Contents:

Lecture 1: AN OVERVIEW OF COMPUTERS AND PROGRAMMING LANGUAGES

TABLE OF CONTENTS

1- AN OVERVIEW OF COMPUTERS AND PROGRAMMING LANGUAGES

Lecture 1:

AN OVERVIEW OF COMPUTERS AND PROGRAMMING LANGUAGES

IN THIS Lecture, 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++.

1.1-Introduction

- 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.

1.2- 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.
- 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.

1.3- 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, 1GBRAM, 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.

1.3.1 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.

1.3.1.1 Central Processing Unit

- 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.

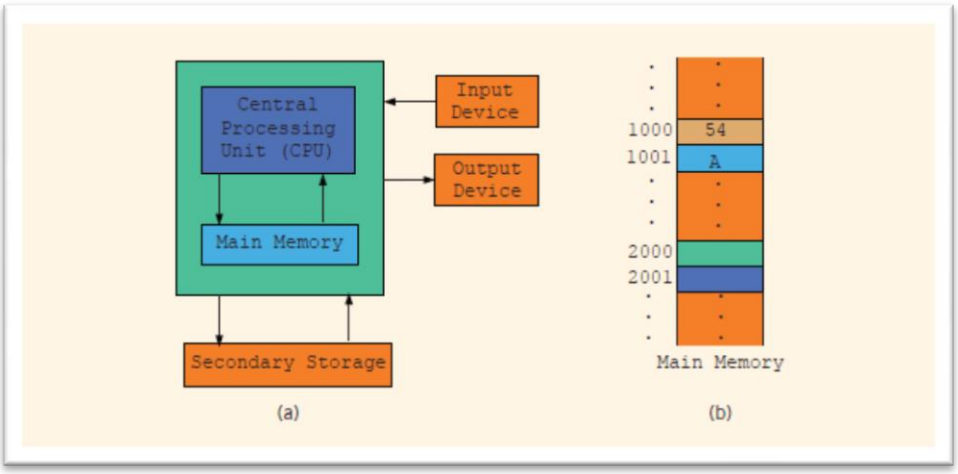

FIGURE 1-1 Hardware components of a computer and main memory

Figure 1-1(a) shows some hardware components.

1.3.1.2 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 lecture, main memory stores everything as sequences of 0s and 1s. The memory addresses are also expressed as sequences of 0s and 1s.

1.3.1.3 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.

1.3.1.4 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

1.3.2 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.

1.4 The Language of a Computer

- What is the language of the computer?
- 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.

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

TABLE 1-1 Binary Units

- 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. Furthermore, the binary number 1000001 is the binary representation of 65.
- The character 3 is encoded as 0110011. It also follows that 0110011 is the binary representation of 51.
- 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.
- 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.

1.5-The Evolution of Programming Languages

1.5.1 The machine language

- 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.

1.5.2 The assembly language

- 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.

1.5.3 High Level Languages

- 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 course.

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 highlevel languages into machine code.

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

1.6- 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.

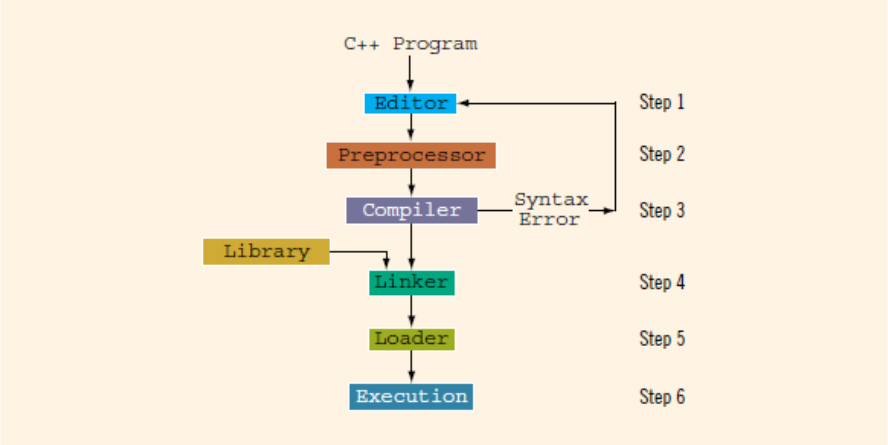

FIGURE 1-3 Processing a C++ program

- 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. 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.

1.7- 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.

