C++ Programming:

From Problem Analysis to Program Design

Seventh Edition

D.S. Malik

Copyright 2015 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

C++ Programming: From Problem Analysis to Program Design, Seventh Edition

D.S. Malik

Product Director: Kathleen McMahon Senior Product

Manager: Jim Gish

Senior Content Developer: Alyssa Pratt Product Assistant:

Gillian Daniels

Content Project Manager: Jennifer Feltri-George

Art Director: GEX Publishing Services Print Buyer: Julio

Esperas

Cover Designer: GEX Publishing Services Cover Photo: [a]

OlegDoroshin/Shutterstock.com Proofreader: Andrea Schein

Indexer: Sharon Hilgenberg

Compositor: Integra Software Services

ISBN- : - - - -

Cengage Learning

First Stamford Place, 4th Floor
Stamford, CT
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: www.cengage.com/global

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

Purchase any of our products at your local college store or at our preferred online store: www.cengagebrain.com

Printed in the United States of America 1 2 3 4 5 6 7 20 19 18 17 16 15 14
ª Cengage Learning

WCN: 02-200-203

Library of Congress Control Number:

warranties or representations, nor do they accept any liabilities with respect to the programs.

TO

My Daughter

Shelly Malik

Copyright 2015 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

[a] HunThomas/Shutterstock.com

[a] HunThomas/Shutterstock.com

11 12

16 17

Copyright 2015 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

[a] HunThomas/Shutterstock.com

WELCOME TO THE SEVENTH EDITION OF C++ Programming: From Problem Analysis to

Program Design. Designed for a first Computer Science (CS1) C++ course, this text provides a breath of fresh air to you and your students. The CS1 course serves as the cornerstone of the Computer Science curriculum. My primary goal is to motivate and excite all CS1 students, regardless of their level. Motivation breeds excitement for learning. Motivation and excitement are critical factors that lead to the success of the programming student. This text is a culmination and development of my classroom notes throughout more than fifty semesters of teaching successful programming to Computer Science students.

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

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

As with any profession, practice is essential. Cooking students practice their recipes. Budding violinists practice their scales. New programmers must practice solving problems and writing code. This is not a C++ cookbook. We do not simply list the C++ syntax followed by an example; we dissect the ''why?'' behind all the concepts. The crucial question of ''why?'' is answered for every topic when first introduced. This technique offers a bridge to learning C++. Students must understand the ''why?'' in 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 Seventh Edition

The seventh edition contains more than 150 new exercises, and more than 30 new

program ming exercises. A special feature of the seventh edition is the mapping of end-of-chapter exercises with their respective learning objective(s) listed at the beginning of the chapter.

The first part of Chapter 4 is reorganized by introducing the if and if...else structures earlier. Also, the first part of Chapter 5 is rewritten. This edition also introduces some features of C++11, such as range-based for loops, in Chapter 8, and illustrates how to use them to process the elements of an array, in Chapters 8 and 12, and a vector object, in Chapter 16. We have also included various new examples, such as Examples 2-19, 3-12, 3-13, 3-14, 4-14, 6-5, 6-16, 8-7, 10-11, 11-4, 13-1, and 16-6.

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 program ming 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 execution. 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 limited 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 disconti nuity, and then later go through this section.

Chapter 8 discusses arrays in detail. This chapter also introduces range-based for loops, a feature of C++11 Standard, and explains how to use them to process the elements of an array. Limitations of ranged-based for loops on arrays passed as parameters to functions are also discussed. Chapter 8 also discusses a sequential search algorithm and a selection sort algorithm. Chapter 9 introduces records (structs). The introduction of structs in this book is similar to C structs. 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). 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.

Chapter 14 discusses exception handling in detail. Chapter 15 introduces and discusses 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 fundamentals 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 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 as well as how to use a random number generator. 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.

Chapter 4

Chapter 5

Chapter 6

FIGURE 1 Chapter dependency diagram

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 structs 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.

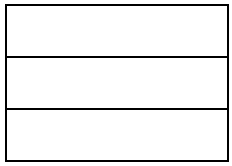
It is recommended that the first six chapters be covered sequentially. After covering the first six chapters, if the reader is interested in learning OOD and OOP early, then Chapter 10 can be studied right after Chapter 6. Chapter 7 can be studied anytime after Chapter 6.

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, 15.

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.

Copyright 2015 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.



Copyright 2015 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

[a] HunThomas/Shutterstock.com

www.cengage.com/coursemate.

The C++ Programming CourseMate includes the following features:

- Videos step you through programs in each chapter, while integrated quizzes provide immediate feedback to gauge your understanding.
- Lab Manual lets you apply material with a wealth of practical, hands-on exercises. • Interactive Quizzes and Study Games drill key chapter concepts, while open ended Assignments develop critical thinking skills.
- Interactive eBook, flashcards, and more!

