 8 << endl;           //Line 24
            break;                       //Line 25
        case 'W':
        case 'X':
        case 'Y':
        case 'Z':
            cout << 9 << endl;           //Line 26
    }
    else //Line 27
        cout << "Invalid input." << endl; //Line 28

    cout << "\nEnter another uppercase "
        << "letter to find its "
        << "corresponding telephone digit."
        << endl; //Line 29
    cout << "To stop the program enter #."
        << endl; //Line 30

    cout << "Enter a letter: "; //Line 31
    cin >> letter; //Line 32
    cout << endl; //Line 33
} //end while

return 0;

}

```

Sample Run: In this sample run, the user input is shaded.

Program to convert uppercase letters to their corresponding telephone digits.

To stop the program enter #.

Enter a letter: **A**

The letter you entered is: A

The corresponding telephone digit is: 2

Enter another uppercase letter to find its corresponding telephone digit.

To stop the program enter #.

Enter a letter: **D**

The letter you entered is: D

The corresponding telephone digit is: 3

Enter another uppercase letter to find its corresponding telephone digit.

To stop the program enter #.

Enter a letter: **#**

This program works as follows. The statements in Lines 2 and 3 tell the user what to do. The statement in Line 4 prompts the user to input a letter; the statement in Line 5 reads and stores that letter into the variable `letter`. The **while** loop in Line 7 checks that the letter is `#`. If the letter entered by the user is not `#`, the body of the **while** loop executes. The statement in Line 8 outputs the letter entered by the user. The **if** statement in Line 10 checks whether the letter entered by the user is uppercase. The statement part of the **if** statement is the **switch** statement (Line 11). If the letter entered by the user is uppercase, the **expression** in the **if** statement (in Line 10) evaluates to **true** and the **switch** statement executes; if the letter entered by the user is not uppercase, the **else** statement (Line 27) executes. The statements in Lines 12 through 26 determine the corresponding telephone digit.

Once the current letter is processed, the statements in Lines 29 and 30 again inform the user what to do next. The statement in Line 31 prompts the user to enter a letter; the statement in Line 32 reads and stores that letter into the variable `letter`. (Note that the statement in Line 29 is similar to the statement in Line 2 and that the statements in Lines 30 through 33 are the same as the statements in Lines 3 through 6.) After the statement in Line 33 (at the end of the **while** loop) executes, the control goes back to the top of the **while** loop and the same process begins again. When the user enters `#`, the program terminates.

Notice that in this program, the variable `letter` is the loop control variable. First, it is initialized in Line 5 by the input statement, and then it is updated in Line 32. The expression in Line 7 checks whether `letter` is `#`.

NOTE

In the program in Example 5-5, you can write the statements between Lines 10 and 28 using a **switch** structure. (See Programming Exercise 3 at the end of this chapter.)

Case 3: Flag-Controlled **while** Loops

A **flag-controlled while loop** uses a **bool** variable to control the loop. Suppose found is a **bool** variable. The flag-controlled **while** loop takes the following form:

```

found = false;           //initialize the loop control variable

while (!found)           //test the loop control variable
{
    .
    .
    .
    if (expression)
        found = true; //update the loop control variable
    .
    .
    .
}

```

The variable `found`, which is used to control the execution of the `while` loop, is called a **flag variable**.

Example 5-6 further illustrates the use of a flag-controlled `while` loop.

EXAMPLE 5-6

Number Guessing Game

The following program randomly generates an integer greater than or equal to 0 and less than 100. The program then prompts the user to guess the number. If the user guesses the number correctly, the program outputs an appropriate message. Otherwise, the program checks whether the guessed number is less than the random number. If the guessed number is less than the random number generated by the program, the program outputs the message “Your guess is lower than the number. Guess again!”; otherwise, the program outputs the message “Your guess is higher than the number. Guess again!”. The program then prompts the user to enter another number. The user is prompted to guess the random number until the user enters the correct number.

To generate a random number, you can use the function `rand` of the header file `cstdlib`. For example, the expression `rand()` returns an `int` value between 0 and 32767. Therefore, the statement:

```
cout << rand() << ", " << rand() << endl;
```

will output two numbers that appear to be random. However, each time the program is run, this statement will output the same random numbers. This is because the function `rand` uses an algorithm that produces the same sequence of random numbers each time the program is executed on the same system. To generate different random numbers each time the program is executed, you also use the function `srand` of the header file `cstdlib`. The function `srand` takes as input an `unsigned int`, which acts as the seed for the algorithm. By specifying different seed values, each time the program is executed, the function `rand` will generate a different sequence of random numbers. To specify a different seed, you can use the function `time` of the header file `ctime`, which returns the number of seconds elapsed since January 1, 1970. For example, consider the following statements:

```
srand(time(0));
num = rand() % 100;
```

The first statement sets the seed, and the second statement generates a random number greater than or equal to 0 and less than 100. Note how the function `time` is used. It is used with an argument, that is, parameter, which is 0.

The program uses the `bool` variable `isGuessed` to control the loop. The `bool` variable `isGuessed` is initialized to `false`. It is set to `true` when the user guesses the correct number.

```
//Flag-controlled while loop.
//Number guessing game.

#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
    //declare the variables
    int num;           //variable to store the random
                      //number
    int guess;         //variable to store the number
                      //guessed by the user
    bool isGuessed;    //boolean variable to control
                      //the loop

    srand(time(0));    //Line 1
    num = rand() % 100; //Line 2

    isGuessed = false; //Line 3

    while (!isGuessed) //Line 4
    {                  //Line 5
        cout << "Enter an integer greater"
              << " than or equal to 0 and "
              << "less than 100: "; //Line 6

        cin >> guess; //Line 7
        cout << endl; //Line 8

        if (guess == num) //Line 9
        {                  //Line 10
            cout << "You guessed the correct "
                  << "number." << endl; //Line 11
            isGuessed = true; //Line 12
        }                  //Line 13
        else if (guess < num) //Line 14
            cout << "Your guess is lower than the "
                  << "number.\n Guess again!"
                  << endl; //Line 15
    }
}
```

```

        else //Line 16
            cout << "Your guess is higher than "
                << "the number.\n Guess again!"
                << endl; //Line 17
    } //end while //Line 18

    return 0;
}

```

Sample Run: In this sample run, the user input is shaded.

Enter an integer greater than or equal to 0 and less than 100: 45

Your guess is higher than the number.

Guess again!

Enter an integer greater than or equal to 0 and less than 100: 20

Your guess is lower than the number.

Guess again!

Enter an integer greater than or equal to 0 and less than 100: 35

Your guess is higher than the number.

Guess again!

Enter an integer greater than or equal to 0 and less than 100: 28

Your guess is lower than the number.

Guess again!

Enter an integer greater than or equal to 0 and less than 100: 32

You guessed the correct number.

The preceding program works as follows: The statement in Line 2 creates an integer greater than or equal to 0 and less than 100 and stores this number in the variable `num`. The statement in Line 3 sets the `bool` variable `isGuessed` to `false`. The expression in the `while` loop at Line 4 evaluates the expression `!isGuessed`. If `isGuessed` is `false`, then `!isGuessed` is `true` and the body of the `while` loop executes; if `isGuessed` is `true`, then `!isGuessed` is `false`, so the `while` loop terminates.

The statement in Line 6 prompts the user to enter an integer greater than or equal to 0 and less than 100. The statement in Line 7 stores the number entered by the user in the variable `guess`. The expression in the `if` statement in Line 9 determines whether the value of `guess` is the same as `num`, that is, if the user guessed the number correctly. If the value of `guess` is the same as `num`, the statement in Line 11 outputs the message:

You guessed the correct number.

The statement in Line 12 sets the variable `isGuessed` to `true`. The control then goes back to Line 3. Because `done` is `true`, `!isGuessed` is `false` and the `while` loop terminates. If the expression in Line 9 evaluates to `false`, then the `else` statement in Line 14 determines whether the value of `guess` is less than or greater than `num` and outputs the appropriate message.

Case 4: EOF-Controlled while Loops

If the data file is frequently altered (for example, if data is frequently added or deleted), it's best not to read the data with a sentinel value. Someone might accidentally erase the sentinel value or add data past the sentinel, especially if the programmer and the data entry person are different people. Also, it can be difficult at times to select a good sentinel value. In such situations, you can use an **end-of-file (EOF)-controlled while loop**.

Until now, we have used an input stream variable, such as `cin`, and the extraction operator, `>>`, to read and store data into variables. However, the input stream variable can also return a value after reading data, as follows:

1. If the program has reached the end of the input data, the input stream variable returns the logical value `false`.
2. If the program reads any faulty data (such as a `char` value into an `int` variable), the input stream enters the fail state. Once a stream enters the fail state, any further I/O operations using that stream are considered to be null operations; that is, they have no effect. Unfortunately, the computer does not halt the program or give any error messages. It just continues executing the program, silently ignoring each additional attempt to use that stream. In this case, the input stream variable returns the value `false`.
3. In cases other than (1) and (2), the input stream variable returns the logical value `true`.

You can use the value returned by the input stream variable to determine whether the program has reached the end of the input data. Because the input stream variable returns the logical value `true` or `false`, in a `while` loop, it can be considered a logical expression.

The following is an example of an EOF-controlled `while` loop:

```
cin >> variable;      //initialize the loop control variable

while (cin)           //test the loop control variable
{
    .
    .
    .
    cin >> variable; //update the loop control variable
    .
    .
    .
}
```

Notice that here, the variable `cin` acts as the loop control variable.

eof Function

In addition to checking the value of an input stream variable, such as `cin`, to determine whether the end of the file has been reached, C++ provides a function that you can use with an input stream variable to determine the end-of-file status. This function is called

`eof`. Like the I/O functions—such as `get`, `ignore`, and `peek`, discussed in Chapter 3—the function `eof` is a member of the data type `istream`.

The syntax to use the function `eof` is:

```
istreamVar.eof()
```

in which `istreamVar` is an input stream variable, such as `cin`.

Suppose you have the declaration:

```
ifstream infile;
```

Further suppose that you opened a file using the variable `infile`. Consider the expression:

```
infile.eof()
```

This is a logical (Boolean) expression. The value of this expression is `true` if the program has read past the end of the input file, `infile`; otherwise, the value of this expression is `false`.

This method of determining the end-of-file status (that is, using the function `eof`) works best if the input is text. The earlier method of determining the end-of-file status works best if the input consists of numeric data.

Suppose you have the declaration:

```
ifstream infile;
char ch;
```

```
infile.open("inputDat.dat");
```

The following `while` loop continues to execute as long as the program has not reached the end of the file.

```
infile.get(ch);

while (!infile.eof())
{
    cout << ch;
    infile.get(ch);
}
```

As long as the program has not reached the end of the input file, the expression:

```
infile.eof()
```

is `false` and so the expression:

```
!infile.eof()
```

in the `while` statement is `true`. When the program reads past the end of the input file, the expression:

```
infile.eof()
```

becomes **true**, so the expression:

```
!infile.eof()
```

in the **while** statement becomes **false** and the loop terminates.

NOTE

In the Windows console environment, the end-of-file marker is entered using **Ctrl+z** (hold the **Ctrl** key and press **z**). In the UNIX environment, the end-of-file marker is entered using **Ctrl+d** (hold the **Ctrl** key and press **d**).

EXAMPLE 5-7

The following code uses an EOF-controlled **while** loop to find the sum of a set of numbers:

```
int sum = 0;
int num;

cin >> num;

while (cin)
{
    sum = sum + num;    //Add the number to sum
    cin >> num;        //Get the next number
}

cout << "Sum = " << sum << endl;
```

EXAMPLE 5-8

Suppose we are given a file consisting of students' names and their test scores, a number between 0 and 100 (inclusive). Each line in the file consists of a student name followed by the test score. We want a program that outputs each student's name followed by the test score followed by the grade. The program also needs to output the average test score for the class. Consider the following program:

```
// This program reads data from a file consisting of students'
// names and their test scores. The program outputs each student's
// name followed by the test score followed by the grade. The
// program also outputs the average test score for all the students.

#include <iostream>                                //Line 1
#include <fstream>                                  //Line 2
#include <string>                                    //Line 3
#include <iomanip>                                    //Line 4

using namespace std;                               //Line 5
```

```

int main() //Line 6
{ //Line 7
    //Declare variables to manipulate data
    string firstName; //Line 8
    string lastName; //Line 9
    double testScore; //Line 10
    char grade = ' '; //Line 11
    double sum = 0; //Line 12
    int count = 0; //Line 13

    //Declare stream variables
    ifstream inFile; //Line 14
    ofstream outFile; //Line 15

    //Open input file
    inFile.open("Ch5_stData.txt"); //Line 16

    if (!inFile) //Line 17
    { //Line 18
        cout << "Cannot open input file. "
              << "Program terminates!" << endl; //Line 19
        return 1; //Line 20
    } //Line 21

    //Open output file
    outFile.open("Ch5_stData.out"); //Line 22

    outFile << fixed << showpoint << setprecision(2); //Line 23

    inFile >> firstName >> lastName; //read the name Line 24
    inFile >> testScore; //read the test score Line 25

    while (inFile) //Line 26
    { //Line 27
        sum = sum + testScore; //update sum Line 28
        count++; //increment count Line 29

        //determine the grade
        switch (static_cast<int> (testScore) / 10) //Line 30
        { //Line 31
            case 0: //Line 32
            case 1: //Line 33
            case 2: //Line 34
            case 3: //Line 35
            case 4: //Line 36
            case 5: //Line 37
                grade = 'F'; //Line 38
                break; //Line 39

            case 6: //Line 40
                grade = 'D'; //Line 41
                break; //Line 42

            case 7: //Line 43
                grade = 'C'; //Line 44
                break; //Line 45
        }
    }
}

```

```

        case 8:                                //Line 46
            grade = 'B';                        //Line 47
            break;                              //Line 48

        case 9:                                //Line 49
        case 10:                               //Line 50
            grade = 'A';                        //Line 51
            break;                              //Line 52

        default:                               //Line 53
            cout << "Invalid score." << endl;  //Line 54
    } //end switch                             //Line 55

    outFile << left << setw(12) << firstName
            << setw(12) << lastName
            << right << setw(4) << testScore
            << setw(2) << grade << endl;        //Line 56

    inFile >> firstName >> lastName; //read the name Line 57
    inFile >> testScore;             //read the test score Line 58
} //end while                        //Line 59

outFile << endl;                     //Line 60

if (count != 0)                      //Line 61
    outFile << "Class Average: " << sum / count
            << endl;                 //Line 62
else                                  //Line 63
    outFile << "No data." << endl;   //Line 64

inFile.close();                      //Line 65
outFile.close();                     //Line 66

return 0;                            //Line 67
}                                     //Line 68

```

Sample Run:**Input File:**

```

Steve Gill 89
Rita Johnson 91.5
Randy Brown 85.5
Seema Arora 76.5
Samir Mann 73
Samantha McCoy 88.5

```

Output File:

Steve	Gill	89.00 B
Rita	Johnson	91.50 A
Randy	Brown	85.50 B
Seema	Arora	76.50 C
Samir	Mann	73.00 C
Samantha	McCoy	88.50 B

Class Average: 84.00

The preceding program works as follows. The statements in Lines 8 to 13 declare and initialize variables needed by the program. The statement in Lines 14 and 15 declares `inFile` to be an `ifstream` variable and `outFile` to be an `ofstream` variable. The statement in Line 16 opens the input file using the variable `inFile`. If the input file does not exist, the statements in Lines 17 to 21 output an appropriate message and terminate the program. The statement in Line 22 opens the output file using the variable `outFile`. The statement in Line 23 sets the output of floating-point numbers to two decimal places in a fixed form with trailing zeros.

The statements in Lines 24 and 25 and the `while` loop in Line 26 read each student's first name, last name, and test score and then output the name followed by the test score followed by the grade. Specifically, the statement in Lines 24 and 57 reads the first and last name; the statement in Lines 25 and 58 reads the test score. The statement in Line 28 updates the value of `sum`. (After reading all the data, the value of `sum` stores the sum of all the test scores.) The statement in Line 29 updates the value of `count`. (The variable `count` stores the number of students in the class.) The `switch` statement from Lines 30 to 55 determines the grade from `testScore` and stores it in the variable `grade`. The statement in Line 56 outputs a student's first name, last name, test score, and grade.

The `if...else` statement in Lines 61 to 64 outputs the class average and the statements in Lines 65 and 66 close the files.

The Programming Example: Checking Account Balance, available on the Web site accompanying this book, further illustrates how to use an EOF-controlled `while` loop in a program.

More on Expressions in `while` Statements

In the examples of the previous sections, the expression in the `while` statement is quite simple. In other words, the `while` loop is controlled by a single variable. However, there are situations when the expression in the `while` statement may be more complex.

For example, the program in Example 5-6 uses a flag-controlled `while` loop to implement the Number Guessing Game. However, the program gives as many tries as the user needs to guess the number. Suppose you want to give the user no more than five tries to guess the number. If the user does not guess the number correctly within five tries, then the program outputs the random number generated by the program as well as a message that you have lost the game. In this case, you can write the `while` loop as follows (assume that `noOfGuesses` is an `int` variable initialized to 0):

```
while ((noOfGuesses < 5) && (!isGuessed))
{
    cout << "Enter an integer greater than or equal to 0 and "
          << "less than 100: ";
    cin >> guess;
    cout << endl;
```

```

noOfGuesses++;
if (guess == num)
{
    cout << "Winner!. You guessed the correct number."
        << endl;
    isGuessed = true;
}
else if (guess < num)
    cout << "Your guess is lower than the number.\n"
        << "Guess again!" << endl;
else
    cout << "Your guess is higher than the number.\n"
        << "Guess again!" << endl;
} //end while

```

You also need the following code to be included after the `while` loop in case the user cannot guess the correct number in five tries.