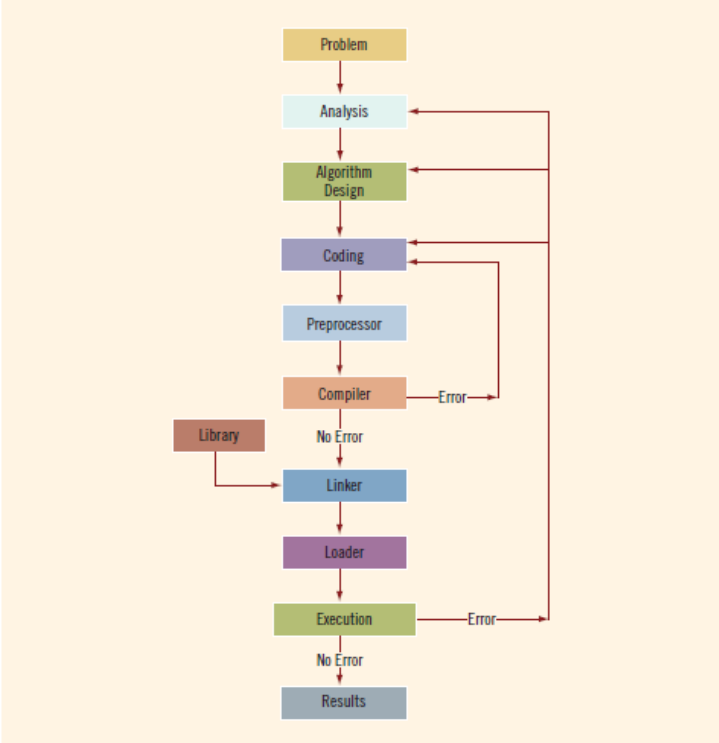

FIGURE 1-4 Problem analysis–coding–execution cycle

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

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.

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

3. Calculate bonus using the following formula:
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:

```
payCheck = baseSalary + bonus + 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 lecture 5, we use this algorithm to write a C++ program to play the guessing the number game.

EXAMPLE 1-5

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

1.8 Programming Methodologies

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

1.8.1 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.

1.8.2 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 lecture 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, you will learn the details of functions later in lecture 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 lectures 4 and 5. Also, in lecture 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 lecture Classes and Data Abstraction (later in the next course).
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 lectures (4 and 5) and user-defined functions into two lectures (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.

1.9 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 course focuses on the syntax of C++ as approved by ANSI/ISO, referred to as ANSI/ISO Standard C++.

QUICK REVIEW 1

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

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.)

Lecture 2: BASIC ELEMENTS OF C++

A C++ Program	
The Basics of a C++ Program	
Comments	
Special Symbols	
Reserved Words (Keywords)	
Identifiers	
Whitespaces	
Data Types	
Simple Data Types	
Floating-Point Data Types	
Arithmetic Operators and Operator Precedence	
Order of Precedence	
Expressions	
Mixed Expressions	
Type Conversion (Casting)	
string Type	
Input	
Allocating Memory with Constants and Variables	
Putting Data into Variables	
Assignment Statement	
Saving and Using the Value of an Expression	
Declaring and Initializing Variables	
Input (Read) Statement	
Variable Initialization	
Increment and Decrement Operators	
Output	
Preprocessor Directives	
namespace and Using cin and cout in a Program	
Using the string Data Type in a Program	
Creating a C++ Program	
Debugging: Understanding and Fixing Syntax Errors	
Program Style and Form	
Syntax	
Use of Blanks	
Use of Semicolons, Brackets, and Commas	
Semantics	
Naming Identifiers	
Prompt Lines	
Documentation	
Form and Style	
More on Assignment Statements	
Programming Example: Convert Length	
Programming Example: Make Change	
Quick Review	
Exercises	
Programming Exercises	

BASIC ELEMENTS OF C++

IN THIS Lecture, 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 lecture, 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.

2.1 A C++ Program

In this lecture, 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.

```
//*********************************************************
// 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 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 lecture, 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.
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.

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.

Note that in C++, << is an operator, called the stream insertion operator.

2.2 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.

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.

2.3 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.