Instructors may add CourseMate to the textbook package, or students may purchase CourseMate directly through www.cengagebrain.com.

Source Code

The source code, in ANSI/ISO Standard C++, is available for students to download at www.cengagebrain.com and through the CourseMate available for this text. These files are also available to instructors at sso.cengage.com. The input files needed to run some of the programs are also included with the source code.

Instructor Resources

The following supplemental materials are available when this book is used in a classroom setting. All teaching tools are available with this book at sso.cengage.com. An instructor account is required.

Electronic Instructor's Manual

The Instructor's Manual follows the text chapter-by-chapter and includes material to assist in planning and organizing an effective, engaging course.

The Manual includes Overviews, Chapter Objectives, Teaching Tips, Quick Quizzes, Class Discussion Topics, Additional Projects, Additional Resources, and Key Terms. A Sample Syllabus is also available.

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 wherever you want

PowerPoint Presentations

This book comes with PowerPoint slides to accompany each chapter. Slides may be used to guide classroom presentation, to make available to students for chapter review, or to print as classroom handouts. Instructors can add their own slides for additional topics that they introduce to the class, as well as customize the slides with the complete Figure Files from the text.

Solution Files

The solution files for all Programming Exercises, in ANSI/ISO C++, are available for instructor download at sso.cengage.com. The input files needed to run some of the Programming Exercises are also included with the solution files.

[a] HunThomas/Shutterstock.com

There are many people that I must thank who, one way or another, contributed to the success of this book. First, I would like to thank all the students who, during the preparation, were spontaneous in telling me if certain portions needed to be reworded for better understanding and clearer reading. Next, I would like to thank those who e-mailed numerous comments to help improve upon the next edition. I am also very grateful to the reviewers who reviewed earlier versions of this book and offered many critical suggestions on how to improve it.

I owe a great deal to the following reviewers who made helpful, critical suggestions for improving this edition of the text: Terry Hoffer: City College Montana State University; Douglas Kranch: North Central State College; Xiangdong Li: New York City College of Technology; and Jeffrey Miller: Occidental College.

Next, I express thanks to Jim Gish, Senior Product Manager, for recognizing the importance and uniqueness of this project. All this would not have been possible without the careful planning of Senior Content Developer, Alyssa Pratt, and Product Development Manager, Leigh Hefferon. I extend my sincere thanks to Alyssa, as well as to Content Project Manager, Jennifer Feltri-George. I also thank Shanthi Guruswamy of Integra Software Services for assisting us in keeping the project on schedule. I would like to thank Chris Scriver and Serge Palladino of the MQA department at Cengage Learning for patiently and carefully testing the code and discovering typos and errors.

I am thankful to my parents for their blessings.

Finally, I am thankful for the support of my wife Sadhana and especially my daughter, Shelly, to whom this book is dedicated. They cheered me up whenever I was overwhelmed during the writing of this book.

I welcome any comments concerning the text. Comments may be forwarded to the following e-mail address: malik@creighton.edu.

D. S. Malik

Copyright 2015 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Introduction

Terms such as ‘‘the Internet,’’ which were unfamiliar just 20 years ago are now common. Students in elementary school regularly ‘‘surf ’’ the Internet and use computers to design 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. C++ is one of the programming languages, which is well suited for developing software to accomplish specific tasks. The main objective of this book is to help you learn C++ 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 subtrac tion 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 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 completed 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 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 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 it can be made to contain thousands of circuits on a single chip. 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 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 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.

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 perfor mance of arithmetic and logical operations. There are two main components of a computer

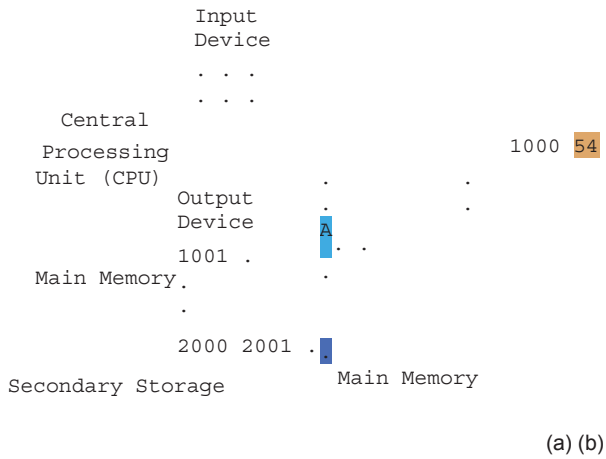
system: hardware and software. In the next few sections, you will learn a brief overview of these components. Let's look at hardware first.

Hardware

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

Central Processing Unit and Main Memory

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



(a) (b)

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

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

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

Today's computers come with main memory consisting of millions to billions of cells.

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

SECONDARY STORAGE

Because programs and data must be loaded into the 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 saved in 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, 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 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.