```

if (!isGuessed)
    cout << "You lose! The correct number is " << num << endl;

```

Programming Exercise 16 at the end of this chapter asks you to write a complete C++ program to implement the Number Guessing Game in which the user has, at most, five tries to guess the number.

As you can see from the preceding `while` loop, the expression in a `while` statement can be complex. The main objective of a `while` loop is to repeat certain statement(s) until certain conditions are met.

PROGRAMMING EXAMPLE: Fibonacci Number

So far, you have seen several examples of loops. Recall that in C++, `while` loops are used when a certain statement(s) must be executed repeatedly until certain conditions are met. Following is a C++ program that uses a `while` loop to find a **Fibonacci number**.

Consider the following sequence of numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34,

Given the first two numbers of the sequence (say, a_1 and a_2), the n th number a_n , $n \geq 3$, of this sequence is given by:

$$a_n = a_{n-1} + a_{n-2}$$

Thus:

$$a_3 = a_2 + a_1 = 1 + 1 = 2,$$

$$a_4 = a_3 + a_2 = 2 + 1 = 3,$$

and so on.

Such a sequence is called a **Fibonacci sequence**. In the preceding sequence, $a_2 = 1$ and $a_1 = 1$. However, given any first two numbers, using this process, you can determine the n th number, $a_n, n \geq 3$, of the sequence. The number determined this way is called the **n th Fibonacci number**. Suppose $a_2 = 6$ and $a_1 = 3$.

Then:

$$a_3 = a_2 + a_1 = 6 + 3 = 9; a_4 = a_3 + a_2 = 9 + 6 = 15$$

Next, we write a program that determines the n th Fibonacci number given the first two numbers.

Input The first two Fibonacci numbers and the desired Fibonacci number.

Output The n th Fibonacci number.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

To find, say, the tenth Fibonacci number of a sequence, you must first find a_9 and a_8 , which requires you to find a_7 and a_6 , and so on. Therefore, to find a_{10} , you must first find $a_3, a_4, a_5, \dots, a_9$. This discussion translates into the following algorithm:

1. Get the first two Fibonacci numbers.
2. Get the desired Fibonacci number. That is, get the position, n , of the Fibonacci number in the sequence.
3. Calculate the next Fibonacci number by adding the previous two elements of the Fibonacci sequence.
4. Repeat Step 3 until the n th Fibonacci number is found.
5. Output the n th Fibonacci number.

Note that the program assumes that the first number of the Fibonacci sequence is less than or equal to the second number of the Fibonacci sequence, and both numbers are nonnegative. Moreover, the program also assumes that the user enters a valid value for the position of the desired number in the Fibonacci sequence; that is, it is a positive integer. (See Programming Exercise 12 at the end of this chapter.)

Variables Because the last two numbers must be known in order to find the current Fibonacci number, you need the following variables: two variables—say, **previous1** and **previous2** to hold the previous two numbers of the Fibonacci sequence; and one variable—say, **current**—to hold the current Fibonacci number. The number of times that Step 2 of the algorithm repeats depends on the position of the Fibonacci number you are calculating. For example, if you want to calculate the tenth Fibonacci number, you must execute Step 3 eight times. (Remember—the user gives the first two numbers of the Fibonacci sequence.) Therefore, you need a variable to store the number of times Step 3 should execute. You also need a variable to track the number of times Step 3 has executed, the loop control variable. You therefore need five variables for the data manipulation:

```
int previous1; //variable to store the first Fibonacci number
int previous2; //variable to store the second Fibonacci number
```



```

int current;    //variable to store the current
                //Fibonacci number
int counter;    //loop control variable
int nthFibonacci; //variable to store the desired
                //Fibonacci number

```

To calculate the third Fibonacci number, add the values of `previous1` and `previous2` and store the result in `current`. To calculate the fourth Fibonacci number, add the value of the second Fibonacci number (that is, `previous2`) and the value of the third Fibonacci number (that is, `current`). Thus, when the fourth Fibonacci number is calculated, you no longer need the first Fibonacci number. Instead of declaring additional variables, which could be too many, after calculating a Fibonacci number to determine the next Fibonacci number, `current` becomes `previous2` and `previous2` becomes `previous1`. Therefore, you can again use the variable `current` to store the next Fibonacci number. This process is repeated until the desired Fibonacci number is calculated. Initially, `previous1` and `previous2` are the first two elements of the sequence, supplied by the user. From the preceding discussion, it follows that you need five variables.

MAIN ALGORITHM

1. Prompt the user for the first two numbers—that is, `previous1` and `previous2`.
2. Read (input) the first two numbers into `previous1` and `previous2`.
3. Output the first two Fibonacci numbers. (Echo input.)
4. Prompt the user for the position of the desired Fibonacci number.
5. Read the position of the desired Fibonacci number into `nthFibonacci`.
6.
 - a. `if (nthFibonacci == 1)`
the desired Fibonacci number is the first Fibonacci number.
Copy the value of `previous1` into `current`.
 - b. `else if (nthFibonacci == 2)`
the desired Fibonacci number is the second Fibonacci number.
Copy the value of `previous2` into `current`.
 - c. `else` calculate the desired Fibonacci number as follows:
Because you already know the first two Fibonacci numbers of the sequence, start by determining the third Fibonacci number.
 - c.1. Initialize `counter` to 3 to keep track of the calculated Fibonacci numbers.
 - c.2. Calculate the next Fibonacci number, as follows:
`current = previous2 + previous1;`
 - c.3. Assign the value of `previous2` to `previous1`.
 - c.4. Assign the value of `current` to `previous2`.
 - c.5. Increment `counter`.

Repeat Steps c.2 through c.5 until the Fibonacci number you want is calculated.

The following **while** loop executes Steps c.2 through c.5 and determines the *n*th Fibonacci number.

```
while (counter <= nthFibonacci)
{
    current = previous2 + previous1;
    previous1 = previous2;
    previous2 = current;
    counter++;
}
```

7. Output the `nthFibonacci` number, which is `current`.

COMPLETE PROGRAM LISTING

```

//*****
// Authors: D.S. Malik
//
// Program: nth Fibonacci number
// Given the first two numbers of a Fibonacci sequence, this
// program determines and outputs the desired number of the
// Fibonacci sequence.
//*****

#include <iostream>

using namespace std;

int main()
{
    //Declare variables
    int previous1;
    int previous2;
    int current;
    int counter;
    int nthFibonacci;

    cout << "Enter the first two Fibonacci "
         << "numbers: ";
    cin >> previous1 >> previous2;
    cout << endl;
    cout << "The first two Fibonacci numbers are "
         << previous1 << " and " << previous2
         << endl;
    cout << "Enter the position of the desired "
         << "Fibonacci number: ";
    cin >> nthFibonacci;
    cout << endl;

    if (nthFibonacci == 1)
        current = previous1;

```

```

else if (nthFibonacci == 2)                //Step 6.b
    current = previous2;
else                                        //Step 6.c
{
    counter = 3;                            //Step 6.c.1

    //Steps 6.c.2 - 6.c.5
    while (counter <= nthFibonacci)
    {
        current = previous2 + previous1;    //Step 6.c.2
        previous1 = previous2;              //Step 6.c.3
        previous2 = current;                //Step 6.c.4
        counter++;                           //Step 6.c.5
    } //end while
} //end else

cout << "The Fibonacci number at position "
    << nthFibonacci << " is " << current
    << endl;                                //Step 7

return 0;
} //end main

```

Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1:

```

Enter the first two Fibonacci numbers: 12 16
The first two Fibonacci numbers are 12 and 16
Enter the position of the desired Fibonacci number: 10
The Fibonacci number at position 10 is 796

```

Sample Run 2:

```

Enter the first two Fibonacci numbers: 1 1
The first two Fibonacci numbers are 1 and 1
Enter the position of the desired Fibonacci number: 15
The Fibonacci number at position 15 is 610

```

for Looping (Repetition) Structure

The **while** loop discussed in the previous section is general enough to implement most forms of repetitions. The C++ **for** looping structure discussed here is a specialized form of the **while** loop. Its primary purpose is to simplify the writing of counter-controlled loops. For this reason, the **for** loop is typically called a counted or indexed **for** loop.

The general form of the **for** statement is:

```
for (initial statement; loop condition; update statement)
    statement
```

The **initial statement**, **loop condition**, and **update statement** (called **for** loop control statements) enclosed within the parentheses control the body (**statement**) of the **for** statement. Figure 5-2 shows the flow of execution of a **for** loop.

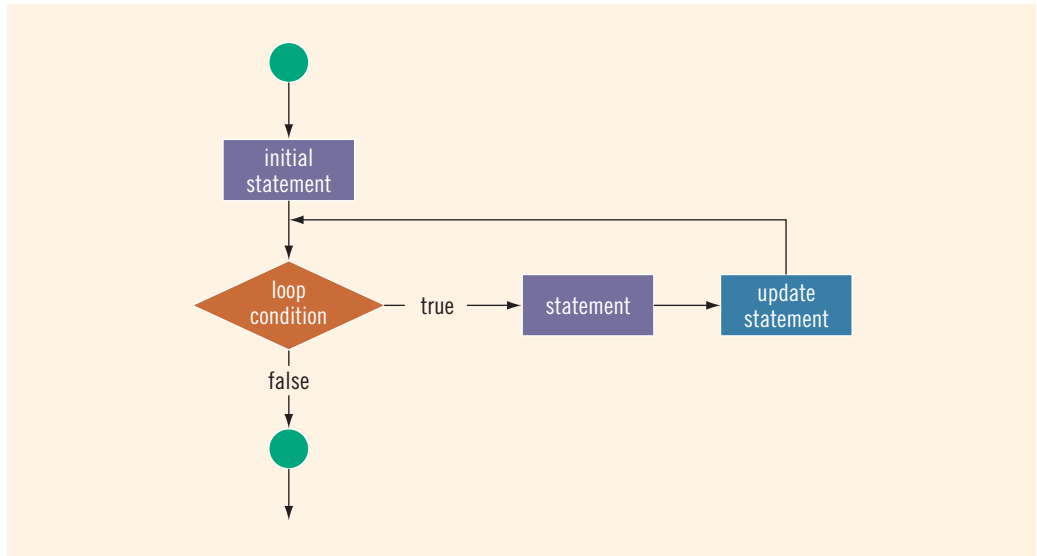


FIGURE 5-2 **for** loop

The **for** loop executes as follows:

1. The **initial statement** executes.
2. The **loop condition** is evaluated. If the **loop condition** evaluates to **true**:
 - i. Execute the **for** loop **statement**.
 - ii. Execute the **update statement** (the third expression in the parentheses).
3. Repeat Step 2 until the **loop condition** evaluates to **false**.

The **initial statement** usually initializes a variable (called the **for** loop **control**, or **for indexed, variable**).

In C++, **for** is a reserved word.

NOTE

As the name implies, the initial statement in the **for** loop is the first statement to execute; it executes only once.

EXAMPLE 5-9

The following **for** loop prints the first 10 nonnegative integers:

```
for (i = 0; i < 10; i++)
    cout << i << " ";
cout << endl;
```

The **initial statement**, `i = 0;`, initializes the **int** variable `i` to 0. Next, the loop condition, `i < 10`, is evaluated. Because `0 < 10` is **true**, the print statement executes and outputs 0. The **update statement**, `i++`, then executes, which sets the value of `i` to 1. Once again, the loop condition is evaluated, which is still **true**, and so on. When `i` becomes 10, the loop condition evaluates to **false**, the **for** loop terminates, and the statement following the **for** loop executes.

A **for** loop can have either a simple or compound statement.

The following examples further illustrate how a **for** loop executes.

EXAMPLE 5-10

1. The following **for** loop outputs **Hello!** and a star (on separate lines) five times:

```
for (i = 1; i <= 5; i++)
{
    cout << "Hello!" << endl;
    cout << "*" << endl;
}
```

2. Consider the following **for** loop:

```
for (i = 1; i <= 5; i++)
    cout << "Hello!" << endl;
    cout << "*" << endl;
```

This loop outputs **Hello!** five times and the star only once. Note that the **for** loop controls only the first output statement because the two output statements are not made into a compound statement. Therefore, the first output statement executes five times because the **for** loop body executes five times. After the **for** loop executes, the second output statement executes only once. The indentation, which is ignored by the compiler, is nevertheless misleading.

EXAMPLE 5-11

The following **for** loop executes five empty statements:

```
for (i = 0; i < 5; i++);      //Line 1
    cout << "*" << endl;    //Line 2
```

The semicolon at the end of the **for** statement (before the output statement, Line 1) terminates the **for** loop. The action of this **for** loop is empty, that is, null.

The preceding examples show that care is required in getting a **for** loop to perform the desired action.

The following are some comments on **for** loops:

- If the **loop condition** is initially **false**, the loop body does not execute.
- The **update expression**, when executed, changes the value of the loop control variable (initialized by the initial expression), which eventually sets the value of the **loop condition** to **false**. The **for** loop body executes indefinitely if the **loop condition** is always **true**.
- C++ allows you to use fractional values for loop control variables of the **double** type (or any real data type). Because different computers can give these loop control variables different results, you should avoid using such variables.
- A semicolon at the end of the **for** statement (just before the body of the loop) is a semantic error. In this case, the action of the **for** loop is empty.
- In the **for** statement, if the **loop condition** is omitted, it is assumed to be **true**.
- In a **for** statement, you can omit all three statements—**initial statement**, **loop condition**, and **update statement**. The following is a legal **for** loop:

```
for (;;)
    cout << "Hello" << endl;
```

This is an infinite **for** loop, continuously printing the word **Hello**.

Following are more examples of **for** loops.

EXAMPLE 5-12

You can count backward using a **for** loop if the **for** loop control expressions are set correctly.

For example, consider the following **for** loop:

```
for (i = 10; i >= 1; i--)
    cout << " " << i;
cout << endl;
```

The output is:

10 9 8 7 6 5 4 3 2 1

In this **for** loop, the variable **i** is initialized to 10. After each iteration of the loop, **i** is decremented by 1. The loop continues to execute as long as **i** \geq 1.

EXAMPLE 5-13

You can increment (or decrement) the loop control variable by any fixed number. In the following **for** loop, the variable is initialized to 1; at the end of the **for** loop, **i** is incremented by 2. This **for** loop outputs the first 10 positive odd integers.

```
for (i = 1; i <= 20; i = i + 2)
    cout << " " << i;
cout << endl;
```

EXAMPLE 5-14

Suppose that **i** is an **int** variable.

1. Consider the following **for** loop:

```
for (i = 10; i <= 9; i++)
    cout << i << " ";
cout << endl;
```

In this **for** loop, the initial statement sets **i** to 10. Because initially the loop condition (**i** \leq 9) is **false**, nothing happens.

2. Consider the following **for** loop:

```
for (i = 9; i >= 10; i--)
    cout << i << " ";
cout << endl;
```

In this **for** loop, the initial statement sets **i** to 9. Because initially the loop condition (**i** \geq 10) is **false**, nothing happens.

3. Consider the following **for** loop:

```
for (i = 10; i <= 10; i++)           //Line 1
    cout << i << " ";               //Line 2
cout << endl;                        //Line 3
```

In this **for** loop, the initial statement sets **i** to 10. The loop condition (**i** \leq 10) evaluates to **true**, so the output statement in Line 2 executes, which outputs 10.

Next, the update statement increments the value of `i` by 1, so the value of `i` becomes 11. Now the loop condition evaluates to `false` and the `for` loop terminates. Note that the output statement in Line 2 executes only once.

4. Consider the following `for` loop:

```
for (i = 1; i <= 10; i++); //Line 1
    cout << i << " ";      //Line 2
cout << endl;              //Line 3
```

This `for` loop has no effect on the output statement in Line 2. The semicolon at the end of the `for` statement terminates the `for` loop; the action of the `for` loop is thus empty. The output statement is all by itself and executes only once.

5. Consider the following `for` loop:

```
for (i = 1; ; i++)
    cout << i << " ";
cout << endl;
```

In this `for` loop, because the loop condition is omitted from the `for` statement, the loop condition is always `true`. This is an infinite loop.

EXAMPLE 5-15

In this example, a `for` loop reads five numbers and finds their sum and average. Consider the following program code, in which `i`, `newNum`, `sum`, and `average` are `int` variables.

```
sum = 0;

for (i = 1; i <= 5; i++)
{
    cin >> newNum;
    sum = sum + newNum;
}

average = sum / 5;
cout << "The sum is " << sum << endl;
cout << "The average is " << average << endl;
```

In the preceding `for` loop, after reading a `newNum`, this value is added to the previously calculated (partial) `sum` of all the numbers read before the current number. The variable `sum` is initialized to 0 before the `for` loop. Thus, after the program reads the first number and adds it to the value of `sum`, the variable `sum` holds the correct `sum` of the first number.

NOTE

The syntax of the **for** loop, which is:

```
for (initial expression; logical expression; update expression)
    statement
```

is functionally equivalent to the following **while** statement:

```
initial expression
while (expression)
{
    statement
    update expression
}
```

For example, the following **for** and **while** loops are equivalent:

<pre>for (int i = 0; i < 10; i++) cout << i << " "; cout << endl;</pre>	<pre>int i = 0; while (i < 10) { cout << i << " "; i++; } cout << endl;</pre>
--	--

If the number of iterations of a loop is known or can be determined in advance, typically programmers use a **for** loop.

EXAMPLE 5-16 (FIBONACCI NUMBER PROGRAM: REVISITED)

The Programming Example: Fibonacci Number given in the previous section uses a **while** loop to determine the desired Fibonacci number. You can replace the **while** loop with an equivalent **for** loop as follows:

```
for (counter = 3; counter <= nthFibonacci; counter++)
{
    current = previous2 + previous1;
    previous1 = previous2;
    previous2 = current;
    counter++;
} //end for
```

The complete program listing of the program that uses a **for** loop to determine the desired Fibonacci number is given at the Web site accompanying this book. The program is named `Ch5_FibonacciNumberUsingAForLoop.cpp`.

In the following C++ program, we recommend that you walk through each step.

EXAMPLE 5-17

The following C++ program finds the sum of the first *n* positive integers.

//Program to determine the sum of the first n positive integers.

```
#include <iostream>

using namespace std;

int main()
{
    int counter;    //loop control variable
    int sum;        //variable to store the sum of numbers
    int n;          //variable to store the number of
                  //first positive integers to be added

    cout << "Line 1: Enter the number of positive "
         << "integers to be added: ";           //Line 1
    cin >> n;                                   //Line 2
    sum = 0;                                    //Line 3
    cout << endl;                               //Line 4

    for (counter = 1; counter <= n; counter++) //Line 5
        sum = sum + counter;                   //Line 6

    cout << "Line 7: The sum of the first " << n
         << " positive integers is " << sum
         << endl;                               //Line 7

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

Line 1: Enter the number of positive integers to be added: **100**

Line 7: The sum of the first 100 positive integers is 5050

The statement in Line 1 prompts the user to enter the number of positive integers to be added. The statement in Line 2 stores the number entered by the user in *n*, and the statement in Line 3 initializes *sum* to 0. The **for** loop in Line 5 executes *n* times. In the **for** loop, *counter* is initialized to 1 and is incremented by 1 after each iteration of the loop. Therefore, *counter* ranges from 1 to *n*. Each time through the loop, the value of *counter* is added to *sum*. The variable *sum* was initialized to 0, *counter* ranges from 1 to *n*, and the current value of *counter* is added to the value of *sum*. Therefore, after the **for** loop executes, *sum* contains the sum of the first *n* values, which in the sample run is 100 positive integers.

Recall that putting one control structure statement inside another is called **nesting**. The following programming example demonstrates a simple instance of nesting. It also nicely demonstrates counting.

PROGRAMMING EXAMPLE: Classifying Numbers

This program reads a given set of integers and then prints the number of odd and even integers. It also outputs the number of zeros.

The program reads 20 integers, but you can easily modify it to read any set of numbers. In fact, you can modify the program so that it first prompts the user to specify how many integers are to be read.

Input 20 integers—positive, negative, or zeros.

Output The number of zeros, even numbers, and odd numbers.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

After reading a number, you need to check whether it is even or odd. Suppose the value is stored in **number**. Divide **number** by 2 and check the remainder. If the remainder is 0, **number** is even. Increment the even count and then check whether **number** is 0. If it is, increment the zero count. If the remainder is not 0, increment the odd count.

The program uses a **switch** statement to decide whether **number** is odd or even. Suppose that **number** is odd. Dividing by 2 gives the remainder 1 if **number** is positive and the remainder -1 if it is negative. If **number** is even, dividing by 2 gives the remainder 0 whether **number** is positive or negative. You can use the mod operator, **%**, to find the remainder. For example:

$6 \% 2 = 0$; $-4 \% 2 = 0$; $-7 \% 2 = -1$; $15 \% 2 = 1$

Repeat the preceding process of analyzing a number for each number in the list.

This discussion translates into the following algorithm:

1. For each number in the list:
 - a. Get the number.
 - b. Analyze the number.
 - c. Increment the appropriate count.
2. Print the results.

Variables Because you want to count the number of zeros, even numbers, and odd numbers, you need three variables of type **int**—say, **zeros**, **evens**, and **odds**—to track the counts. You also need a variable—say, **number**—to read and store the number to be analyzed and another variable—say, **counter**—to count the numbers analyzed. Therefore, you need the following variables in the program:

```

int counter;    //loop control variable
int number;    //variable to store the number read
int zeros;     //variable to store the zero count
int evens;     //variable to store the even count
int odds;      //variable to store the odd count

```

Clearly, you must initialize the variables `zeros`, `evens`, and `odds` to zero. You can initialize these variables when you declare them.

MAIN ALGORITHM

1. Initialize the variables.
2. Prompt the user to enter 20 numbers.
3. For each number in the list:
 - a. Read the number.
 - b. Output the number (echo input).
 - c. If the number is even:


```

          {
              i. Increment the even count.
              ii. If the number is zero, increment the zero count.
          }
          otherwise
              Increment the odd count.
          
```
4. Print the results.

Before writing the C++ program, let us describe Steps 1–4 in greater detail. It will be much easier for you to then write the instructions in C++.