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 lectures, 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.
*/
```

2.4 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. 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.

2.5 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. The following is a complete list of reserved words:

and	and_eq	asm	auto
bitand	bitor	bool	break
case	catch	char	class
compl	const	const_cast	continue
default	delete	do	double
dynamic_cast	else	enum	explicit
export	extern	false	float
for	friend	goto	if
include	inline	int	long
mutable	namespace	new	not
not_eq	operator	or	or_eq
private	protected	public	register
reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct

switch	template	this	throw
true	try	typedef	typeid
typename	union	unsigned	using
virtual	void	volatile	wchar_t
while	xor	xor_eq	

Throughout this lectures, reserved words are shown in blue.

2.6 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 lecture, 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.

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.

The following are legal identifiers in C++:

first
conversion
payRate
counter1

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

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.

TABLE 2-1 Examples of Illegal Identifiers

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 (_).

2.7 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 canb e used to make the program readable.

2.8 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

2.9 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 lecture 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

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

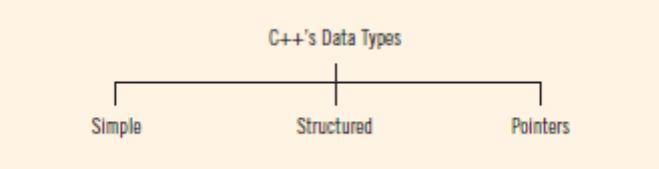

FIGURE 2-1 C++ data types

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 lectures 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. Use this table only as a guide. Different compilers may allow different ranges of values. Check your compiler's documentation.

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

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

2.9.1 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.

2.9.2 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. In C++, bool, true, and false are reserved words.

2.9.3 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.

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.

2.9. 4 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.
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.

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

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

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 -3.4E+38 and 3.4E+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 -1.7E+308 and 1.7E+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 (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.
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.
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 course mostly uses the data type double to manipulate floating point numbers.

2.10 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.

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

2.11 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.

2.12 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.

2.12.1 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 (Rule1(b)) = 12.8

4 + 5 / 2.0 = 4 + 2.5 = 6.5

5 / 2.0 = 5.0 / 2.0 (Rule1(b)) = 2.5

4 + 2.5 = 4.0 + 2.5 (Rule1(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.

2.13 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
static_cast<int>(7.9) 7
static_cast<int>(3.3) 3
static_cast<double>(25) 25.0
static_cast<double>(5 + 3) = static_cast<double>(8) = 8.0
static_cast<double>(15) / 2 = 15.0 / 2
(because static_cast<double> (15) = 15.0)
= 15.0 / 2.0 = 7.5
static_cast<double>(15 / 2) = static_cast<double>(7) (because 15/2=7)
= 7.0
static_cast<int>(7.8 +
static_cast<double>(15) / 2) = static_cast<int>(7.8 + 7.5)
= static_cast<int>(15.3)
= 15
static_cast<int>(7.8 +
static_cast<double>(15 / 2)) = static_cast<int>(7.8 + 7.0)
= static_cast<int>(14.8)
= 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.

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.

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.

2.14 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++.
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.

2.15 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.

2.15.1 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.

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, . . .;
```

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.

- 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.
- A data type is called simple if the variable or named constant of that type can store only one value at a time or if the data type cannot be divided into other simpler data type. For example, if x is an int variable, at a given time, only one value can be stored in x.

2.16 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.

2.16.1 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.

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 complier 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.

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

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

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

Note: Tracing values through a sequence, called a walk-through, is a valuable tool to learn and practice. Try it in the sequence above.

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.

2.16.1.1Saving 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.

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 + (b2 – 4ac) and –b –(b2 – 4ac) and assign the values of these expressions to x and y, respectively.

Because the expression b2 – 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.

2.16.1.2 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.

2.16.2 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. 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.