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 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 program that organizes secondary storage so that you can conveniently access information. Some well-known operating systems are Windows 8, 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 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 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 0s and 1s. The digit 0 or 1 is called a binary digit, or bit. Sometimes a sequence of 0s and 1s is referred to as a binary code or a binary number.

Bit: A binary digit 0 or 1.

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

TABLE 1-1 Binary Units

Unit Symbol	Bits/Bytes
KB	
MB	
GB	
TB	
PB	
EB	

Byte 8 bits

Kilobyte 2^{10} bytes ¼ 1024 bytes

Megabyte 1024 KB ¼ 2^{10} KB ¼ 2^{20} bytes ¼ 1,048,576 bytes

Gigabyte 1024 MB ¼ 2^{10} MB ¼ 2^{30} bytes ¼ 1,073,741,824 bytes

Terabyte 1024 GB ¼ 2^{10} GB ¼ 2^{40} bytes ¼ 1,099,511,627,776 bytes

Petabyte 1024 TB ¼ 2^{10} TB ¼ 2^{50} bytes ¼ 1,125,899,906,842,624 bytes

Exabyte 1024 PB ¼ 2^{10} PB ¼ 2^{60} bytes ¼ 1,152,921,504,606,846,976 bytes

Zettabyte ZB 1024 EB ¼ 2^{10} EB ¼ 2^{70} bytes ¼ 1,180,591,620,717,411,303,424 bytes

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

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

Inside the computer, every character is represented as a sequence of eight

bits, that is, as a byte. Now the eight-bit binary representation of 65 is 01000001. Note that we added 0 to the left of the seven-bit representation of 65 to convert it to an eight-bit representation. Similarly, we add one 0 to the binary value of 51 to get its eight-bit binary representation 00110011.

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

There are other encoding schemes, such as EBCDIC (used by IBM) and Unicode, which is a more recent development. EBCDIC consists of 256 characters; Unicode consists of 65,536 characters. To store a character belonging to Unicode, you need 16 bits or two bytes. Unicode was created to represent a variety of characters and is continuously expanding. It consists of characters from languages other than English.

The Evolution of Programming Languages

The most basic language of a computer, the machine language, provides program instructions in bits. Even though most computers perform the same kinds of operations, the designers of the computer may have chosen different sets of binary codes to perform the operations. Therefore, the machine language of one machine is not necessarily the same as the machine language of another machine. The only consistency among com puters 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:

$$\text{wages} = \text{rate} \quad \text{hours}$$

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 example, 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 instructions. The next step toward making programming easier was to devise high-level languages that were closer to natural languages, such as English, French, German, and Spanish. Basic, FORTRAN, COBOL, C, C++, C#, and Java are all high-level languages. You will learn the high-level language C++ in this book.

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

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

The instruction written in C++ is much easier to understand and is self-explanatory to a novice user who is familiar with basic arithmetic. As in the case of assembly language,

however, the computer cannot directly execute instructions written in a high-level language. To execute on a computer, these C++ instructions first need to be translated into machine language. A program called a compiler translates instructions written in high-level languages into machine code.

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

Processing a C++ Program

In the previous sections, we discussed machine language and high-level languages and showed a C++ statement. 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 1-2, 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 pro cessed 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 pro grams 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 execu tion. 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 1-2 shows how a typical C++ program is processed.

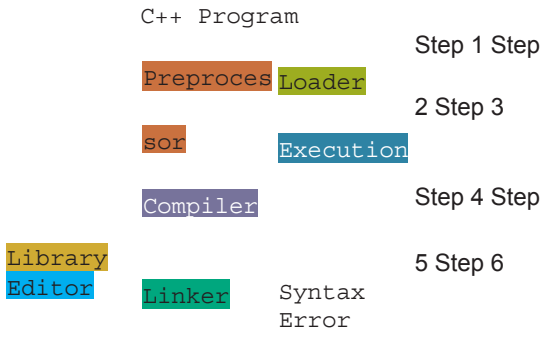


FIGURE 1-2 Processing a C++ program

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 C++ include Visual C++ 2012 Express and Visual Studio 2012 (from Microsoft), and C++ Builder (from Borland). You can also use Dev C++ IDE from Bloodshed Software to create and test C++ 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. More over, 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++ 2012 Express and Visual Studio 2012 is Build or

Rebuild. (For further clarification regarding the use of these commands, check the documentation of these IDEs.) If the program is not yet compiled, each of these commands first compiles the program and then links and produces the executable code.

The Web site http://msdn.microsoft.com/en-us/library/vstudio/ms235629.aspx explains how to use Visual C++ 2012 Express and Visual Studio 2012 to create a C++ 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 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 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 C++, and verify that the algorithm works.
3. Maintain the program by using and modifying it if the problem domain changes.