1. Initialize the variables. You can initialize the variables `zeros`, `evens`, and `odds` when you declare them.
2. Use an output statement to prompt the user to enter 20 numbers.
3. For Step 3, you can use a `for` loop to process and analyze the 20 numbers. In pseudocode, this step is written as follows:

```

for (counter = 1; counter <= 20; counter++)
{
    read the number;
    output number;

    switch (number % 2)    // check the remainder
    {
        case 0:
            increment even count;
            if (number == 0)
                increment zero count;
            break;
    }
}

```

```

        case 1:
        case -1:
            increment odd count;
        } //end switch
    } //end for

```

4. Print the result. Output the value of the variables `zeros`, `evens`, and `odds`.

COMPLETE PROGRAM LISTING

```

//*****
// Author: D.S. Malik
//
// Program: Counts zeros, odds, and evens
// This program counts the number of odd and even numbers.
// The program also counts the number of zeros.
//*****

#include <iostream>
#include <iomanip>

using namespace std;

const int N = 20;

int main()
{
    //Declare variables
    int counter;    //loop control variable
    int number;     //variable to store the new number
    int zeros = 0;    //Step 1
    int odds = 0;     //Step 1
    int evens = 0;    //Step 1

    cout << "Please enter " << N << " integers, "
         << "positive, negative, or zeros."
         << endl;    //Step 2

    cout << "The numbers you entered are:" << endl;

    for (counter = 1; counter <= N; counter++)    //Step 3
    {
        cin >> number;    //Step 3a
        cout << number << " ";    //Step 3b

        //Step 3c
        switch (number % 2)

```

```

    {
        case 0:
            evens++;
            if (number == 0)
                zeros++;
            break;
        case 1:
        case -1:
            odds++;
        } //end switch
    } //end for loop

    cout << endl;

    //Step 4
    cout << "There are " << evens << " evens, "
        << "which includes " << zeros << " zeros."
        << endl;
    cout << "The number of odd numbers is: " << odds
        << endl;

    return 0;
}

```

Sample Run: In this sample run, the user input is shaded.

Please enter 20 integers, positive, negative, or zeros.

The numbers you entered are:

0 0 -2 -3 -5 6 7 8 0 3 0 -23 -8 0 2 9 0 12 67 54

0 0 -2 -3 -5 6 7 8 0 3 0 -23 -8 0 2 9 0 12 67 54

There are 13 evens, which includes 6 zeros.

The number of odd numbers is: 7

We recommend that you do a walk-through of this program using the above sample input.

do...while Looping (Repetition) Structure

This section describes the third type of looping or repetition structure, called a **do...while** loop. The general form of a **do...while** statement is as follows:

```

do
    statement
while (expression);

```

Of course, **statement** can be either a simple or compound statement. If it is a compound statement, enclose it between braces. Figure 5-3 shows the flow of execution of a **do...while** loop.

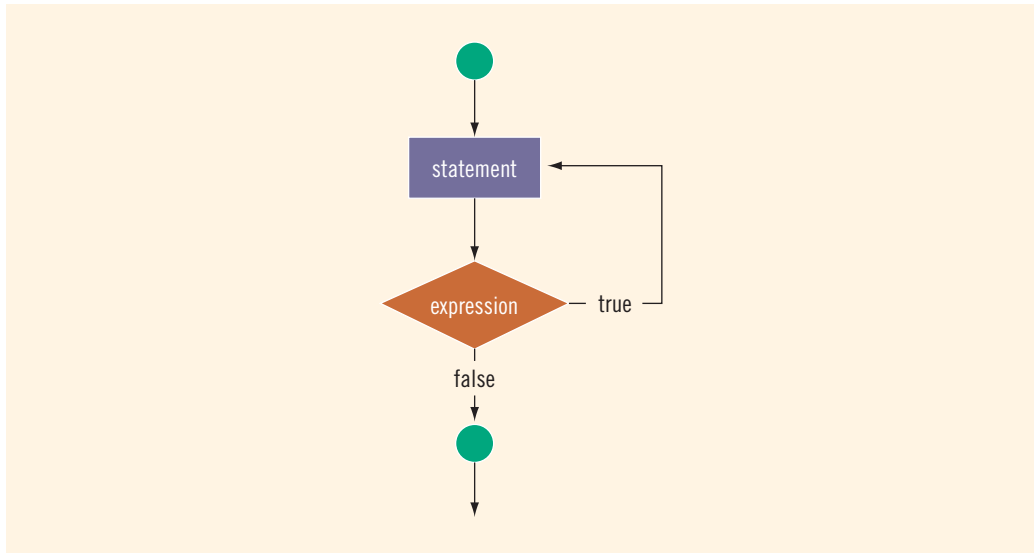


FIGURE 5-3 do...while loop

In C++, **do** is a reserved word.

The **statement** executes first, and then the **expression** is evaluated. If the **expression** evaluates to **true**, the **statement** executes again. As long as the **expression** in a **do...while** statement is **true**, the **statement** executes. To avoid an infinite loop, you must, once again, make sure that the loop body contains a statement that ultimately makes the **expression** **false** and assures that it exits properly.

EXAMPLE 5-18

```

i = 0;

do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 20);
  
```

The output of this code is:

0 5 10 15 20

After 20 is output, the statement:

```
i = i + 5;
```

changes the value of **i** to 25 and so **i <= 20** becomes **false**, which halts the loop.

In a **while** and **for** loop, the loop condition is evaluated before executing the body of the loop. Therefore, **while** and **for** loops are called **pretest loops**. On the other hand, the loop condition in a **do...while** loop is evaluated after executing the body of the loop. Therefore, **do...while** loops are called **posttest loops**.

Because the **while** and **for** loops both have entry conditions, these loops may never activate. The **do...while** loop, on the other hand, has an exit condition and therefore always executes the statement at least once.

EXAMPLE 5-19

Consider the following two loops:

```
a. i = 11;
   while (i <= 10)
   {
       cout << i << " ";
       i = i + 5;
   }
   cout << endl;

b. i = 11;
   do
   {
       cout << i << " ";
       i = i + 5;
   }
   while (i <= 10);

   cout << endl;
```

In (a), the **while** loop produces nothing. In (b), the **do...while** loop outputs the number 11 and also changes the value of **i** to 16.

A **do...while** loop can be used for input validation. Suppose that a program prompts a user to enter a test score, which must be greater than or equal to 0 and less than or equal to 50. If the user enters a score less than 0 or greater than 50, the user should be prompted to re-enter the score. The following **do...while** loop can be used to accomplish this objective:

```
int score;

do
{
    cout << "Enter a score between 0 and 50: ";
    cin >> score;
    cout << endl;
}
while (score < 0 || score > 50);
```


EXAMPLE 5-20**Divisibility Test by 3 and 9**

Suppose that m and n are integers and m is nonzero. Then m is called a **divisor** of n if $n = mt$ for some integer t ; that is, when m divides n , the remainder is 0.

Let $n = a_k a_{k-1} a_{k-2} \dots a_1 a_0$ be an integer. Let $s = a_k + a_{k-1} + a_{k-2} + \dots + a_1 + a_0$ be the sum of the digits of n . It is known that n is divisible by 3 and 9 if s is divisible by 3 and 9. In other words, an integer is divisible by 3 and 9 if and only if the sum of its digits is divisible by 3 and 9.

For example, suppose $n = 27193257$. Then $s = 2 + 7 + 1 + 9 + 3 + 2 + 5 + 7 = 36$. Because 36 is divisible by both 3 and 9, it follows that 27193257 is divisible by both 3 and 9.

Next, we write a program that determines whether a positive integer is divisible by 3 and 9 by first finding the sum of its digits and then checking whether the sum is divisible by 3 and 9.

To find the sum of the digits of a positive integer, we need to extract each digit of the number. Consider the number 951372. Note that $951372 \% 10 = 2$, which is the last digit of 951372. Also note that $951372 / 10 = 95137$; that is, when the number is divided by 10, it removes the last digit. Next, we repeat this process on the number 95137. Of course, we need to add the extracted digits.

Suppose that `sum` and `num` are `int` variables and the positive integer is stored in `num`. We thus have the following algorithm to find the sum of the digits:

```
sum = 0;

do
{
    sum = sum + num % 10; //extract the last digit
                        //and add it to sum
    num = num / 10;      //remove the last digit
}
while (num > 0);
```

Using this algorithm, we can write the following program that uses a `do...while` loop to implement the preceding divisibility test algorithm.

```
//Program: Divisibility test by 3 and 9

#include <iostream>

using namespace std;

int main()
{
    int num, temp, sum;

    cout << "Enter a positive integer: ";
    cin >> num;
```

```

    cout << endl;

    temp = num;

    sum = 0;

    do
    {
        sum = sum + num % 10; //extract the last digit
                             //and add it to sum
        num = num / 10;      //remove the last digit
    }
    while (num > 0);

    cout << "The sum of the digits = " << sum << endl;

    if (sum % 3 == 0)
        cout << temp << " is divisible by 3" << endl;
    else
        cout << temp << " is not divisible by 3" << endl;

    if (sum % 9 == 0)
        cout << temp << " is divisible by 9" << endl;
    else
        cout << temp << " is not divisible by 9" << endl;
}

```

Sample Run: In these sample runs, the user input is shaded.

Sample Run 1:

Enter a positive integer: 27193257

```

The sum of the digits = 36
27193257 is divisible by 3
27193257 is divisible by 9

```

Sample Run 2:

Enter a positive integer: 609321

```

The sum of the digits = 21
609321 is divisible by 3
609321 is not divisible by 9

```

Sample Run 3:

Enter a positive integer: 161905102

```

The sum of the digits = 25
161905102 is not divisible by 3
161905102 is not divisible by 9

```

Choosing the Right Looping Structure

All three loops have their place in C++. If you know, or the program can determine in advance, the number of repetitions needed, the **for** loop is the correct choice. If you do not know, and the program cannot determine in advance the number of repetitions needed, and it could be zero, the **while** loop is the right choice. If you do not know, and the program cannot determine in advance the number of repetitions needed, and it is at least one, the **do...while** loop is the right choice.

break and continue Statements

The **break** statement, when executed in a **switch** structure, provides an immediate exit from the **switch** structure. Similarly, you can use the **break** statement in **while**, **for**, and **do...while** loops. When the **break** statement executes in a repetition structure, it immediately exits from the structure. The **break** statement is typically used for two purposes:

- To exit early from a loop.
- To skip the remainder of the **switch** structure.

After the **break** statement executes, the program continues to execute with the first statement after the structure. The use of a **break** statement in a loop can eliminate the use of certain (flag) variables. The following C++ code segment helps illustrate this idea. (Assume that all variables are properly declared.)

```
sum = 0;
isNegative = false;

cin >> num;

while (cin && !isNegative)
{
    if (num < 0)    //if num is negative, terminate the loop
                   //after this iteration
    {
        cout << "Negative number found in the data." << endl;
        isNegative = true;
    }
    else
    {
        sum = sum + num;
        cin >> num;
    }
}
```

This **while** loop is supposed to find the sum of a set of positive numbers. If the data set contains a negative number, the loop terminates with an appropriate error message. This **while** loop uses the flag variable **isNegative** to accomplish the desired result. The variable **isNegative** is initialized to **false** before the **while** loop. Before adding **num**

to `sum`, check whether `num` is negative. If `num` is negative, an error message appears on the screen and `isNegative` is set to `true`. In the next iteration, when the expression in the `while` statement is evaluated, it evaluates to `false` because `!isNegative` is `false`. (Note that because `isNegative` is `true`, `!isNegative` is `false`.)

The following `while` loop is written without using the variable `isNegative`:

```
sum = 0;
cin >> num;

while (cin)
{
    if (num < 0)    //if num is negative, terminate the loop
    {
        cout << "Negative number found in the data." << endl;
        break;
    }

    sum = sum + num;
    cin >> num;
}
```

In this form of the `while` loop, when a negative number is found, the expression in the `if` statement evaluates to `true`; after printing an appropriate message, the `break` statement terminates the loop. (After executing the `break` statement in a loop, the remaining statements in the loop are discarded.)

NOTE

The `break` statement is an effective way to avoid extra variables to control a loop and produce an elegant code. However, `break` statements must be used very sparingly within a loop. An excessive use of these statements in a loop will produce spaghetti-code (loops with many exit conditions) that can be very hard to understand and manage. You should be extra careful in using `break` statements and ensure that the use of the `break` statements makes the code more readable and not less readable. If you're not sure, don't use `break` statements.

The `continue` statement is used in `while`, `for`, and `do...while` structures. When the `continue` statement is executed in a loop, it skips the remaining statements in the loop and proceeds with the next iteration of the loop. In a `while` and `do...while` structure, the `expression` (that is, the loop-continue test) is evaluated immediately after the `continue` statement. In a `for` structure, the `update statement` is executed after the `continue` statement, and then the `loop condition` (that is, the loop-continue test) executes.

If the previous program segment encounters a negative number, the `while` loop terminates. If you want to discard the negative number and read the next number rather than terminate the loop, replace the `break` statement with the `continue` statement, as shown in the following example:

```
sum = 0;
cin >> num;
```

```

while (cin)
{
    if (num < 0)
    {
        cout << "Negative number found in the data." << endl;
        cin >> num;
        continue;
    }

    sum = sum + num;
    cin >> num;
}

```

It was stated earlier that all three loops have their place in C++ and that one loop can often replace another. The execution of a `continue` statement, however, is where the `while` and `do...while` structures differ from the `for` structure. When the `continue` statement is executed in a `while` or a `do...while` loop, the update statement may not execute. In a `for` structure, the update statement *always* executes.

5

Nested Control Structures

In this section, we give examples that illustrate how to use nested loops to achieve useful results and process data.

EXAMPLE 5-21

Suppose you want to create the following pattern:

```

*
**
***
****
*****

```

Clearly, you want to print five lines of stars. In the first line, you want to print one star, in the second line, two stars, and so on. Because five lines will be printed, start with the following `for` statement:

```
for (i = 1; i <= 5; i++)
```

The value of `i` in the first iteration is 1, in the second iteration it is 2, and so on. You can use the value of `i` as the limiting condition in another `for` loop nested within this loop to control the number of stars in a line. A little more thought produces the following code:

```

for (i = 1; i <= 5; i++)           //Line 1
{                                   //Line 2
    for (j = 1; j <= i; j++)       //Line 3
        cout << "*";              //Line 4
    cout << endl;                  //Line 5
}                                   //Line 6

```

A walk-through of this code shows that the **for** loop in Line 1 starts with **i** = 1. When **i** is 1, the inner **for** loop in Line 3 outputs one star and the insertion point moves to the next line. Then **i** becomes 2, the inner **for** loop outputs two stars, and the output statement in Line 5 moves the insertion point to the next line, and so on. This process continues until **i** becomes 6 and the loop stops.

What pattern does this code produce if you replace the `for` statement in Line 1 with the following?

```
for (i = 5; i >= 1; i--)
```

EXAMPLE 5-22

Suppose you want to create the following multiplication table:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50

The multiplication table has five lines. Therefore, as in Example 5-21, we use a **for** statement to output these lines as follows:

```
for (i = 1; i <= 5; i++)
    //output a line of numbers
```

In the first line, we want to print the multiplication table of one, in the second line we want to print the multiplication table of 2, and so on. Notice that the first line starts with 1 and when this line is printed, `i` is 1. Similarly, the second line starts with 2 and when this line is printed, the value of `i` is 2, and so on. If `i` is 1, `i * 1` is 1; if `i` is 2, `i * 2` is 2; and so on. Therefore, to print a line of numbers, we can use the value of `i` as the starting number and 10 as the limiting value. That is, consider the following `for` loop:

```
for (j = 1; j <= 10; j++)
    cout << setw(3) << i * j;
```

Let us take a look at this **for** loop. Suppose **i** is 1. Then we are printing the first line of the multiplication table. Also, **j** goes from 1 to 10 and so this **for** loop outputs the numbers 1 through 10, which is the first line of the multiplication table. Similarly, if **i** is 2, we are printing the second line of the multiplication table. Also, **j** goes from 1 to 10, and so this **for** loop outputs the second line of the multiplication table, and so on.

A little more thought produces the following nested loops to output the desired grid:

```
for (i = 1; i <= 5; i++) //Line 1
{ //Line 2
    for (j = 1; j <= 10; j++) //Line 3
        cout << setw(3) << i * j; //Line 4
    cout << endl; //Line 5
} //Line 6
```

EXAMPLE 5-23

Consider the following data:

```
65 78 65 89 25 98 -999
87 34 89 99 26 78 64 34 -999
23 99 98 97 26 78 100 63 87 23 -999
62 35 78 99 12 93 19 -999
```

The number `-999` at the end of each line acts as a sentinel and therefore is not part of the data. Our objective is to find the sum of the numbers in each line and output the sum. Moreover, assume that this data is to be read from a file, say, `Exp_5_23.txt`. We assume that the input file has been opened using the input file stream variable `infile`.

This particular data set has four lines of input. So we can use a `for` loop or a counter-controlled `while` loop to process each line of data. Let us use a `while` loop to process these four lines. It follows that the `while` loop takes the following form:

```
counter = 0;                //Line 1
while (counter < 4)          //Line 2
{                            //Line 3
    //process the line      //Line 4

    //output the sum
    counter++;
}
```

Let us now concentrate on processing a line. Each line has a varying number of data items. For example, the first line has six numbers, the second line has eight numbers, and so on. Because each line ends with `-999`, we can use a sentinel-controlled `while` loop to find the sum of the numbers in each line. (Remember how a sentinel-controlled loop works.) Consider the following `while` loop:

```
sum = 0;                    //Line 4
infile >> num;              //Line 5
while (num != -999)         //Line 6
{                            //Line 7
    sum = sum + num;        //Line 8
    infile >> num;          //Line 9
}                            //Line 10
```

The statement in Line 4 initializes `sum` to 0, and the statement in Line 5 reads and stores the first number of the line into `num`. The Boolean expression `num != -999` in Line 6 checks whether the number is `-999`. If `num` is not `-999`, the statements in Lines 8 and 9 execute. The statement in Line 8 updates the value of `sum`; the statement in Line 9 reads and stores the next number into `num`. The loop continues to execute as long as `num` is not `-999`.

It now follows that the nested loop to process the data is as follows. (Assume that all variables are properly declared.)

```

counter = 0;                                //Line 1
while (counter < 4)                          //Line 2
{
    sum = 0;                                //Line 3
    infile >> num;                           //Line 4
    while (num != -999)                      //Line 5
    {
        sum = sum + num;                    //Line 6
        infile >> num;                      //Line 7
    }                                       //Line 8

    cout << "Line " << counter + 1
        << ": Sum = " << sum << endl;      //Line 9
    counter++;                              //Line 10
}                                           //Line 11

```

EXAMPLE 5-24

Suppose that we want to process data similar to the data in Example 5-23, but the input file is of an unspecified length. That is, each line contains the same data as the data in each line in Example 5-23, but we do not know the number of input lines.

Because we do not know the number of input lines, we must use an EOF-controlled **while** loop to process the data. In this case, the required code is as follows. (Assume that all variables are properly declared and the input file has been opened using the input file stream variable **infile**.)

```

counter = 0;                                //Line 1
infile >> num;                              //Line 2
while (infile)                              //Line 3
{
    sum = 0;                                //Line 4
    while (num != -999)                    //Line 5
    {
        sum = sum + num;                  //Line 6
        infile >> num;                    //Line 7
    }                                     //Line 8

    cout << "Line " << counter + 1
        << ": Sum = " << sum << endl;      //Line 9
    counter++;                            //Line 10
    infile >> num;                        //Line 11
}                                           //Line 12

```

Notice that we have again used the variable **counter**. The only reason to do so is because we want to print the line number with the sum of each line.

EXAMPLE 5-25

Consider the following data:

```
101
John Smith
65 78 65 89 25 98 -999
102
Peter Gupta
87 34 89 99 26 78 64 34 -999
103
Buddy Friend
23 99 98 97 26 78 100 63 87 23 -999
104
Doctor Miller
62 35 78 99 12 93 19 -999
...
```

The number `-999` at the end of a line acts as a sentinel and therefore is not part of the data.

Assume that this is the data of certain candidates seeking the student council's presidential seat.

For each candidate, the data is in the following form:

```
ID
Name
Votes
```

The objective is to find the total number of votes received by the candidate. We assume that the data is input from the file `Exp_5_25.txt` of unknown size. We also assume that the input file has been opened using the input file stream variable `infile`.