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:
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
}
```

Sample Run:
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.

- 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.

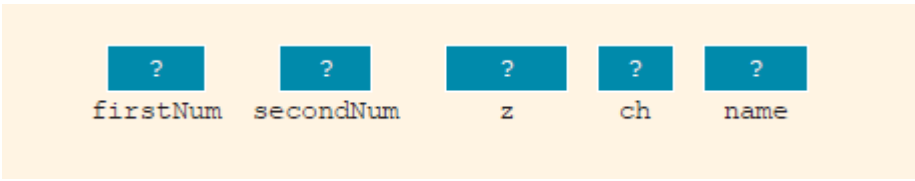

FIGURE 2-2 Variables before statement 1 executes

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

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.

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.

After St.	Values of the Variables					Explanation
1	4 firstNum	? secondNum	? z	? ch	? name	Store 4 into firstNum.
2	4 firstNum	14 secondNum	? z	? ch	? name	2 * firstNum + 6 = 2 * 4 + 6 = 14. Store 14 into secondNum.
3	4 firstNum	14 secondNum	2.5 z	? ch	? name	(firstNum + 1) / 2.0 = (4 + 1) / 2.0 = 5 / 2.0 = 2.5. Store 2.5 into z.
4	4 firstNum	14 secondNum	2.5 z	A ch	? name	Store 'A' into ch.
5	4 firstNum	8 secondNum	2.5 z	A ch	? name	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	4 firstNum	8 secondNum	16.3 z	A ch	? name	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	32 firstNum	8 secondNum	16.3 z	A ch	? name	2 * secondNum + static_cast<int>(z) = 2 * 8 + 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	32 firstNum	8 secondNum	16.3 z	A ch	Jenny name	Read the next input, Jenny, from the keyboard and store it into name.
9	32 firstNum	9 secondNum	16.3 z	A ch	Jenny name	secondNum + 1 = 8 + 1 = 9. Store 9 into secondNum.
10	32 firstNum	9 secondNum	16.3 z	D ch	Jenny name	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	100 (firstNum), 9 (secondNum), 16.3 (z), D (ch), Jenny (name)	firstNum + static_cast<int>(ch) = 32 + static_cast<int>('D') = 32 + 68 = 100. Store 100 into firstNum.
12	100 (firstNum), 9 (secondNum), 83.7 (z), D (ch), Jenny (name)	firstNum – z = 100 – 16.3 = 100.0 – 16.3 = 83.7. Store 83.7 into z.

2.17 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 and decrement 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:

Predecrement: --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.

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.

2.18 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.

- 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 el), which causes the insertion point to move to the beginning of the next line. 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. 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).

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

Statement	Output
1 cout << 29 / 4 << endl;	7
2 cout << "Hello there." << endl;	Hello there.
3 cout << 12 << endl;	12
4 cout << "4 + 7" << endl;	4 + 7
5 cout << 4 + 7 << endl;	11
6 cout << 'A' << endl; A	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.
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. Outputting or accessing the value of a variable in an expression does not destroy or modify the contents of the variable.

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

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

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.
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.

2.19 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 pre-processor to modify the text of a C++ program before it is compiled. All pre-processor 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.
Note that the preprocessor commands are processed by the preprocessor before the program goes through the compiler.
From Figure 1-3 (lecture 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 lecture 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. Oneway 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 course 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.
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.

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.

The following statements are examples of executable statements:

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

78 | Chapter 2: Basic Elements of C++

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.

```
//*************************************************************
//
//
// 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
Creating a C++ Program | 79
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:
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:

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>
]
Debugging: Understanding and Fixing Syntax Errors | 81
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 familiar and experienced with C++, you will learn how to quickly spot and fix syntax errors. Also, compilers not only discover syntax errors, but also hint and sometimes tell the user where the syntax errors are and how to fix them.

Use of Blanks

In C++, you use one or more blanks to separate numbers when data is input. Blanks are also used to separate reserved words and identifiers from each other and from other symbols. Blanks must never appear within a reserved word or identifier.

Use of Semicolons, Brackets, and Commas

All C++ statements must end with a semicolon. The semicolon is also called a statement terminator.

Note that curly braces, { and }, are not C++ statements in and of themselves, even though they often appear on a line with no other code. You might regard brackets as delimiters, because they enclose the body of a function and set it off from other parts of the program. Brackets have other uses, which will be explained later.

Recall that commas are used to separate items in a list. For example, you use commas when you declare more than one variable following a data type.

Semantics

The set of rules that gives meaning to a language is called semantics. For example, the order-of-precedence rules for arithmetic operators are semantic rules.

If a program contains syntax errors, the compiler will warn you. What happens when a program contains semantic errors? It is quite possible to eradicate all syntax errors in a program and still not have it run. And if it runs, it may not do what you meant it to do.

For example, the following two lines of code are both syntactically correct expressions, but they have different meanings:

```
2 + 3 * 5
```

and:

```
(2 + 3) * 5
```

If you substitute one of these lines of code for the other in a program, you will not get the same results—even though the numbers are the same, the semantics are different. You will learn about semantics throughout this book.

Naming Identifiers

Consider the following two sets of statements:

```
const double A = 2.54; //conversion constant
double x; //variable to hold centimeters
double y; //variable to hold inches
x = y * a;
```

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 runtogether 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.

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;
88 | Chapter 2: Basic Elements of C++
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.

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

Simple Assignment Statement	Compound Assignment Statement
i = i + 5;	i += 5;
counter = counter + 1;	counter += 1;
sum = sum + number;	sum += number;
amount = amount * (interest + 1);	amount *= interest + 1;
x = x / (y + 5);	x /= y + 5;

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.
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
```

Programming Example: Convert Length | 91

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 lecture 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 lecturer 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 halfdollars, 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 2

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 (¼ _2$_{31}$) to 2147483647 (¼ 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 (%).

98 | Chapter 2: Basic Elements of C++

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.

Quick Review | 99

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.