Because the input file is of an unspecified length, we use an EOF-controlled `while` loop. For each candidate, the first data item is the `ID` of type `int` on a line by itself; the second data item is the name, which may consist of more than one word; and the third line contains the votes received from the various departments.

To read the `ID`, we use the extraction operator `>>`; to read the name, we use the stream function `getline`. Notice that after reading the `ID`, the reading marker is after the `ID` and the character after the `ID` is the newline character. Therefore, after reading the `ID`, the reading marker is after the `ID` and before the newline character (of the line containing the `ID`).

The function `getline` reads until the end of the line. Therefore, if we read the name immediately after reading the `ID`, then what is stored in the variable name is the newline character (after the `ID`). It follows that to read the name, we must read and discard the newline character after the `ID`, which we can accomplish using the stream function `get`. Therefore, the statements to read the `ID` and name are as follows:

```
infile >> ID;           //read the ID
infile.get(ch);         //read the newline character after the ID
getline(infile, name);   //read the name
```

(Assume that `ch` is a variable of type `char`.) The general loop to process the data is:

```
infile >> ID;                                //Line 1
while (infile)                                //Line 2
{                                              //Line 3
    infile.get(ch);                            //Line 4
    getline(infile, name);                     //Line 5

    //process the numbers in each line        //Line 6
    //output the name and total votes
    infile >> ID;    //begin processing the next line
}
```

The code to read and sum up the voting data is:

```
sum = 0;                                     //Line 6
infile >> num;                               //Line 7; read the first number
while (num != -999)                           //Line 8
{                                              //Line 9
    sum = sum + num;                           //Line 10; update sum
    infile >> num;                             //Line 11; read the next number
}                                              //Line 12
```

We can now write the following nested loop to process data as follows:

```
infile >> ID;                                //Line 1
while (infile)                                //Line 2
{                                              //Line 3
    infile.get(ch);                            //Line 4
    getline(infile, name);                     //Line 5
    sum = 0;                                   //Line 6
    infile >> num;                             //Line 7; read the first number
    while (num != -999)                       //Line 8
    {                                          //Line 9
        sum = sum + num;                       //Line 10; update sum
        infile >> num;                         //Line 11; read the next number
    }

    cout << "Name: " << name
         << ", Votes: " << sum
         << endl;                             //Line 12

    infile >> ID;    //Line 13; begin processing the next line
}
```

Avoiding Bugs by Avoiding Patches

Debugging sections in the previous chapters illustrated how to debug syntax and logical errors, and how to avoid partially understood concepts. In this section, we illustrate how to avoid a software patch to fix a code. A software patch is a piece of code written on top of an existing piece of code and intended to fix a bug in the original code.

Suppose that the following data is in the file `Ch5_LoopWithBugsData.txt`.

```
87 78 83 94
23 89 92 70
92 78 34 56
```

The objective is to find the sum of the numbers in each line. For each line, output the numbers together with their sum. Let us consider the following program:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream infile;

    int i;
    int j;
    int sum;
    int num;

    infile.open("Ch5_LoopWithBugsData.txt");

    for (i = 1; i <= 4; i++)
    {
        sum = 0;

        for (j = 1; j <= 4; j++)
        {
            infile >> num;
            cout << num << " ";
            sum = sum + num;
        }

        cout << "sum = " << sum << endl;
    }

    return 0;
}
```

Sample Run:

```
87 78 83 94 sum = 342
23 89 92 70 sum = 274
92 78 34 56 sum = 260
56 56 56 56 sum = 224
```

The sample run shows that there is a bug in the program because the file contains three lines of input and the output contains four lines. Also, the number `56` in the last line repeats four times. Clearly, there is a bug in the program and we must fix the code. Some programmers, especially some beginners, address the symptom of the problem by adding a software patch. In this case, the output should contain only three lines of output.

A beginning programmer might fix the code by adding a software patch as shown in the following modified program.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream infile;

    int i;
    int j;
    int sum;
    int num;

    infile.open("Ch5_LoopWithBugsData.txt");

    for (i = 1; i <= 4; i++)
    {
        sum = 0;

        if (i != 4)
        {
            for (j = 1; j <= 4; j++)
            {
                infile >> num;
                cout << num << " ";
                sum = sum + num;
            }

            cout << "sum = " << sum << endl;
        }
    }

    return 0;
}
```

Sample Run:

```
87 78 83 94 sum = 342
23 89 92 70 sum = 274
92 78 34 56 sum = 260
```

Clearly, the program is working correctly now.

As we can see, the programmer merely observed the symptom and addressed the problem by adding a software patch. However, if you look at the code, not only does the program execute extra statements, it is also an example of a partially understood concept. It appears that the programmer does not have a good grasp of why the earlier program produced four lines rather than three. Adding a patch eliminated the symptom, but it is a poor programming practice. The programmer must resolve why the program produced four lines. Looking at the

program closely, we can see that the four lines are produced because the outer loop executes four times. The values assigned to loop control variable `i` are 1, 2, 3, and 4. This is an example of the classic “off-by-one” problem. (In an “off-by-one problem,” either the loop executes one too many or one too few times.) We can eliminate this problem by correctly setting the values of the loop control variable. For example, we can rewrite the loops as follows:

```
for (i = 1; i <= 3; i++)
{
    sum = 0;

    for (j = 1; j <= 4; j++)
    {
        infile >> num;
        cout << num << " ";
        sum = sum + num;
    }

    cout << "sum = " << sum << endl;
}
```

This code fixes the original problem without using a software patch. It also represents good programming practice. The complete modified program is available at the Web site accompanying this book and is named `Ch5_LoopWithBugsCorrectedProgram.cpp`.

Debugging Loops

As we have seen in the earlier debugging sections, no matter how careful a program is designed and coded, errors are likely to occur. If there are syntax errors, the compiler will identify them. However, if there are logical errors, we must carefully look at the code or even maybe at the design and try to find the errors. To increase the reliability of the program, errors must be discovered and fixed before the program is released to the users.

Once an algorithm is written, the next step is to verify that it works properly. If the algorithm is a simple sequential flow or contains a branch, it can be hand traced or you can use the debugger, if any, provided by the IDE. Typically, loops are harder to debug. The correctness of a loop can be verified by using loop invariants. A loop invariant is a set of statements that remains true each time the loop body is executed. Let p be a loop invariant and q be the (logical) expression in a loop statement. Then $p \ \&\& \ q$ remains true before each iteration of the loop and $p \ \&\& \ \text{not}(q)$ is true after the loop terminates. The full discussion of loop invariants is beyond the scope of the book. However, you can learn about loop invariants in the book: *Discrete Mathematical Structures: Theory and Applications*, D.S. Malik and M.K. Sen, Course Technology, 2004. Here, we give a few tips that you can use to debug a loop.

As discussed in the previous section, the most common error associated with loops is off-by-one. If a loop turns out to be an infinite loop, the error is most likely in the logical expression that controls the execution of the loop. Check the logical expression carefully and see if you have reversed an inequality, an assignment statement symbol appears in place of the equality operator, or `&&` appears in place of `||`. If the loop changes the values of

variables, you can print the values of the variables before and/or after each iteration or you can use your IDE's debugger, if any, and watch the values of variables during each iteration.

The debugging sections in this book are designed to help you understand the debugging process. However, as you will realize, debugging can be a tiresome process. If your program is very bad, do not debug. Throw it away and start over.

QUICK REVIEW

1. C++ has three looping (repetition) structures: **while**, **for**, and **do...while**.
2. The syntax of the **while** statement is:

```
while (expression)
    statement
```
3. In C++, **while** is a reserved word.
4. In the **while** statement, the parentheses around the **expression** (the decision maker) are important; they mark the beginning and end of the expression.
5. The **statement** is called the body of the loop.
6. The body of the **while** loop must contain a statement that eventually sets the expression to **false**.
7. A counter-controlled **while** loop uses a counter to control the loop.
8. In a counter-controlled **while** loop, you must initialize the counter before the loop, and the body of the loop must contain a statement that changes the value of the counter variable.
9. A sentinel is a special value that marks the end of the input data. The sentinel must be similar to, yet differ from, all the data items.
10. A sentinel-controlled **while** loop uses a sentinel to control the **while** loop. The **while** loop continues to execute until the sentinel is read.
11. An EOF-controlled **while** loop continues to execute until the program detects the end-of-file marker.
12. In the Windows console environment, the end-of-file marker is entered using **Ctrl+z** (hold the **Ctrl** key and press **z**). In the UNIX environment, the end-of-file marker is entered using **Ctrl+d** (hold the **Ctrl** key and press **d**).
13. A **for** loop simplifies the writing of a counter-controlled **while** loop.
14. In C++, **for** is a reserved word.
15. The syntax of the **for** loop is:

```
for (initialize statement; loop condition; update statement)
    statement
```

statement is called the body of the **for** loop.

16. Putting a semicolon at the end of the **for** loop (before the body of the **for** loop) is a semantic error. In this case, the action of the **for** loop is empty.
17. The syntax of the **do...while** statement is:

```
do
    statement
while (expression);
```

statement is called the body of the **do...while** loop.

18. Both **while** and **for** loops are called pretest loops. A **do...while** loop is called a posttest loop.
19. The **while** and **for** loops may not execute at all, but the **do...while** loop always executes at least once.
20. Executing a **break** statement in the body of a loop immediately terminates the loop.
21. Executing a **continue** statement in the body of a loop skips the loop's remaining statements and proceeds with the next iteration.
22. When a **continue** statement executes in a **while** or **do...while** loop, the expression update statement in the body of the loop may not execute.
23. After a **continue** statement executes in a **for** loop, the update statement is the next statement executed.

EXERCISES

1. Mark the following statements as true or false.
 - a. In a counter-controlled **while** loop, it is not necessary to initialize the loop control variable.
 - b. It is possible that the body of a **while** loop may not execute at all.
 - c. In an infinite **while** loop, the **while** expression (the decision maker) is initially false, but after the first iteration it is always true.
 - d. The **while** loop:


```
j = 0;
while (j <= 10)
    j++;
```

 terminates if $j > 10$.
 - e. A sentinel-controlled **while** loop is an event-controlled **while** loop whose termination depends on a special value.
 - f. A loop is a control structure that causes certain statements to execute over and over.
 - g. To read data from a file of an unspecified length, an EOF-controlled loop is a good choice.

h. When a **while** loop terminates, the control first goes back to the statement just before the **while** statement, and then the control goes to the statement immediately following the **while** loop.

2. What is the output of the following C++ code?

```
int count = 1;
int y = 100;
while (count < 100)
{
    y = y - 1;
    count++;
}
cout << " y = " << y << " and count = " << count << endl;
```

3. What is the output of the following C++ code?

```
int num = 5;
while (num > 5)
    num = num + 2;
cout << num << endl;
```

4. What is the output of the following C++ code?

```
int num = 1;
while (num < 10)
{
    cout << num << " ";
    num = num + 2;
}
cout << endl;
```

5. When does the following **while** loop terminate?

```
ch = 'D';
while ('A' <= ch && ch <= 'Z')
    ch = static_cast<char>(static_cast<int>(ch) + 1);
```

6. Suppose that the input is 38 35 71 14 -1. What is the output of the following code? Assume all variables are properly declared.

```
cin >> sum;
cin >> num;

for (j = 1; j <= 3; j++)
{
    cin >> num;
    sum = sum + num;
}
cout << "Sum = " << sum << endl;
```

7. Suppose that the input is 38 35 71 14 -1. What is the output of the following code? Assume all variables are properly declared.

```
cin >> sum;
cin >> num;

while (num != -1)
```



```

{
    sum = sum + num;
    cin >> num;
}
cout << "Sum = " << sum << endl;

```

8. Suppose that the input is 38 35 71 14 -1. What is the output of the following code? Assume all variables are properly declared.

```

cin >> num;
sum = num;

while (num != -1)
{
    cin >> num;
    sum = sum + num;
}
cout << "Sum = " << sum << endl;

```

9. Suppose that the input is 38 35 71 14 -1. What is the output of the following code? Assume all variables are properly declared.

```

sum = 0;
cin >> num;

while (num != -1)
{
    sum = sum + num;
    cin >> num;
}
cout << "Sum = " << sum << endl;

```

10. Correct the following code so that it finds the sum of 20 numbers.

```

sum = 0;

while (count < 20)
    cin >> num;
    sum = sum + num;
    count++;

```

11. What is the output of the following program?

```

#include <iostream>

using namespace std;

int main()
{
    int x, y, z;

    x = 4;    y = 5;
    z = y + 6;

    while (((z - x) % 4) != 0)
    {
        cout << z << " ";
        z = z + 7;
    }
}

```

```

        cout << endl;

        return 0;
    }

```

12. Suppose that the input is:

```
58 23 46 75 98 150 12 176 145 -999
```

What is the output of the following program?

```

#include <iostream>

using namespace std;

int main()
{
    int num;

    cin >> num;

    while (num != -999)
    {
        cout << num % 25 << " ";
        cin >> num;
    }

    cout << endl;

    return 0;
}

```

13. The following program is designed to input two numbers and output their sum. It asks the user if he/she would like to run the program. If the answer is Y or y, it prompts the user to enter two numbers. After adding the numbers and displaying the results, it again asks the user if he/she would like to add more numbers. However, the program fails to do so. Correct the program so that it works properly.

```

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    char response;
    double num1;
    double num2;

    cout << "This program adds two numbers." << endl;
    cout << "Would you like to run the program: (Y/y) ";
    cin >> response;
    cout << endl;
}

```

```

cout << fixed << showpoint << setprecision(2);

while (response == 'Y' && response == 'y')
{
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;
    cout << endl;

    cout << num1 << " + " << num2 << " = " << (num1 - num2)
        << endl;

    cout << "Would you like to add again: (Y/y) ";
    cin >> response;
    cout << endl;
}

return 0;
}

```

14. What is the output of the following program segment?

```

int count = 0;

while (count++ < 10)
    cout << "This loop can repeat statements." << endl;

```

15. What is the output of the following program segment?

```

int count = 5;

while (--count > 0)
    cout << count << " ";
cout << endl;

```

16. What is the output of the following program segment?

```

int count = 5;

while (count-- > 0)
    cout << count << " ";
cout << endl;

```

17. What is the output of the following program segment?

```

int count = 1;
while (count++ <= 5)
    cout << count * (count - 2) << " ";

cout << endl;

```

18. What type of loop, such as counter-control and sentinel-control, will you use in each of the following situations?

- Sum the following series: $1 + (2 / 1) + (3 / 2) + (4 / 3) + (5 / 4) + \dots + (10 / 9)$
- Sum the following numbers, except the last number: 17, 32, 62, 48, 58, -1
- A file contains an employee's salary. Update the employee's salary.

19. Consider the following **for** loop:

```
int j, s;

s = 0;
for (j = 1; j <= 10; j++)
    s = s + j * (j - 1);
```

In this **for** loop, identify the loop control variable, the initialization statement, the loop condition, the update statement, and the statement that updates the value of **s**.

20. Given that the following code is correctly inserted into a program, state its entire output as to content and form. (Assume all variables are properly declared.)

```
num = 0;
for (i = 1; i <= 4; i++)
{
    num = num + 10 * (i - 1);
    cout << num << " ";
}
cout << endl;
```

21. Given that the following code is correctly inserted into a program, state its entire output as to content and form. (Assume all variables are properly declared.)

```
j = 2;
for (i = 0; i <= 5; i++)
{
    cout << j << " ";
    j = 2 * j + 3;
}
cout << j << " " << endl;
```

22. Assume that the following code is correctly inserted into a program:

```
int s = 0;

for (i = 0; i < 5; i++)
{
    s = 2 * s + i;
    cout << s << " ";
}
cout << endl;
```

- a. What is the final value of **s**?
 - (i) 11
 - (ii) 4
 - (iii) 26
 - (iv) none of these
- b. If a semicolon is inserted after the right parenthesis in the **for** loop statement, what is the final value of **s**?
 - (i) 0
 - (ii) 1
 - (iii) 2
 - (iv) 5
 - (v) none of these
- c. If the 5 is replaced with a 0 in the **for** loop control expression, what is the final value of **s**?
 - (i) 0
 - (ii) 1
 - (iii) 2
 - (iv) none of these

23. State what output, if any, results from each of the following statements:

- a.

```
for (i = 1; i <= 1; i++)
    cout << "*";
cout << endl;
```
- b.

```
for (i = 2; i >= 1; i++)
    cout << "*";
cout << endl;
```
- c.

```
for (i = 1; i <= 1; i--)
    cout << "*";
cout << endl;
```
- d.

```
for (i = 12; i >= 9; i--)
    cout << "*";
cout << endl;
```
- e.

```
for (i = 0; i <= 5; i++)
    cout << "*";
cout << endl;
```
- f.

```
for (i = 1; i <= 5; i++)
{
    cout << "*";
    i = i + 1;
}
cout << endl;
```

24. Write a **for** statement to add all the multiples of 3 between 1 and 100.

25. What is the output of the following code? Is there a relationship between the variables **x** and **y**? If yes, state the relationship? What is the output?

```
int x = 19683;
int i;
int y = 0;

for (i = x; i >= 1; i = i / 3)
    y++;
cout << "x = " << x << ", y = " << y << endl;
```

26. Suppose that the input is 5 3 8. What is the output of the following code? Assume all variables are properly declared.

```
cin >> a >> b >> c;
for (j = 1; j < a; j++)
{
    d = b + c;
    b = c;
    c = d;
    cout << c << " ";
}
cout << endl;
```

27. What is the output of the following C++ program segment? Assume all variables are properly declared.

```

for (j = 0; j < 8; j++)
{
    cout << j * 25 << " - ";

    if (j != 7)
        cout << (j + 1) * 25 - 1 << endl;
    else
        cout << (j + 1) * 25 << endl;
}

```

28. Suppose that the input is 38 35 71 44 -1. What is the output of the following code? Assume all variables are properly declared.

```

sum = 0;
cin >> num;

for (j = 1; j <= 3; j++)
{
    cin >> num;
    sum = sum + num;
}
cout << "Sum = " << sum << endl;

```

29. Which of the following apply to the **while** loop only? To the **do...while** loop only? To both?
- It is considered a conditional loop.
 - The body of the loop executes at least once.
 - The logical expression controlling the loop is evaluated before the loop is entered.
 - The body of the loop may not execute at all.
30. The following program has more than five mistakes that prevent it from compiling and/or running. Correct all such mistakes.

```

#include <iostream>

using namespace std;
const int N = 2,137;

main ()
{
    int a, b, c, d:

    a := 3;
    b = 5;
    c = c + d;
    N = a + n;
    for (i = 3; i <= N; i++)
    {
        cout << setw(5) << i;
        i = i + 1;
    }
    return 0;
}

```

31. What is the difference between a pretest loop and a posttest loop?
32. How many times will each of the following loops execute? What is the output in each case?

a. `x = 5; y = 50;`
`do`
 `x = x + 10;`
`while (x < y);`
`cout << x << " " << y << endl;`

b. `x = 5; y = 80;`
`do`
 `x = x * 2;`
`while (x < y);`
`cout << x << " " << y << endl;`

c. `x = 5; y = 20;`
`do`
 `x = x + 2;`
`while (x >= y);`
`cout << x << " " << y << endl;`

d. `x = 5; y = 35;`
`while (x < y)`
 `x = x + 10;`
`cout << x << " " << y << endl;`

e. `x = 5; y = 30;`
`while (x <= y)`
 `x = x * 2;`
`cout << x << " " << y << endl;`

f. `x = 5; y = 30;`
`while (x > y)`
 `x = x + 2;`
`cout << x << " " << y << endl;`

33. Write an input statement validation loop that prompts the user to enter a number less than 20 or greater than 75.

34. Rewrite the following as a **for** loop.

```
int i = 0, value = 0;

while (i <= 20)
{
    if (i % 2 == 0 && i <= 10)
        value = value + i * i;
    else if (i % 2 == 0 && i > 10)
        value = value + i;
    else
        value = value - i;
    i = i + 1;
}

cout << "value = " << value << endl;
```

What is the output of this loop?

35. Write the `while` loop of Exercise 34 as a `do...while` loop.
36. The `do...while` loop in the following program is supposed to read some numbers until it reaches a sentinel (in this case, `-1`). It is supposed to add all of the numbers except for the sentinel. If the data looks like:

12 5 30 48 -1

the program does not add the numbers correctly. Correct the program so that it adds the numbers correctly.

```
#include <iostream>

using namespace std;
int main()
{
    int total = 0,
        count = 0,
        number;

    do
    {
        cin >> number;
        total = total + number;
        count++;
    }
    while (number != -1);

    cout << "The number of data read is " << count << endl;
    cout << "The sum of the numbers entered is " << total
        << endl;

    return 0;
}
```

37. Using the same data as in Exercise 36, the following loop also fails. Correct it.

```
cin >> number;
while (number != -1)
    total = total + number;
cin >> number;
cout << endl;
cout << total << endl;
```

38. Using the same data as in Exercise 36, the following loop also fails. Correct it.

```
cin >> number;
while (number != -1)
{
    cin >> number;
    total = total + number;
}
cout << endl;
cout << total << endl;
```

39. Given the following program segment:

```
for (number = 1; number <= 10; number++)
    cout << setw(3) << number;
```

write a `while` loop and a `do...while` loop that have the same output.

40. Given the following program segment:

```
j = 2;
for (i = 1; i <= 5; i++);
{
    cout << setw(4) << j;
    j = j + 5;
}
cout << endl;
```

write a **while** loop and a **do...while** loop that have the same output.

41. What is the output of the following program?

```
#include <iostream>

using namespace std;

int main()
{
    int x, y, z;
    x = 4; y = 5;
    z = y + 6;
    do
    {
        cout << z << " ";
        z = z + 7;
    }
    while ((z - x) % 4 != 0);

    cout << endl;

    return 0;
}
```

42. To learn how nested **for** loops work, do a walk-through of the following program segments and determine, in each case, the exact output.

a. **int** i, j;

```
for (i = 1; i <= 5; i++)
{
    for (j = 1; j <= 5; j++)
        cout << setw(3) << i;
    cout << endl;
}
```

b. **int** i, j;

```
for (i = 1; i <= 5; i++)
{
    for (j = (i + 1); j <= 5; j++)
        cout << setw(5) << j;
    cout << endl;
}
```

```

c.  int i, j;
    for (i = 1; i <= 5; i++)
    {
        for (j = 1; j <= i; j++)
            cout << setw(3) << j;
        cout << endl;
    }

d.  const int M = 10;
    const int N = 10;
    int i, j;

    for (i = 1; i <= M; i++)
    {
        for (j = 1; j <= N; j++)
            cout << setw(3) << M * (i - 1) + j;
        cout << endl;
    }

e.  int i, j;

    for (i = 1; i <= 9; i++)
    {
        for (j = 1; j <= (9 - i); j++)
            cout << " ";
        for (j = 1; j <= i; j++)
            cout << setw(1) << j;
        for (j = (i - 1); j >= 1; j--)
            cout << setw(1) << j;
        cout << endl;
    }

```

43. What is the output of the following program segment?

```

int count = 1;
do
    cout << count * (count - 2) << " ";
while (count++ <= 5);

cout << endl;

```

44. What is the output of the following code?

```

int num = 12;

while (num >= 0)
{
    if (num % 5 == 0)
        break;

    cout << num << " ";
    num = num - 2;
}

cout << endl;

```

45. What is the output of the following code?

```
int num = 12;

while (num >= 0)
{
    if (num % 5 == 0)
    {
        num++;
        continue;
    }

    cout << num << " ";
    num = num - 2;
}
cout << endl;
```

46. What does a **break** statement do in a loop?

PROGRAMMING EXERCISES

- Write a program that prompts the user to input an integer and then outputs both the individual digits of the number and the sum of the digits. For example, it should output the individual digits of 3456 as 3 4 5 6, output the individual digits of 8030 as 8 0 3 0, output the individual digits of 2345526 as 2 3 4 5 5 2 6, output the individual digits of 4000 as 4 0 0 0, and output the individual digits of -2345 as 2 3 4 5.

- The value of π can be approximated by using the following series:

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{1}{2n-1} + \frac{1}{2n+1} \right)$$

The following program uses this series to find the approximate value of π . However, the statements are in the incorrect order, and there is also a bug in this program. Rearrange the statements and also find and remove the bug so that this program can be used to approximate π .

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double pi = 0;
    long i;
    long n;

    cin >> n;
    cout << "Enter the value of n: ";
    cout << endl;
```

```

    if (i % 2 == 0)
        pi = pi + (1 / (2 * i + 1));
    else
        pi = pi - (1 / (2 * i + 1));

    for (i = 0; i < n; i++)
    {
        pi = 0;
        pi = 4 * pi;
    }

    cout << endl << "pi = " << pi << endl;

    return 0;
}

```

3. Rewrite the program of Example 5-5, Telephone Digits. Replace the statements from Line 10 to Line 28 so that the program uses only a **switch** structure to find the digit that corresponds to an uppercase letter.
4. The program Telephone Digits outputs only telephone digits that correspond to uppercase letters. Rewrite the program so that it processes both uppercase and lowercase letters and outputs the corresponding telephone digit. If the input is something other than an uppercase or lowercase letter, the program must output an appropriate error message.
5. To make telephone numbers easier to remember, some companies use letters to show their telephone number. For example, using letters, the telephone number 438-5626 can be shown as **GET LOAN**. In some cases, to make a telephone number meaningful, companies might use more than seven letters. For example, 225-5466 can be displayed as **CALL HOME**, which uses eight letters. Write a program that prompts the user to enter a telephone number expressed in letters and outputs the corresponding telephone number in digits. If the user enters more than seven letters, then process only the first seven letters. Also output the – (hyphen) after the third digit. Allow the user to use both uppercase and lowercase letters as well as spaces between words. Moreover, your program should process as many telephone numbers as the user wants.
6. Write a program that reads a set of integers and then finds and prints the sum of the even and odd integers.
7. Write a program that prompts the user to input a positive integer. It should then output a message indicating whether the number is a prime number. (*Note:* An even number is prime if it is 2. An odd integer is prime if it is not divisible by any odd integer less than or equal to the square root of the number.)
8. Let $n = a_k a_{k-1} a_{k-2} \dots a_1 a_0$ be an integer and $t = a_0 - a_1 + a_2 - \dots + (-1)^k a_k$. It is known that n is divisible by 11 if and only if t is divisible by 11. For example, suppose that $n = 8784204$. Then, $t = 4 - 0 + 2 - 4 + 8 - 7 + 8 = 11$. Because 11 is divisible by 11, it follows that 8784204 is divisible by 11. If $n = 54063297$, then $t = 7 - 9 + 2 - 3 + 6 - 0 + 4 - 5 = 2$. Because 2 is not

divisible by 11, 54063297 is not divisible by 11. Write a program that prompts the user to enter a positive integer and then uses this criterion to determine whether the number is divisible by 11.

9. Write a program that uses **while** loops to perform the following steps:
 - a. Prompt the user to input two integers: **firstNum** and **secondNum** (**firstNum** must be less than **secondNum**).
 - b. Output all odd numbers between **firstNum** and **secondNum**.
 - c. Output the sum of all even numbers between **firstNum** and **secondNum**.
 - d. Output the numbers and their squares between 1 and 10.
 - e. Output the sum of the square of the odd numbers between **firstNum** and **secondNum**.
 - f. Output all uppercase letters.
10. Redo Exercise 9 using **for** loops.
11. Redo Exercise 9 using **do...while** loops.
12. The program in the Programming Example: Fibonacci Number does not check whether the first number entered by the user is less than or equal to the second number and whether both the numbers are nonnegative. Also, the program does not check whether the user entered a valid value for the position of the desired number in the Fibonacci sequence. Rewrite that program so that it checks for these things.
13. The population of a town A is less than the population of town B . However, the population of town A is growing faster than the population of town B . Write a program that prompts the user to enter the population and growth rate of each town. The program outputs after how many years the population of town A will be greater than or equal to the population of town B and the populations of both the towns at that time. (A sample input is: Population of town $A = 5000$, growth rate of town $A = 4\%$, population of town $B = 8000$, and growth rate of town $B = 2\%$.)
14. Suppose that the first number of a sequence is x , in which x is an integer. Define $a_0 = x$; $a_{n+1} = a_n/2$ if a_n is even; $a_{n+1} = 3 \times a_n + 1$ if a_n is odd. Then, there exists an integer k such that $a_k = 1$. Write a program that prompts the user to input the value of x . The program output the integer k such that $a_k = 1$ and the numbers $a_0, a_1, a_2, \dots, a_k$. (For example, if $x = 75$, then $k = 14$, and the numbers $a_0, a_1, a_2, \dots, a_{14}$, respectively, are 75, 226, 113, 340, 170, 85, 256, 128, 64, 32, 16, 8, 4, 2, 1.) Test your program for the following values of x : 75, 111, 678, 732, 873, 2048, and 65535.
15. Enhance your program from Exercise 14 by outputting the position of the largest number and the largest number of the sequence $a_0, a_1, a_2, \dots, a_k$. (For example, the largest number of the sequence 75, 226, 113, 340, 170,

85, 256, 128, 64, 32, 16, 8, 4, 2, 1 is 340, and its position is 4.) Test your program for the following values of x : 75, 111, 678, 732, 873, 2048, and 65535.

16. The program in Example 5-6 implements the Number Guessing Game. However, in that program, the user is given as many tries as needed to guess the correct number. Rewrite the program so that the user has no more than five tries to guess the correct number. Your program should print an appropriate message, such as “You win!” or “You lose!”.
17. Example 5-6 implements the Number Guessing Game program. If the guessed number is not correct, the program outputs a message indicating whether the guess is low or high. Modify the program as follows: Suppose that the variables `num` and `guess` are as declared in Example 5-6 and `diff` is an `int` variable. Let `diff` = the absolute value of (`num` – `guess`). If `diff` is 0, then `guess` is correct and the program outputs a message indicating that the user guessed the correct number. Suppose `diff` is not 0. Then the program outputs the message as follows:
 - a. If `diff` is greater than or equal to 50, the program outputs the message indicating that the guess is very high (if `guess` is greater than `num`) or very low (if `guess` is less than `num`).
 - b. If `diff` is greater than or equal to 30 and less than 50, the program outputs the message indicating that the guess is high (if `guess` is greater than `num`) or low (if `guess` is less than `num`).
 - c. If `diff` is greater than or equal to 15 and less than 30, the program outputs the message indicating that the guess is moderately high (if `guess` is greater than `num`) or moderately low (if `guess` is less than `num`).
 - d. If `diff` is greater than 0 and less than 15, the program outputs the message indicating that the guess is somewhat high (if `guess` is greater than `num`) or somewhat low (if `guess` is less than `num`).

As in Programming Exercise 16, give the user no more than five tries to guess the number. (To find the absolute value of `num` – `guess`, use the expression `abs(num - guess)`. The function `abs` is from the header file `cstdlib`.)

18. A high school has 1000 students and 1000 lockers, one locker for each student. On the first day of school, the principal plays the following game: She asks the first student to go and open all the lockers. She then asks the second student to go and close all the even-numbered lockers. The third student is asked to check every third locker. If it is open, the student closes it; if it is closed, the student opens it. The fourth student is asked to check every fourth locker. If it is open, the student closes it; if it is closed, the student opens it. The remaining students continue this game. In general, the n th student checks every n th locker. If the locker is open, the student closes it; if it is closed, the student opens it. After all the students have taken their turn, some of the lockers are open and some are closed. Write a program

that prompts the user to enter the number of lockers in a school. After the game is over, the program outputs the number of lockers that are opened. Test run your program for the following inputs: 1000, 5000, 10000. Do you see any pattern developing?

(*Hint:* Consider locker number 100. This locker is visited by student numbers 1, 2, 4, 5, 10, 20, 25, 50, and 100. These are the positive divisors of 100. Similarly, locker number 30 is visited by student numbers 1, 2, 3, 5, 6, 10, 15, and 30. Notice that if the number of positive divisors of a locker number is odd, then at the end of the game, the locker is opened. If the number of positive divisors of a locker number is even, then at the end of the game, the locker is closed.)

19. When you borrow money to buy a house, a car, or for some other purpose, you repay the loan by making periodic payments over a certain period of time. Of course, the lending company will charge interest on the loan. Every periodic payment consists of the interest on the loan and the payment toward the principal amount. To be specific, suppose that you borrow \$1000 at the interest rate of 7.2% per year and the payments are monthly. Suppose that your monthly payment is \$25. Now, the interest is 7.2% per year and the payments are monthly, so the interest rate per month is $7.2/12 = 0.6\%$. The first month's interest on \$1000 is $1000 \times 0.006 = 6$. Because the payment is \$25 and interest for the first month is \$6, the payment toward the principal amount is $25 - 6 = 19$. This means after making the first payment, the loan amount is $1000 - 19 = 981$. For the second payment, the interest is calculated on \$981. So the interest for the second month is $981 \times 0.006 = 5.886$, that is, approximately \$5.89. This implies that the payment toward the principal is $25 - 5.89 = 19.11$ and the remaining balance after the second payment is $981 - 19.11 = 961.89$. This process is repeated until the loan is paid. Write a program that accepts as input the loan amount, the interest rate per year, and the monthly payment. (Enter the interest rate as a percentage. For example, if the interest rate is 7.2% per year, then enter 7.2.) The program then outputs the number of months it would take to repay the loan. (Note that if the monthly payment is less than the first month's interest, then after each payment, the loan amount will increase. In this case, the program must warn the borrower that the monthly payment is too low, and with this monthly payment, the loan amount could not be repaid.)
20. Enhance your program from Exercise 19 by first telling the user the minimum monthly payment and then prompting the user to enter the monthly payment. Your last payment might be more than the remaining loan amount and interest on it. In this case, output the loan amount before the last payment and the actual amount of the last payment. Also, output the total interest paid.
21. Write a complete program to test the code in Example 5-21.
22. Write a complete program to test the code in Example 5-22.

23. Write a complete program to test the code in Example 5-23.
24. Write a complete program to test the code in Example 5-24.
25. Write a complete program to test the code in Example 5-25.
26. **(The conical paper cup problem)** You have been given the contract for making little conical cups that come with bottled water. These cups are to be made from a circular waxed paper of 4 inches in radius by removing a sector of length x (see Figure 5-4). By closing the remaining part of the circle, a conical cup is made. Your objective is to remove the sector so that the cup is of maximum volume.

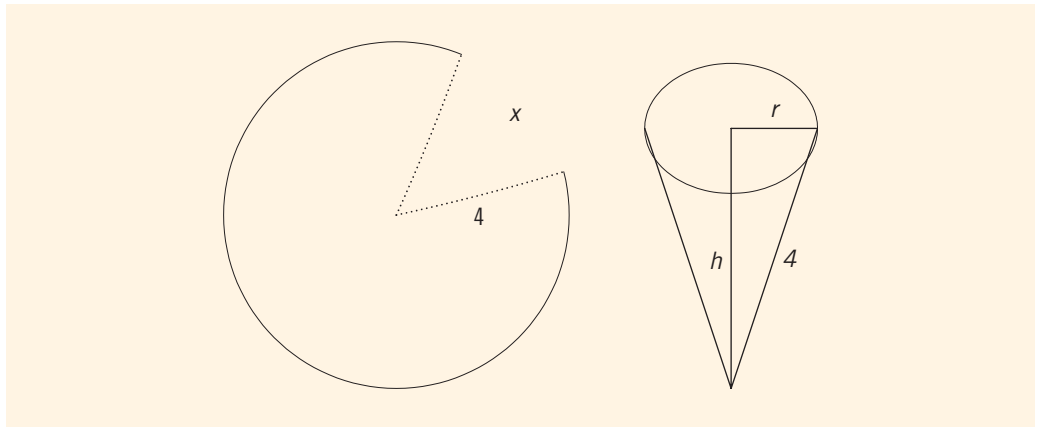


FIGURE 5-4 Conical paper cup

Write a program that prompts the user to enter the radius of the circular waxed paper. The program should then output the length of the removed sector so that the resulting cup is of maximum volume. Calculate your answer to two decimal places.

27. **(Apartment problem)** A real estate office handles, say, 50 apartment units. When the rent is, say, \$600 per month, all the units are occupied. However, for each, say, \$40 increase in rent, one unit becomes vacant. Moreover, each occupied unit requires an average of \$27 per month for maintenance. How many units should be rented to maximize the profit?

Write a program that prompts the user to enter:

- a. The rent to occupy all the units.
- b. The increase in rent that results in a vacant unit.
- c. Amount to maintain a rented unit.

The program then outputs the number of units to be rented to maximize the profit.



6 CHAPTER

USER-DEFINED FUNCTIONS I

IN THIS CHAPTER, YOU WILL:

- Learn about standard (predefined) functions and discover how to use them in a program
- Learn about user-defined functions
- Examine value-returning functions, including actual and formal parameters
- Explore how to construct and use a value-returning, user-defined function in a program

In Chapter 2, you learned that a C++ program is a collection of functions. One such function is `main`. The programs in Chapters 1 through 5 use only the function `main`; the programming instructions are packed into one function. This technique, however, is good only for short programs. For large programs, it is not practical (although it is possible) to put the entire programming instructions into one function, as you will soon discover. You must learn to break the problem into manageable pieces. This chapter first discusses the functions previously defined and then discusses user-defined functions.

Let us imagine an automobile factory. When an automobile is manufactured, it is not made from basic raw materials; it is put together from previously manufactured parts. Some parts are made by the company itself; others, by different companies.

Functions are like building blocks. They let you divide complicated programs into manageable pieces. They have other advantages, too:

- While working on one function, you can focus on just that part of the program and construct it, debug it, and perfect it.
- Different people can work on different functions simultaneously.
- If a function is needed in more than one place in a program or in different programs, you can write it once and use it many times.
- Using functions greatly enhances the program's readability because it reduces the complexity of the function `main`.

Functions are often called **modules**. They are like miniature programs; you can put them together to form a larger program. When user-defined functions are discussed, you will see that this is the case. This ability is less apparent with predefined functions because their programming code is not available to us. However, because predefined functions are already written for us, you will learn these first so that you can use them when needed.

Predefined Functions

Before formally discussing predefined functions in C++, let us review a concept from a college algebra course. In algebra, a function can be considered a rule or correspondence between values, called the function's arguments, and the unique values of the function associated with the arguments. Thus, if $f(x) = 2x + 5$, then $f(1) = 7$, $f(2) = 9$, and $f(3) = 11$, where 1, 2, and 3 are the arguments of f , and 7, 9, and 11 are the corresponding values of the function f .

In C++, the concept of a function, either predefined or user-defined, is similar to that of a function in algebra. For example, every function has a name and, depending on the values specified by the user, it does some computation. This section discusses various predefined functions.

Some of the predefined mathematical functions are `pow(x, y)`, `sqrt(x)`, and `floor(x)`.

The *power* function, `pow(x, y)`, calculates x^y ; that is, the value of `pow(x, y) = x^y` . For example, `pow(2, 3) = $2^3 = 8.0$` and `pow(2.5, 3) = $2.5^3 = 15.625$` . Because the value of `pow(x, y)` is of type `double`, we say that the function `pow` is of type `double` or that the function `pow` returns a value of type `double`. Moreover, `x` and `y` are called the parameters (or arguments) of the function `pow`. Function `pow` has two parameters.

The *square root* function, `sqrt(x)`, calculates the nonnegative square root of `x` for `x >= 0.0`. For example, `sqrt(2.25)` is `1.5`. The function `sqrt` is of type `double` and has only one parameter.

The *floor* function, `floor(x)`, calculates the largest whole number that is less than or equal to `x`. For example, `floor(48.79)` is `48.0`. The function `floor` is of type `double` and has only one parameter.

In C++, predefined functions are organized into separate libraries. For example, the header file `iostream` contains I/O functions, and the header file `cmath` contains math functions. Table 6-1 lists some of the predefined functions, the name of the header file in which each function's specification can be found, the data type of the parameters, and the function type. The function type is the data type of the final value returned by the function. (For a list of additional predefined functions, see Appendix F.)

TABLE 6-1 Predefined Functions

Function	Header File	Purpose	Parameter(s) Type	Result
<code>abs(x)</code>	<code><cmath></code>	Returns the absolute value of its argument: <code>abs(-7) = 7</code>	<code>int</code> (<code>double</code>)	<code>int</code> (<code>double</code>)
<code>ceil(x)</code>	<code><cmath></code>	Returns the smallest whole number that is not less than <code>x</code> : <code>ceil(56.34) = 57.0</code>	<code>double</code>	<code>double</code>
<code>cos(x)</code>	<code><cmath></code>	Returns the cosine of angle: <code>x: cos(0.0) = 1.0</code>	<code>double</code> (radians)	<code>double</code>
<code>exp(x)</code>	<code><cmath></code>	Returns e^x , where $e = 2.718$: <code>exp(1.0) = 2.71828</code>	<code>double</code>	<code>double</code>
<code>fabs(x)</code>	<code><cmath></code>	Returns the absolute value of its argument: <code>fabs(-5.67) = 5.67</code>	<code>double</code>	<code>double</code>

TABLE 6-1 Predefined Functions (continued)

Function	Header File	Purpose	Parameter(s) Type	Result
<code>floor(x)</code>	<code><cmath></code>	Returns the largest whole number that is not greater than <code>x</code> : <code>floor(45.67) = 45.00</code>	<code>double</code>	<code>double</code>
<code>islower(x)</code>	<code><cctype></code>	Returns <code>true</code> if <code>x</code> is a lowercase letter; otherwise, it returns <code>false</code> ; <code>islower('h')</code> is <code>true</code>	<code>int</code>	<code>int</code>
<code>isupper(x)</code>	<code><cctype></code>	Returns <code>true</code> if <code>x</code> is an uppercase letter; otherwise, it returns <code>false</code> ; <code>isupper('K')</code> is <code>true</code>	<code>int</code>	<code>int</code>
<code>pow(x, y)</code>	<code><cmath></code>	Returns <code>x^y</code> ; if <code>x</code> is negative, <code>y</code> must be a whole number: <code>pow(0.16, 0.5) = 0.4</code>	<code>double</code>	<code>double</code>
<code>sqrt(x)</code>	<code><cmath></code>	Returns the nonnegative square root of <code>x</code> ; <code>x</code> must be nonnegative: <code>sqrt(4.0) = 2.0</code>	<code>double</code>	<code>double</code>
<code>tolower(x)</code>	<code><cctype></code>	Returns the lowercase value of <code>x</code> if <code>x</code> is uppercase; otherwise, it returns <code>x</code>	<code>int</code>	<code>int</code>
<code>toupper(x)</code>	<code><cctype></code>	Returns the uppercase value of <code>x</code> if <code>x</code> is lowercase; otherwise, it returns <code>x</code>	<code>int</code>	<code>int</code>

To use predefined functions in a program, you must include the header file that contains the function’s specification via the include statement. For example, to use the function `pow`, the program must include:

```
#include <cmath>
```

Example 6-1 shows you how to use some of the predefined functions.

EXAMPLE 6-1

```
//How to use predefined functions.
#include <iostream>
#include <cmath>
#include <cctype>

using namespace std;

int main()
{
    int    x;
    double u, v;

    cout << "Line 1: Uppercase a is "
          << static_cast<char>(toupper('a'))
          << endl;                                     //Line 1

    u = 4.2;                                           //Line 2
    v = 3.0;                                           //Line 3
    cout << "Line 4: " << u << " to the power of "
          << v << " = " << pow(u, v) << endl;         //Line 4

    cout << "Line 5: 5.0 to the power of 4 = "
          << pow(5.0, 4) << endl;                       //Line 5

    u = u + pow(3.0, 3);                             //Line 6
    cout << "Line 7: u = " << u << endl;               //Line 7

    x = -15;                                           //Line 8
    cout << "Line 9: Absolute value of " << x
          << " = " << abs(x) << endl;                 //Line 9

    return 0;
}
```

Sample Run:

```
Line 1: Uppercase a is A
Line 4: 4.2 to the power of 3 = 74.088
Line 5: 5.0 to the power of 4 = 625
Line 7: u = 31.2
Line 9: Absolute value of -15 = 15
```

This program works as follows. The statement in Line 1 outputs the uppercase letter that corresponds to 'a', which is A. Note that the function `toupper` returns an `int` value. Therefore, the value of the expression `toupper('a')` is 65, which is the ASCII value of 'A'. To print A rather than 65, you need to apply the `cast` operator, as shown in the statement in Line 1. In the statement in Line 4, the function `pow` is used to output u^v . In C++ terminology, it is said that the function `pow` is called with the parameters `u` and `v`. In this case, the values of `u` and `v` are passed to the function `pow`. The other statements have similar meanings. Note that the program includes the header files `cctype` and `cmath`, because it uses the functions `toupper`, `pow`, and `abs` from these header files.

User-Defined Functions

As Example 6-1 illustrates, using functions in a program greatly enhances the program's readability because it reduces the complexity of the function `main`. Also, once you write and properly debug a function, you can use it in the program (or different programs) again and again without having to rewrite the same code repeatedly. For instance, in Example 6-1, the function `pow` is used more than once.

Because C++ does not provide every function that you will ever need and designers cannot possibly know a user's specific needs, you must learn to write your own functions.

User-defined functions in C++ are classified into two categories:

- **Value-returning functions**—functions that have a return type. These functions return a value of a specific data type using the `return` statement, which we will explain shortly.
- **Void functions**—functions that do not have a return type. These functions *do not* use a `return` statement to return a value.

The remainder of this chapter discusses value-returning functions. Many of the concepts discussed in regard to value-returning functions also apply to void functions. Chapter 7 describes void functions.

Value-Returning Functions

The previous section introduced some predefined C++ functions such as `pow`, `abs`, `islower`, and `toupper`. These are examples of value-returning functions. To use these functions in your programs, you must know the name of the header file that contains the functions' specification. You need to include this header file in your program using the include statement and know the following items:

1. The name of the function
2. The number of **parameters**, if any
3. The data type of each parameter
4. The data type of the value computed (that is, the value returned) by the function, called the type of the function

Because the value returned by a value-returning function is unique, the natural thing for you to do is to use the value in one of three ways:

- Save the value for further calculation.
- Use the value in some calculation.
- Print the value.

This suggests that a value-returning function is used:

- In an assignment statement.
- As a parameter in a function call.
- In an output statement.

That is, a value-returning function is used (called) in an expression.

Before we look at the syntax of a user-defined, value-returning function, let us consider the things associated with such functions. In addition to the four properties described previously, one more thing is associated with functions (both value-returning and void):

5. The code required to accomplish the task

The first four properties form what is called the **heading** of the function (also called the **function header**); the fifth property is called the **body** of the function. Together, these five properties form what is called the **definition** of the function. For example, for the function `abs`, the heading might look like:

```
int abs(int number)
```

Similarly, the function `abs` might have the following definition:

```
int abs(int number)
{
    if (number < 0)
        number = -number;

    return number;
}
```

The variable declared in the heading of the function `abs` is called the **formal parameter** of the function `abs`. Thus, the formal parameter of `abs` is `number`.

The program in Example 6-1 contains several statements that use the function `pow`. That is, in C++ terminology, the function `pow` is called several times. Later in this chapter, we discuss what happens when a function is called.

Suppose that the heading of the function `pow` is:

```
double pow(double base, double exponent)
```

From the heading of the function `pow`, it follows that the formal parameters of `pow` are `base` and `exponent`. Consider the following statements:

```
double u = 2.5;
double v = 3.0;
double x, y;

x = pow(u, v); //Line 1
y = pow(2.0, 3.2) + 5.1; //Line 2
cout << u << " to the power of 7 = " << pow(u, 7) << endl; //Line 3
```

In Line 1, the function `pow` is called with the parameters `u` and `v`. In this case, the values of `u` and `v` are passed to the function `pow`. In fact, the value of `u` is copied into `base`, and the value of `v` is copied into `exponent`. The variables `u` and `v` that appear in the call to the function `pow` in Line 1 are called the **actual parameters** of that call. In Line 2, the function `pow` is called with the parameters `2.0` and `3.2`. In this call, the value `2.0` is copied into `base`, and `3.2` is copied into `exponent`. Moreover, in this call of the function `pow`, the actual parameters are `2.0` and `3.2`, respectively. Similarly, in Line 3, the actual parameters of the function `pow` are `u` and `7`; the value of `u` is copied into `base`, and `7.0` is copied into `exponent`.

We can now formally present two definitions:

Formal Parameter: A variable declared in the function heading.

Actual Parameter: A variable or expression listed in a call to a function.

For predefined functions, you need to be concerned only with the first four properties. Software companies do not give out the actual source code, which is the body of the function. Otherwise, software costs would be exorbitant.

Syntax: Value-Returning function

The syntax of a value-returning function is:

```
functionType functionName(formal parameter list)
{
    statements
}
```

in which statements are usually declaration statements and/or executable statements. In this syntax, `functionType` is the type of the value that the function returns. The `functionType` is also called the **data type** or the **return type** of the value-returning function. Moreover, statements enclosed between curly braces form the body of the function.

Syntax: Formal Parameter List

The syntax of the formal parameter list is:

```
dataType identifier, dataType identifier, ...
```

Function Call

The syntax to call a value-returning function is:

```
functionName(actual parameter list)
```


Syntax: Actual Parameter List

The syntax of the actual parameter list is:

```
expression or variable, expression or variable, ...
```

(In this syntax, **expression** can be a single constant value.) Thus, to call a value-returning function, you use its name, with the actual parameters (if any) in parentheses.

A function's formal parameter list *can* be empty. However, if the formal parameter list is empty, the parentheses are still needed. The function heading of the value-returning function thus takes, if the formal parameter list is empty, the following form:

```
functionType functionName()
```

If the formal parameter list of a value-returning function is empty, the actual parameter is also empty in a function call. In this case (that is, an empty formal parameter list), in a function call, the empty parentheses are still needed. Thus, a call to a value-returning function with an empty formal parameter list is:

```
functionName()
```

In a function call, the number of actual parameters, together with their data types, must match with the formal parameters in the order given. That is, actual and formal parameters have a one-to-one correspondence. (Chapter 7 discusses functions with default parameters.)

As stated previously, a value-returning function is called in an expression. The expression can be part of either an assignment statement or an output statement, or a parameter in a function call. A function call in a program causes the body of the called function to execute.

return Statement

Once a value-returning function computes the value, the function returns this value via the **return** statement. In other words, it passes this value outside the function via the **return** statement.

Syntax: return Statement

The **return** statement has the following syntax:

```
return expr;
```

in which **expr** is a variable, constant value, or expression. The **expr** is evaluated, and its value is returned. The data type of the value that **expr** computes must match the function type.

In C++, **return** is a reserved word.

When a **return** statement executes in a function, the function immediately terminates and the control goes back to the caller. Moreover, the function call statement is replaced by the value returned by the **return** statement. When a **return** statement executes in the function **main**, the program terminates.

To put the ideas in this discussion to work, let us write a function that determines the larger of two numbers. Because the function compares two numbers, it follows that this function has two parameters and that both parameters are numbers. Let us assume that the data type of these numbers is floating-point (decimal)—say, **double**. Because the larger number is of type **double**, the function's data type is also **double**. Let us name this function **larger**. The only thing you need to complete this function is the body of the function. Thus, following the syntax of a function, you can write this function as follows:

```
double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}
```

Note that the function **larger** requires that you use an additional variable **max** (called a **local declaration**, in which **max** is a variable local to the function **larger**). Figure 6-1 describes various parts of the function **larger**.

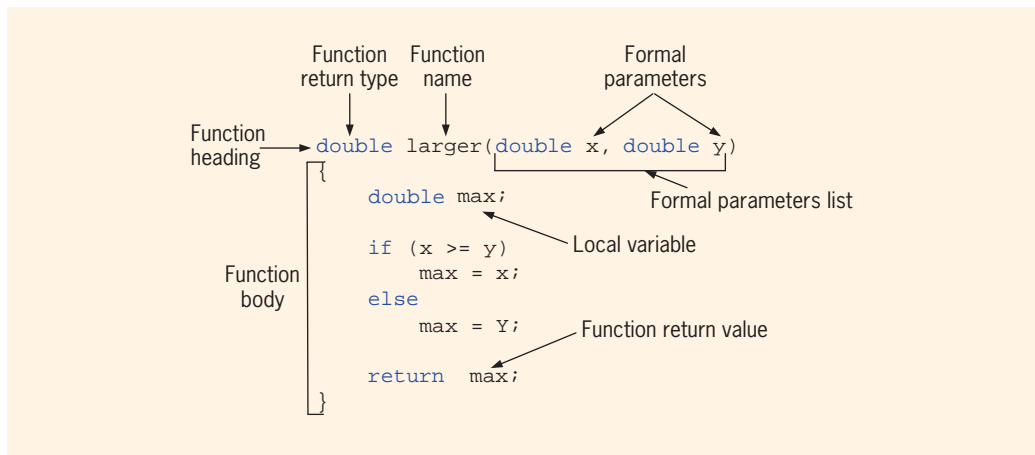


FIGURE 6-1 Various parts of the function `larger`

Suppose that `num`, `num1`, and `num2` are `double` variables. Also suppose that `num1 = 45.75` and `num2 = 35.50`. Figure 6-2 shows various calls to the function `larger`.

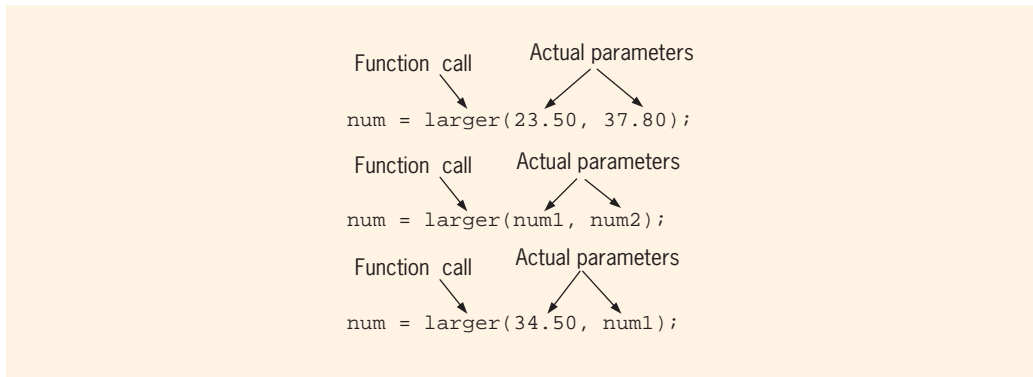


FIGURE 6-2 Function calls

You can also write the definition of the function `larger` as follows:

```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Because the execution of a `return` statement in a function terminates the function, the preceding function `larger` can also be written (without the word `else`) as:

```
double larger(double x, double y)
{
    if (x >= y)
        return x;

    return y;
}
```

Note that these forms of the function `larger` do not require you to declare any local variable.

NOTE

1. In the definition of the function `larger`, `x` and `y` are formal parameters.
2. The `return` statement can appear anywhere in the function. Recall that once a `return` statement executes, all subsequent statements are skipped. Thus, it's a good idea to return the value as soon as it is computed.

EXAMPLE 6-2

Now that the function `larger` is written, the following C++ code illustrates how to use it.

```
double one = 13.00;
double two = 36.53;
double maxNum;
```

Consider the following statements:

```
cout << "The larger of 5 and 6 is " << larger(5, 6)
    << endl;                                     //Line 1

cout << "The larger of " << one << " and " << two
    << " is " << larger(one, two) << endl;       //Line 2

cout << "The larger of " << one << " and 29 is "
    << larger(one, 29) << endl;                 //Line 3

maxNum = larger(38.45, 56.78);                  //Line 4
```

- The expression `larger(5, 6)` in Line 1 is a function call, and 5 and 6 are actual parameters. When the expression `larger(5, 6)` executes, 5 is copied into `x`, and 6 is copied into `y`. Therefore, the statement in Line 1 outputs the larger of 5 and 6.
- The expression `larger(one, two)` in Line 2 is a function call. Here, `one` and `two` are actual parameters. When the expression `larger(one, two)` executes, the value of `one` is copied into `x`, and the value of `two` is copied into `y`. Therefore, the statement in Line 2 outputs the larger of `one` and `two`.
- The expression `larger(one, 29)` in Line 3 is also a function call. When the expression `larger(one, 29)` executes, the value of `one` is copied into `x`, and 29 is copied into `y`. Therefore, the statement in Line 3 outputs the larger of `one` and 29.
- The expression `larger(38.45, 56.78)` in Line 4 is a function call. In this call, the actual parameters are 38.45 and 56.78. In this statement, the value returned by the function `larger` is assigned to the variable `maxNum`.

NOTE

In a function call, you specify only the actual parameter, not its data type. For example, in Example 6-2, the statements in Lines 1, 2, 3, and 4 show how to call the function `larger` with the actual parameters. However, the following statements contain incorrect calls to the function `larger` and would result in syntax errors. (Assume that all variables are properly declared.)

```
x = larger(int one, 29);           //illegal
y = larger(int one, int 29);       //illegal
cout << larger(int one, int two);   //illegal
```

Once a function is written, you can use it anywhere in the program. The function `larger` compares two numbers and returns the larger of the two. Let us now write another function that uses this function to determine the largest of three numbers. We call this function `compareThree`.

```
double compareThree(double x, double y, double z)
{
    return larger(x, larger(y, z));
}
```

In the function heading, `x`, `y`, and `z` are formal parameters.

Let us take a look at the expression:

```
larger(x, larger(y, z))
```

in the definition of the function `compareThree`. This expression has two calls to the function `larger`. The actual parameters to the outer call are `x` and `larger(y, z)`; the actual parameters to the inner call are `y` and `z`. It follows that, first, the expression `larger(y, z)` is evaluated; that is, the inner call executes first, which gives the larger of `y` and `z`. Suppose that `larger(y, z)` evaluates to, say, `t`. (Notice that `t` is either `y` or `z`.) Next, the outer call determines the larger of `x` and `t`. Finally, the `return` statement returns the largest number. It thus follows that to execute a function call, the parameters are evaluated first. For example, the actual parameter `larger(y, z)` of the outer call evaluates first.

Note that the function `larger` is much more general purpose than the function `compareThree`. Here, we are merely illustrating that once you have written a function, you can use it to write other functions. Later in this chapter, we will show how to use the function `larger` to determine the largest number from a set of numbers.

Function Prototype

Now that you have some idea of how to write and use functions in a program, the next question relates to the order in which user-defined functions should appear in a program. For example, do you place the function `larger` before or after the function `main`? Should `larger` be placed before `compareThree` or after it? Following the rule that you must declare an identifier before you can use it and knowing that the function `main` uses the identifier `larger`, logically you must place `larger` before `main`.

In reality, C++ programmers customarily place the function `main` before all other user-defined functions. However, this organization could produce a compilation error because functions are compiled in the order in which they appear in the program. For example, if the function `main` is placed before the function `larger`, the identifier `larger` is undefined when the function `main` is compiled. To work around this problem of undeclared identifiers, we place **function prototypes** before any function definition (including the definition of `main`).

Function Prototype: The function heading without the body of the function.

Syntax: Function Prototype

The general syntax of the function prototype of a value-returning function is:

```
functionType functionName(parameter list);
```

(Note that the function prototype ends with a semicolon.)

For the function `larger`, the prototype is:

```
double larger(double x, double y);
```

NOTE

When writing the function prototype, you do not have to specify the variable name in the parameter list. However, you must specify the data type of each parameter.

You can rewrite the function prototype of the function `larger` as follows:

```
double larger(double, double);
```

FINAL PROGRAM

You now know enough to write the entire program, compile it, and run it. The following program uses the functions `larger`, `compareThree`, and `main` to determine the larger/largest of two or three numbers.

```
//Program: Largest of three numbers

#include <iostream>

using namespace std;

double larger(double x, double y);
double compareThree(double x, double y, double z);

int main()
{
    double one, two;                                //Line 1

    cout << "Line 2: The larger of 5 and 10 is "
          << larger(5, 10) << endl;                  //Line 2

    cout << "Line 3: Enter two numbers: ";           //Line 3
    cin >> one >> two;                                //Line 4
    cout << endl;                                     //Line 5

    cout << "Line 6: The larger of " << one
          << " and " << two << " is "
          << larger(one, two) << endl;                //Line 6
}
```

```

        cout << "Line 7: The largest of 43.48, 34.00, "
              << "and 12.65 is "
              << compareThree(43.48, 34.00, 12.65)
              << endl;                                     //Line 7

    return 0;
}

double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}

double compareThree (double x, double y, double z)
{
    return larger(x, larger(y, z));
}

```

Sample Run: In this sample run, the user input is shaded.

Line 2: The larger of 5 and 10 is 10

Line 3: Enter two numbers: 25.6 73.85

Line 6: The larger of 25.6 and 73.85 is 73.85

Line 7: The largest of 43.48, 34.00, and 12.65 is 43.48

NOTE

In the previous program, the function prototypes of the functions `larger` and `compareThree` appear before their function definitions. Therefore, the definition of the functions `larger` and `compareThree` can appear in any order.

Value-Returning Functions: Some Peculiarity

A value-returning function must return a value. Consider the following function, `secret`, that takes as a parameter an `int` value. If the value of the parameter, `x`, is greater than 5, it returns twice the value of `x`; otherwise, the value of `x` remains unchanged.

```

int secret(int x)
{
    if (x > 5)           //Line 1
        return 2 * x;    //Line 2
}

```

Because this is a value-returning function of type `int`, it must return a value of type `int`. Suppose the value of `x` is 10. Then the expression `x > 5` in Line 1 evaluates to `true`. So the `return` statement in Line 2 returns the value 20. Now suppose that `x` is 3. The

expression `x > 5` in Line 1 now evaluates to `false`. The `if` statement therefore fails, and the `return` statement in Line 2 *does not* execute. However, there are no more statements to be executed in the body of the function. In this case, the function returns a strange value. It thus follows that if the value of `x` is less than or equal to 5, the function does not contain any valid `return` statements to return the value of `x`.

A correct definition of the function `secret` is:

```
int secret(int x)
{
    if (x > 5)           //Line 1
        return 2 * x;    //Line 2

    return x;           //Line 3
}
```

Here, if the value of `x` is less than or equal to 5, the `return` statement in Line 3 executes, which returns the value of `x`. On the other hand, if the value of `x` is, say 10, the `return` statement in Line 2 executes, which returns the value 20 and also terminates the function.

Recall that in a value-returning function, the `return` statement returns the value. Consider the following `return` statement:

```
return x, y; //only the value of y will be returned
```

This is a legal `return` statement. You might think that this `return` statement is returning the values of `x` and `y`. However, this is not the case. Remember, a `return` statement returns only one value, even if the `return` statement contains more than one expression. If a `return` statement contains more than one expression, *only the value of the last expression is returned*. Therefore, in the case of the above `return` statement, the value of `y` is returned. The following program further illustrates this concept.

```
// This program illustrates that a value-returning function
// returns only one value, even if the return statement
// contains more than one expression.

#include <iostream>

using namespace std;

int funcRet1();
int funcRet2(int z);

int main()
{
    int num = 4;

    cout << "Line 1: The value returned by funcRet1: "
          << funcRet1() << endl;           // Line 1
```



```

        cout << "Line 2: The value returned by funcRet2: "
              << funcRet2(num) << endl;           // Line 2

    return 0;
}

int funcRet1()
{
    int x = 45;

    return 23, x; //only the value of x is returned
}

int funcRet2(int z)
{
    int a = 2;
    int b = 3;

    return 2 * a + b, z + b; //only the value of z + b is returned
}

```

Sample Run:

Line 1: The value returned by funcRet1: 45
 Line 2: The value returned by funcRet2: 7

Even though a **return** statement can contain more than one expression, a return statement in your program should contain only one expression. Having more than one expression in a **return** statement may result in redundancy, wasted code, and a confusing syntax.

More Examples of Value-Returning Functions

EXAMPLE 6-3

In this example, we write the definition of function **courseGrade**. This function takes as a parameter an **int** value specifying the score for a course and returns the grade, a value of type **char**, for the course. (We assume that the test score is a value between 0 and 100 inclusive.)

```

char courseGrade(int score)
{
    switch (score / 10)
    {
        case 0:
        case 1:
        case 2:
        case 3:

```

```

    case 4:
    case 5:
        return 'F';
    case 6:
        return 'D';
    case 7:
        return 'C';
    case 8:
        return 'B';
    case 9:
    case 10:
        return 'A';
    }
}

```

You can also write an equivalent definition of the function `courseGrade` that uses an `if...else` structure to determine the course grade.

EXAMPLE 6-4 (ROLLING A PAIR OF DICE)

In this example, we write a function that rolls a pair of dice until the sum of the numbers rolled is a specific number. We also want to know the number of times the dice are rolled to get the desired sum.

The smallest number on each die is 1, and the largest number is 6. So the smallest sum of the numbers rolled is 2, and the largest sum of the numbers rolled is 12. Suppose that we have the following declarations.

```

int die1;
int die2;
int sum;
int rollCount = 0;

```

We use the random number generator, discussed in Chapter 5, to randomly generate a number between 1 and 6. Then, the following statement randomly generates a number between 1 and 6 and stores that number into `die1`, which becomes the number rolled by `die1`.

```
die1 = rand() % 6 + 1;
```

Similarly, the following statement randomly generates a number between 1 and 6 and stores that number into `die2`, which becomes the number rolled by `die2`.

```
die2 = rand() % 6 + 1;
```

The sum of the numbers rolled by two dice is:

```
sum = die1 + die2;
```

Next, we determine whether `sum` contains the desired sum of the numbers rolled by the dice. If `sum` does not contain the desired sum, then we roll the dice again. This can be accomplished by the following `do...while` loop. (Assume that the `int` variable `num` contains the desired sum to be rolled.)

```
do
{
    die1 = rand() % 6 + 1;
    die2 = rand() % 6 + 1;
    sum = die1 + die2;
    rollCount++;
}
while (sum != num);
```

We can now write the function `rollDice` that takes as a parameter the desired sum of the numbers to be rolled and returns the number of times the dice are rolled to roll the desired sum.

```
int rollDice(int num)
{
    int die1;
    int die2;
    int sum;
    int rollCount = 0;

    srand(time(0));

    do
    {
        die1 = rand() % 6 + 1;
        die2 = rand() % 6 + 1;
        sum = die1 + die2;
        rollCount++;
    }
    while (sum != num);

    return rollCount;
}
```

The following program shows how to use the function `rollDice` in a program.

//Program: Roll dice

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int rollDice(int num);
```

```

int main()
{
    cout << "The number of times the dice are rolled to "
          << "get the sum 10 = " << rollDice(10) << endl;
    cout << "The number of times the dice are rolled to "
          << "get the sum 6 = " << rollDice(6) << endl;

    return 0;
}

int rollDice(int num)
{
    int die1;
    int die2;
    int sum;
    int rollCount = 0;

    srand(time(0));

    do
    {
        die1 = rand() % 6 + 1;
        die2 = rand() % 6 + 1;
        sum = die1 + die2;
        rollCount++;
    }
    while (sum != num);

    return rollCount;
}

```

Sample Run:

```

The number of times the dice are rolled to get the sum 10 = 11
The number of times the dice are rolled to get the sum 6 = 7

```

We leave it as an exercise for you to modify this program so that it allows the user to enter the desired sum of the numbers to be rolled. (See Programming Exercise 8 at the end of this chapter.)

Following is an example of a function that returns a Boolean value.

EXAMPLE 6-5 (PALINDROME NUMBER)

In this example, a function, `isNumPalindrome`, is designed that returns `true` if a nonnegative integer is a palindrome and returns `false` otherwise. A nonnegative integer

is a palindrome if it reads forward and backward in the same way. For example, the integers 5, 44, 434, 1881, and 789656987 are all palindromes.

Suppose `num` is a nonnegative integer. If `num < 10`, it is a palindrome, so the function should return `true`. Suppose `num >= 10`. To determine whether `num` is a palindrome, first compare the first and the last digits of `num`. If the first and the last digits of `num` are not the same, it is not a palindrome, so the function should return `false`. If the first and the last digits of `num` are the same, remove the first and last digits of `num` and repeat this process on the new number, which is obtained from `num` after removing the first and last digits of `num`. Repeat this process as long as the number is `>= 10`.

For example, suppose that the input is 18281. Because the first and last digits of 18281 are the same, remove the first and last digits to get the number 828. Repeat this process of comparing the first and last digits on 828. Once again, the first and last digits are the same. After removing the first and last digits of 828, the resulting number is 2, which is less than 10. Thus, 18281 is a palindrome.

To remove the first and last digits of `num`, you first need to find the highest power of 10 that divides `num` and call it `pwr`. The highest power of 10 that divides 18281 is 4, that is, `pwr = 4`. Now `18281 % 10pwr = 8281`, so the first digit is removed. Also, because `8281 / 10 = 828`, the last digit is removed. Therefore, to remove the first digit, you can use the mod operator, in which the divisor is `10pwr`. To remove the last digit, divide the number by 10. You then decrement `pwr` by 2 for the next iteration. The following algorithm implements this discussion:

1. If `num < 10`, it is a palindrome, so the function should return `true`.
2. Suppose `num` is an integer and `num >= 10`. To see if `num` is a palindrome:
 - a. Find the highest power of 10 that divides `num` and call it `pwr`. For example, the highest power of 10 that divides 434 is 2; the highest power of 10 that divides 789656987 is 8.
 - b. While `num` is greater than or equal to 10, compare the first and last digits of `num`.
 - b.1. If the first and last digits of `num` are not the same, `num` is not a palindrome. Return `false`.
 - b.2. If the first and the last digits of `num` are the same:
 - b.2.1. Remove the first and last digits of `num`.
 - b.2.2. Decrement `pwr` by 2.
 - c. Return `true`.

The following function implements this algorithm:

```
bool isNumPalindrome(int num)
{
    int pwr = 0;

    if (num < 10)                                //Step 1
        return true;
    else                                          //Step 2
    {
        while (num / static_cast<int>(pow(10.0, pwr)) >= 10) //Step 2.a
            pwr++;
        while (num >= 10)                          //Step 2.b
        {
            int tenTopwr = static_cast<int>(pow(10.0, pwr));

            if ((num / tenTopwr) != (num % 10))      //Step 2.b.1
                return false;
            else                                     //Step 2.b.2
            {
                num = num % tenTopwr;                //Step 2.b.2.1
                num = num / 10;                       //Step 2.b.2.1
                pwr = pwr - 2;                         //Step 2.b.2.2
            }
        } //end while
        return true;
    } //end else
}
```

NOTE

In the definition of the function `isNumPalindrome`, the function `pow` from the header file `cmath` is used to find the highest power of 10 that divides the number. Therefore, make sure to include the header file `cmath` in your program.

Flow of Execution

As stated earlier, a C++ program is a collection of functions. Recall that functions can appear in any order. The only thing that you have to remember is that you must declare an identifier before you can use it. The program is compiled by the compiler sequentially from beginning to end. Thus, if the function `main` appears before any other user-defined functions, it is compiled first. However, if `main` appears at the end (or middle) of the program, all functions whose definitions (not prototypes) appear before the function `main` are compiled before the function `main`, in the order they are placed.

Function prototypes appear before any function definition, so the compiler translates these first. The compiler can then correctly translate a function call. However, when the

program executes, the first statement in the function **main** always executes first, regardless of where in the program the function **main** is placed. Other functions execute only when they are called.

A function call statement transfers control to the first statement in the body of the function. In general, after the last statement of the called function executes, control is passed back to the point immediately following the function call. A value-returning function returns a value. Therefore, after executing the value-returning function, when the control goes back to the caller, the value that the function returns replaces the function call statement. The execution continues at the point immediately following the function call.

Suppose that a program contains functions **funcA** and **funcB**, and **funcA** contains a statement that calls **funcB**. Suppose that the program calls **funcA**. When the statement that contains a call to **funcB** executes, **funcB** executes, and while **funcB** is executing, the execution of the current call of **funcA** is on hold until **funcB** is done.

PROGRAMMING EXAMPLE: Largest Number

In this programming example, the function **larger** is used to determine the largest number from a set of numbers. For the purpose of illustration, this program determines the largest number from a set of 10 numbers. You can easily enhance this program to accommodate any set of numbers.

Input A set of 10 numbers.

Output The largest of 10 numbers.

Suppose that the input data is:

15 20 7 8 28 21 43 12 35 3

Read the first number of the data set. Because this is the only number read to this point, you may assume that it is the largest number so far and call it **max**. Read the second number and call it **num**. Now compare **max** and **num** and store the larger number into **max**. Now **max** contains the larger of the first two numbers. Read the third number. Compare it with **max** and store the larger number into **max**. At this point, **max** contains the largest of the first three numbers. Read the next number, compare it with **max**, and store the larger into **max**. Repeat this process for each remaining number in the data set. Eventually, **max** will contain the largest number in the data set. This discussion translates into the following algorithm:

1. Read the first number. Because this is the only number that you have read so far, it is the largest number so far. Save it in a variable called **max**.

2. For each remaining number in the list:
 - a. Read the next number. Store it in a variable called `num`.
 - b. Compare `num` and `max`. If `max < num`, then `num` is the new largest number, so update the value of `max` by copying `num` into `max`. If `max >= num`, discard `num`; that is, do nothing.
3. Because `max` now contains the largest number, print it.

To find the larger of two numbers, the program uses the function `larger`.

COMPLETE PROGRAM LISTING

```

//*****
// Author: D.S. Malik
//
// This program finds the largest number of a set of 10
// numbers.
//*****

#include <iostream>

using namespace std;

double larger(double x, double y);

int main()
{
    double num; //variable to hold the current number
    double max; //variable to hold the larger number
    int count; //loop control variable

    cout << "Enter 10 numbers." << endl;
    cin >> num; //Step 1
    max = num; //Step 1

    for (count = 1; count < 10; count++) //Step 2
    {
        cin >> num; //Step 2a
        max = larger(max, num); //Step 2b
    }

    cout << "The largest number is " << max
         << endl; //Step 3

    return 0;
} //end main

```



```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter 10 numbers.
10 56 73 42 22 67 88 26 62 11
The largest number is 88
```

PROGRAMMING EXAMPLE: Cable Company

Chapter 4 contains a program to calculate the bill for a cable company. In that program, all of the programming instructions are packed in the function `main`. Here, we rewrite the same program using user-defined functions, further illustrating structured programming. The problem analysis phase shows how to divide a complex problem into smaller subproblems. It also shows that while solving a particular subproblem, you can focus on only that part of the problem.

Input to and output of the program are the same as before.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Because there are two types of customers, residential and business, the program contains two separate functions: one to calculate the bill for residential customers and one to calculate the bill for business customers. Both functions calculate the billing amount and then return the billing amount to the function `main`. The function `main` prints the amount due. Let us call the function that calculates the residential bill `residential` and the function that calculates the business bill `business`. The formulas to calculate the bills are the same as before.

As in Chapter 4, data such as the residential bill processing fee, the cost of residential basic service connection, and so on are special. Therefore, these are declared as named constants.

Function `residential`

To compute the residential bill, you need to know the number of premium channels to which the customer subscribes. Based on the number of premium channels, you can calculate the billing amount. After calculating the billing amount, the function returns the billing amount using the `return` statement. The following four steps describe this function:

- a. Prompt the user for the number of premium channels.
- b. Read the number of premium channels.

- c. Calculate the bill.
- d. Return the amount due.

This function contains a statement to prompt the user to enter the number of premium channels (Step a) and a statement to read the number of premium channels (Step b). Other items needed to calculate the billing amount, such as the cost of basic service connection and bill-processing fees, are defined as named constants (before the definition of the function `main`). Therefore, to calculate the billing amount, this function does not need to get any value from the function `main`. This function, therefore, has no parameters.

Local Variables (Function residential) From the previous discussion, it follows that the function `residential` requires variables to store both the number of premium channels and the billing amount. This function needs only two local variables to calculate the billing amount:

```
int noOfPChannels; //number of premium channels
double bAmount;   //billing amount
```

The definition of the function `residential` can now be written as follows:

```
double residential()
{
    int noOfPChannels; //number of premium channels
    double bAmount;    //billing amount
    cout << "Enter the number of premium "
          << "channels used: ";
    cin >> noOfPChannels;
    cout << endl;

    bAmount = RES_BILL_PROC_FEES +
              RES_BASIC_SERV_COST +
              noOfPChannels * RES_COST_PREM_CHANNEL;

    return bAmount;
}
```

Function business To compute the business bill, you need to know the number of both the basic service connections and the premium channels to which the customer subscribes. Then, based on these numbers, you can calculate the billing amount. The billing amount is then returned using the `return` statement. The following six steps describe this function:

- a. Prompt the user for the number of basic service connections.
- b. Read the number of basic service connections.
- c. Prompt the user for the number of premium channels.
- d. Read the number of premium channels.
- e. Calculate the bill.
- f. Return the amount due.

This function contains the statements to prompt the user to enter the number of basic service connections and premium channels (Steps a and c). The function also contains statements to input the number of basic service connections and premium channels (Steps b and d). Other items needed to calculate the billing amount, such as the cost of basic service connections and bill-processing fees, are defined as named constants (before the definition of the function `main`). It follows that to calculate the billing amount, this function does not need to get any values from the function `main`. Therefore, it has no parameters.

Local Variables (Function business) From the preceding discussion, it follows that the function `business` requires variables to store the number of basic service connections and premium channels, as well as the billing amount. In fact, this function needs only three local variables to calculate the billing amount:

```
int noOfBasicServiceConnections;
int noOfPChannels;           //number of premium channels
double bAmount;              //billing amount
```

The definition of the function `business` can now be written as follows:

```
double business()
{
    int noOfBasicServiceConnections;
    int noOfPChannels;           //number of premium channels
    double bAmount;              //billing amount
    cout << "Enter the number of basic "
          << "service connections: ";
    cin >> noOfBasicServiceConnections;
    cout << endl;

    cout << "Enter the number of premium "
          << "channels used: ";
    cin >> noOfPChannels;
    cout << endl;

    if (noOfBasicServiceConnections <= 10)
        bAmount = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST +
                  noOfPChannels * BUS_COST_PREM_CHANNEL;
    else
        bAmount = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST +
                  (noOfBasicServiceConnections - 10) *
                  BUS_BASIC_CONN_COST +
                  noOfPChannels * BUS_COST_PREM_CHANNEL;

    return bAmount;
}
```

MAIN
ALGORITHM
(Function
main)

1. To output floating-point numbers in a fixed decimal format with the decimal point and trailing zeros, set the manipulators **fixed** and **showpoint**.
2. To output floating-point numbers to two decimal places, set the precision to two decimal places.
3. Prompt the user for the account number.
4. Get the account number.
5. Prompt the user to enter the customer type.
6. Get the customer type.
7.
 - a. If the customer type is R or r:
 - i. Call the function **residential** to calculate the bill.
 - ii. Print the bill.
 - b. If the customer type is B or b:
 - i. Call the function **business** to calculate the bill.
 - ii. Print the bill.
 - c. If the customer type is other than R, r, B, or b, it is an invalid customer type.

COMPLETE PROGRAM LISTING

```

//*****
// Author: D. S. Malik
//
// Program: Cable Company Billing
// This program calculates and prints a customer's bill for
// a local cable company. The program processes two types of
// customers: residential and business.
//*****

#include <iostream>
#include <iomanip>
using namespace std;

    //Named constants - residential customers
const double RES_BILL_PROC_FEES = 4.50;
const double RES_BASIC_SERV_COST = 20.50;
const double RES_COST_PREM_CHANNEL = 7.50;

    //Named constants - business customers
const double BUS_BILL_PROC_FEES = 15.00;
const double BUS_BASIC_SERV_COST = 75.00;
const double BUS_BASIC_CONN_COST = 5.00;
const double BUS_COST_PREM_CHANNEL = 50.00;

```

```

double residential();      //Function prototype
double business();        //Function prototype

int main()
{
    //declare variables
    int accountNumber;
    char customerType;
    double amountDue;

    cout << fixed << showpoint;           //Step 1
    cout << setprecision(2);              //Step 2

    cout << "This program computes a cable bill."
         << endl;
    cout << "Enter account number: ";      //Step 3
    cin >> accountNumber;                  //Step 4
    cout << endl;

    cout << "Enter customer type: R, r "
         << "(Residential), B, b (Business): "; //Step 5
    cin >> customerType;                  //Step 6
    cout << endl;

    switch (customerType)                 //Step 7
    {
        case 'r':                        //Step 7a
        case 'R':
            amountDue = residential();    //Step 7a.i
            cout << "Account number = "
                 << accountNumber << endl; //Step 7a.ii
            cout << "Amount due = $"
                 << amountDue << endl;    //Step 7a.ii
            break;
        case 'b':                        //Step 7b
        case 'B':
            amountDue = business();        //Step 7b.i
            cout << "Account number = "
                 << accountNumber << endl; //Step 7b.ii
            cout << "Amount due = $"
                 << amountDue << endl;    //Step 7b.ii
            break;
        default:
            cout << "Invalid customer type."
                 << endl;                 //Step 7c
    }

    return 0;
}

```

```

double residential()
{
    int noOfPChannels;    //number of premium channels
    double bAmount;      //billing amount

    cout << "Enter the number of premium "
          << "channels used: ";
    cin >> noOfPChannels;
    cout << endl;

    bAmount = RES_BILL_PROC_FEES +
              RES_BASIC_SERV_COST +
              noOfPChannels * RES_COST_PREM_CHANNEL;

    return bAmount;
}

double business()
{
    int noOfBasicServiceConnections;
    int noOfPChannels;    //number of premium channels
    double bAmount;      //billing amount

    cout << "Enter the number of basic "
          << "service connections: ";
    cin >> noOfBasicServiceConnections;
    cout << endl;
    cout << "Enter the number of premium "
          << "channels used: ";
    cin >> noOfPChannels;
    cout << endl;

    if (noOfBasicServiceConnections <= 10)
        bAmount = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST +
                  noOfPChannels * BUS_COST_PREM_CHANNEL;
    else
        bAmount = BUS_BILL_PROC_FEES + BUS_BASIC_SERV_COST +
                  (noOfBasicServiceConnections - 10) *
                  BUS_BASIC_CONN_COST +
                  noOfPChannels * BUS_COST_PREM_CHANNEL;

    return bAmount;
}

```

Sample Run: In this sample run, the user input is shaded.

This program computes a cable bill.

Enter account number: 21341

Enter customer type: R, r (Residential), B, b (Business): B

Enter the number of basic service connections: 25

Enter the number of premium channels used: 9

Account number = 21341

Amount due = \$615.00

QUICK REVIEW

1. Functions are like miniature programs and are called modules.
2. Functions enable you to divide a program into manageable tasks.
3. The C++ system provides the standard (predefined) functions.
4. To use a standard function, you must:
 - i. Know the name of the header file that contains the function's specification,
 - ii. Include that header file in the program, and
 - iii. Know the name and type of the function and number and types of the parameters (arguments).
5. There are two types of user-defined functions: value-returning functions and void functions.
6. Variables defined in a function heading are called formal parameters.
7. Expressions, variables, or constant values used in a function call are called actual parameters.
8. In a function call, the number of actual parameters and their types must match with the formal parameters in the order given.
9. To call a function, use its name together with the actual parameter list.
10. A value-returning function returns a value. Therefore, a value-returning function is used (called) in either an expression or an output statement or as a parameter in a function call.
11. The general syntax of a user-defined function is:


```
functionType functionName(formal parameter list)
{
    statements
}
```
12. The line `functionType functionName(formal parameter list)` is called the function heading (or function header). Statements enclosed between braces (`{` and `}`) are called the body of the function.
13. The function heading and the body of the function are called the definition of the function.

14. If a function has no parameters, you still need the empty parentheses in both the function heading and the function call.
15. A value-returning function returns its value via the `return` statement.
16. A function can have more than one `return` statement. However, whenever a `return` statement executes in a function, the remaining statements are skipped and the function exits.
17. A `return` statement returns only one value.
18. A function prototype is the function heading without the body of the function; the function prototype ends with the semicolon.
19. A function prototype announces the function type, as well as the type and number of parameters, used in the function.
20. In a function prototype, the names of the variables in the formal parameter list are optional.
21. Function prototypes help the compiler correctly translate each function call.
22. In a program, function prototypes are placed before every function definition, including the definition of the function `main`.
23. When you use function prototypes, user-defined functions can appear in any order in the program.
24. When the program executes, the execution always begins with the first statement in the function `main`.
25. User-defined functions execute only when they are called.
26. A call to a function transfers control from the caller to the called function.
27. In a function call statement, you specify only the actual parameters, not their data type or the function type.
28. When a function exits, the control goes back to the caller.

EXERCISES

1. Mark the following statements as true or false.
 - a. To use a predefined function in a program, you need to know only the name of the function and how to use it.
 - b. A value-returning function returns only one value.
 - c. Parameters allow you to use different values each time the function is called.
 - d. When a `return` statement executes in a user-defined function, the function immediately exits.
 - e. A value-returning function returns only integer values.

2. Determine the value of each of the following expressions.
 - a. `static_cast<char>(toupper('b'))`
 - b. `static_cast<char>(toupper('7'))`
 - c. `static_cast<char>(toupper('K'))`
 - d. `static_cast<char>(toupper('*'))`
 - e. `static_cast<char>(tolower('D'))`
 - f. `static_cast<char>(tolower('8'))`
 - g. `static_cast<char>(tolower('h'))`
 - h. `static_cast<char>(tolower('$'))`
3. Determine the value of each of the following expressions.
 - a. `abs(-4)` b. `fabs(10.8)` c. `fabs(-2.5)` d. `pow(3.2, 2)`
 - e. `pow(2.5, 3)` f. `sqrt(25.0)` g. `sqrt(6.25)`
 - h. `pow(3.0, 4.0) / abs(-9)` i. `floor(28.95)` j. `ceil(35.2)`
4. Using the functions described in Table 6-1, write each of the following as a C++ expression. (The expression in (e) denotes the absolute value of $x + 2.5$.)
 - a. $2.0^{5.2}$ b. $\sqrt{x+y}$ c. $u^v - 3$ d. $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$ e. $|x + 2.5|$
5. Consider the following function definition:

```
double func(double x, int y, string name)
{
    //function body
}
```

Which of the following is the correct function prototype of the function `func`?

- i. `double func();`
 - ii. `double func(double, int, string);`
 - iii. `double func(double x, int y, string name)`
 - iv. `func(double x, int y, string name);`
6. Consider the following program:

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int num1;
    int num2;

    cout << "Enter two integers: ";
    cin >> num1 >> num2;
    cout << endl;
```

```

    if (num1 != 0 && num2 != 0)
        cout << sqrt(fabs(num1 + num2 + 0.0)) << endl;
    else if (num1 != 0)
        cout << floor(num1 + 0.0) << endl;
    else if (num2 != 0)
        cout << ceil(num2 + 0.0) << endl;
    else
        cout << 0 << endl;

    return 0;
}

```

- a. What is the output if the input is 12 4?
 - b. What is the output if the input is 3 27?
 - c. What is the output if the input is 25 0?
 - d. What is the output if the input is 0 49?
7. Consider the following statements:

```

double num1, num2, num3;
int int1, int2, int3;
int value;

```

```

num1 = 5.0; num2 = 6.0; num3 = 3.0;
int1 = 4; int2 = 7; int3 = 8;

```

and the function prototype:

```
double cube(double a, double b, double c);
```

Which of the following statements are valid? If they are invalid, explain why.

- a. `value = cube (num1, 15.0, num3);`
 - b. `cout << cube(num1, num3, num2) << endl;`
 - c. `cout << cube(6.0, 8.0, 10.5) << endl;`
 - d. `cout << cube(num1, num3) << endl;`
 - e. `value = cube(num1, int2, num3);`
 - f. `value = cube(7, 8, 9);`
 - g. `value = cube(int1, int2, int3);`
8. Consider the following functions:

```

int secret(int x)
{
    int i, j;

    i = 2 * x;

    if (i > 10)
        j = x / 2;
}

```

```

        else
            j = x / 3;

        return j - 1;
    }

    int another(int a, int b)
    {
        int i, j;

        j = 0;

        for (i = a; i <= b; i++)
            j = j + i;

        return j;
    }

```

What is the output of each of the following program segments? Assume that `x`, `y`, and `k` are `int` variables.

- a. `x = 10;`
`cout << secret(x) << endl;`
 - b. `x = 5; y = 8;`
`cout << another(x, y) << endl;`
 - c. `x = 10; k = secret(x);`
`cout << x << " " << k << " " << another(x, k) << endl;`
 - d. `x = 5; y = 8;`
`cout << another(y, x) << endl;`
9. Consider the following function prototypes:
- ```

int test(int, char, double, int);
double two(double, double);
char three(int, int, char, double);

```

Answer the following questions.

- a. How many parameters does the function `test` have? What is the type of the function `test`?
- b. How many parameters does function `two` have? What is the type of function `two`?
- c. How many parameters does function `three` have? What is the type of function `three`?
- d. How many actual parameters are needed to call the function `test`? What is the type of each actual parameter, and in what order should you use these parameters in a call to the function `test`?
- e. Write a C++ statement that prints the value returned by the function `test` with the actual parameters 5, 5, 7.3, and 'z'.

- f. Write a C++ statement that prints the value returned by function `two` with the actual parameters 17.5 and 18.3, respectively.
- g. Write a C++ statement that prints the next character returned by function `three`. (Use your own actual parameters.)
- 10. Why do you need to include function prototypes in a program that contains user-defined functions?
- 11. Write the definition of a function that takes as input a `char` value and returns `true` if the character is uppercase; otherwise, it returns `false`.
- 12. Consider the following function:

```
int mystery(int x, double y, char ch)
{
 int u;
 if ('A' <= ch && ch <= 'R')
 return (2 * x + static_cast<int>(y));
 else
 return (static_cast<int>(2 * y) - x);
}
```

What is the output of the following C++ statements?

- a. `cout << mystery(5, 4.3, 'B') << endl;`
- b. `cout << mystery(4, 9.7, 'v') << endl;`
- c. `cout << 2 * mystery(6, 3.9, 'D') << endl;`
- 13. Consider the following function:

```
int secret(int one)
{
 int i;
 int prod = 1;

 for (i = 1; i <= 3; i++)
 prod = prod * one;
 return prod;
}
```

- a. What is the output of the following C++ statements?
  - i. `cout << secret(5) << endl;`
  - ii. `cout << 2 * secret(6) << endl;`
- b. What does the function `secret` do?
- 14. Write the definition of a function that takes as input the three numbers. The function returns `true` if the first number to the power of the second number equals the third number; otherwise, it returns `false`. (Assume that the three numbers are of type `double`.)

15. What is the output of the following C++ program?

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
 int counter;

 for (counter = 1; counter <= 100; counter++)
 if (pow(floor(sqrt(counter + 0.0)), 2) == counter)
 cout << counter << " ";

 cout << endl;

 return 0;
}
```

16. What is the output of the following program?

```
#include <iostream>

using namespace std;

int mystery(int x, int y, int z);

int main()
{
 cout << mystery(7, 8, 3) << endl;
 cout << mystery(10, 5, 30) << endl;
 cout << mystery(9, 12, 11) << endl;
 cout << mystery(5, 5, 8) << endl;
 cout << mystery(10, 10, 10) << endl;

 return 0;
}

int mystery(int x, int y, int z)
{
 if (x <= y && x <= z)
 return (y + z - x);
 else if (y <= z && y <= x)
 return (z + x - y);
 else
 return (x + y - z);
}
```

17. Write the definition of a function that takes as input three numbers and returns the sum of the first two numbers multiplied by the third number. (Assume that the three numbers are of type `double`.)

18. Show the output of the following program:

```
#include <iostream>

using namespace std;

int mystery(int);

int main()
{
 int n;

 for (n = 1; n <= 5; n++)
 cout << mystery(n) << endl;

 return 0;
}

int mystery(int k)
{
 int x, y;

 y = k;

 for (x = 1; x <= (k - 1); x++)
 y = y * (k - x);

 return y;
}
```

## PROGRAMMING EXERCISES

---

1. Write a program that uses the function `isNumPalindrome` given in Example 6-5 (Palindrome Number). Test your program on the following numbers: 10, 34, 22, 333, 678, 67876, 44444, and 123454321.
2. Write a value-returning function, `isVowel`, that returns the value `true` if a given character is a vowel and otherwise returns `false`.
3. Write a program that prompts the user to input a sequence of characters and outputs the number of vowels. (Use the function `isVowel` written in Programming Exercise 2.)
4. Write a program that defines the named constant `PI`, `const double PI = 3.1419;`, which stores the value of  $\pi$ . The program should use `PI` and the functions listed in Table 6-1 to accomplish the following.
  - a. Output the value of  $\sqrt{\pi}$ .
  - b. Prompt the user to input the value of a `double` variable `r`, which stores the radius of a sphere. The program then outputs the following:
    - i. The value of  $4\pi r^2$ , which is the surface area of the sphere.
    - ii. The value of  $(4/3)\pi r^3$ , which is the volume of the sphere.

5. The following program is designed to find the area of a rectangle, the area of a circle, or the volume of a cylinder. However, (a) the statements are in the incorrect order; (b) the function calls are incorrect; (c) the logical expression in the `while` loop is incorrect; and (d) function definitions are incorrect. Rewrite the program so that it works correctly. Your program must be properly indented. (Note that the program is menu driven and allows the user to run the program as long as the user wishes.)

```
#include <iostream>

using namespace std;

const double PI = 3.1419;

double rectangle(double l, double w);

#include <iomanip>

int main()
{
 double radius;
 double height;

 cout << fixed << showpoint << setprecision(2) << endl;
 cout << "This program can calculate the area of a rectangle, "
 << "the area of a circle, or volume of a cylinder." << endl;
 cout << "To run the program enter: " << endl;
 cout << "1: To find the area of rectangle." << endl;
 cout << "2: To find the area of a circle." << endl;
 cout << "3: To find the volume of a cylinder." << endl;
 cout << "-1: To terminate the program." << endl;
 cin >> choice;
 cout << endl;

 int choice;

 while (choice != -1)
 {
 {
 case 1:
 cout << "Enter the radius of the base and the "
 << "height of the cylinder: ";
 cin >> radius >> height;
 cout << endl;

 cout << "Area = " << circle(length, height) << endl;
 break;

 case 3:
 double length, width;
 cout << "Enter the radius of the circle: ";
 cin >> radius;
 cout << endl;
```

```

 cout << "Area = " << rectangle(radius)
 << endl;
 break;

 case 2:
 cout << "Enter the length and the width "
 << "of the rectangle: ";
 cin >> length >> width;
 cout << endl;

 cout << "Volume = " << cylinder(radius, height)
 << endl;
 break;
 default:
 cout << "Invalid choice!" << endl;
 }
 switch (choice)
 }

double circle(double r)
double cylinder(double bR, double h);

cout << "To run the program enter: " << endl;
cout << "2: To find the area of a circle." << endl;
cout << "1: To find the area of rectangle." << endl;
cout << "3: To find the colume of a cylinder." << endl;
cout << "-1: To terminate the program." << endl;
cin >> choice;
cout << endl;

return 0;
}

double rectangle(double l, double w)
{
 return l * r;
}

double circle(double r)
{
 return PI * r * w;
}

double cylinder(double bR, double h)
{
 return PI * bR * bR * l;
}

```

6. Write a function, `reverseDigit`, that takes an integer as a parameter and returns the number with its digits reversed. For example, the value of `reverseDigit(12345)` is 54321; the value of `reverseDigit(5600)` is 65; the value of `reverseDigit(7008)` is 8007; and the value of `reverseDigit(-532)` is -235.



7. Modify the roll dice program, Example 6-4, so that it allows the user to enter the desired sum of the numbers to be rolled. Also allow the user to call the `rollDice` function as many times as the user desires.
8. The following formula gives the distance between two points,  $(x_1, y_1)$  and  $(x_2, y_2)$  in the Cartesian plane:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Given the center and a point on the circle, you can use this formula to find the radius of the circle. Write a program that prompts the user to enter the center and a point on the circle. The program should then output the circle's radius, diameter, circumference, and area. Your program must have at least the following functions:

- a. **distance**: This function takes as its parameters four numbers that represent two points in the plane and returns the distance between them.
- b. **radius**: This function takes as its parameters four numbers that represent the center and a point on the circle, calls the function **distance** to find the radius of the circle, and returns the circle's radius.
- c. **circumference**: This function takes as its parameter a number that represents the radius of the circle and returns the circle's circumference. (If  $r$  is the radius, the circumference is  $2\pi r$ .)
- d. **area**: This function takes as its parameter a number that represents the radius of the circle and returns the circle's area. (If  $r$  is the radius, the area is  $\pi r^2$ .)

Assume that  $\pi = 3.1416$ .

9. Rewrite the program in Programming Exercise 15 of Chapter 4 (cell phone company) so that it uses the following functions to calculate the billing amount. (In this programming exercise, do not output the number of minutes during which the service is used.)
  - a. **regularBill**: This function calculates and returns the billing amount for regular service.
  - b. **premiumBill**: This function calculates and returns the billing amount for premium service.
10. Write a program that takes as input five numbers and outputs the mean (average) and standard deviation of the numbers. If the numbers are  $x_1, x_2, x_3, x_4$ , and  $x_5$ , then the mean is  $x = (x_1 + x_2 + x_3 + x_4 + x_5)/5$  and the standard deviation is:

$$s = \sqrt{\frac{(x_1 - x)^2 + (x_2 - x)^2 + (x_3 - x)^2 + (x_4 - x)^2 + (x_5 - x)^2}{5}}$$

Your program must contain at least the following functions: a function that calculates and returns the mean and a function that calculates the standard deviation.

11. When you borrow money to buy a house, a car, or for some other purposes, then you typically repay it by making periodic payments. Suppose that the loan amount is  $L$ ,  $r$  is the interest rate per year,  $m$  is the number of payments in a year, and the loan is for  $t$  years. Suppose that  $i = (r / m)$  and  $r$  is in decimal. Then the periodic payment is:

$$R = \frac{Li}{1 - (1 + i)^{-mt}},$$

You can also calculate the unpaid loan balance after making certain payments. For example, the unpaid balance after making  $k$  payments is:

$$L' = R \left[ \frac{1 - (1 + i)^{-(mt-k)}}{i} \right],$$

where  $R$  is the periodic payment. (Note that if the payments are monthly, then  $m = 12$ .)

Write a program that prompts the user to input the values of  $L$ ,  $r$ ,  $m$ ,  $t$ , and  $k$ . The program then outputs the appropriate values. Your program must contain at least two functions, with appropriate parameters, to calculate the periodic payments and the unpaid balance after certain payments. Make the program menu driven and use a loop so that the user can repeat the program for different values.

12. During the tax season, every Friday, J&J accounting firm provides assistance to people who prepare their own tax returns. Their charges are as follows.
  - a. If a person has low income ( $\leq 25,000$ ) and the consulting time is less than or equal to 30 minutes, there are no charges; otherwise, the service charges are 40% of the regular hourly rate for the time over 30 minutes.
  - b. For others, if the consulting time is less than or equal to 20 minutes, there are no service charges; otherwise, service charges are 70% of the regular hourly rate for the time over 20 minutes.

(For example, suppose that a person has low income and spent 1 hour and 15 minutes, and the hourly rate is \$70.00. Then the billing amount is  $70.00 \times 0.40 \times (45 / 60) = \$21.00$ .)

Write a program that prompts the user to enter the hourly rate, the total consulting time, and whether the person has low income. The program should output the billing amount. Your program must contain a function that takes as input the hourly rate, the total consulting time, and a value indicating whether the person has low income. The function should return the billing amount. Your program may prompt the user to enter the consulting time in minutes.