Figure 1-3 summarizes the first two steps of this programming process.

Problem

Analysis

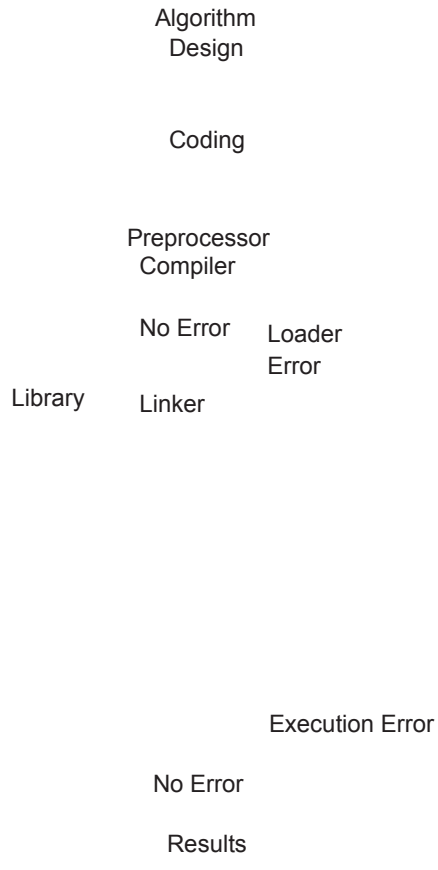


FIGURE 1-3 Problem analysis–coding–execution cycle

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 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 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 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 C++, but you will also learn problem-solving techniques. Most of the chapters contain programming exam ples 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 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:

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

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
Programming with the Problem Analysis–Coding–Execution Cycle | 15

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

the Video

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

Next, you can determine luxuryTax as follows:

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

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 sales person 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. 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. 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 C++ program to play the guessing the number 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
         if average is greater than or equal to 60
               grade = D
         otherwise
               grade = F
```

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

1. totalAverage = 0;
2. Repeat the following steps for each student in the class:
   a. Get student's name.
   b. Use the algorithm as discussed above to find the average test score.
   c. Use the algorithm as discussed above to find the grade.
   d. Update totalAverage by adding the current student's average test score.
3. Determine the class average as follows:

```
classAverage = totalAverage / 10
```

A programming exercise in Chapter 8 asks you to write a C++ program to calculate the average test score and grade for each student in a class.

Earlier in this chapter, we described the problem analysis, coding, and execution cycle. In this section, we gave various examples to illustrate the problem analysis and coding cycle. It must be pointed out that problem analysis is the most important part of programming. Once you have analyzed the problem and written the necessary steps of the solution in your native language, then, as you will see throughout the text, writing the C++ code to implement your solution is relatively easy. In addition, soon you will recognize that the steps of your solutions can be effecively translated into a C++ code. Furthermore, a good problem analysis will lead to a better and cleaner program. Even though we have not yet introduced the syntax of C++, to illustrate how to write a C++ code corresponding to the steps of your solution, let us consider the algorithm designed in Example 1-1. Suppose length, width, perimeter, and area represents the length, width, perimeter, and area of a rectangle. Here are the four steps of the algorithm and their corresponding C++ statement:

Algorithm Step	C++ Instruction (Code)
1. Get the length of the rectangle.	`cin >> length;`
2. Get the width of the rectangle.	`cin >> width;`
3. Calculate the perimeter.	`perimeter = 2 * (length + width);`
4. Calculate the area.	`area = length * width;`

Consider the first statement. In C++, cin stands for common input. During program execution, the code associated with it instructs the user to input data and if the user enters a valid datum, then that datum will be stored in the memory, that is, will become the value of length. The C++ 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 C++ program to compute the area and perimeter, you need to know the basic structure of a C++ program, which will be introduced in the next chapter. However, if you are curious to know how the complete C++ program looks, you can visit the Web site 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 subproblem is then analyzed, and a solution is obtained to solve the subproblem. The solutions to all of the subproblems are then combined to solve the overall problem. This process of implementing a structured design is called structured programming. The structured-design approach is also known as top-down design, bottom-up design, stepwise refinement, and modular programming.

Object-Oriented Programming

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

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

- movie name
- year released
- producer
- production company
- number of copies in stock

Some of the operations on a video object might include:

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

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

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

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

As you can see, you need to learn quite a few things before working with the

OOD methodology. To make this learning easier and more effective, this book purposely divides control structures into two chapters (Chapter 4—selection; Chapter 5— repetition).

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 opera tional, the OOD approach is more effective. C++ 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 C++, such as data types, input/output, control structures, user-defined functions, and arrays, required by either type of pro gramming. 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 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 Standards Institute (ANSI) and International Organization for Standardization (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. Over the last several years, the

C++ committee met several times to further standardize the syntax of C++. In 2011, the second standard of C++ was approved. The main objective of this standard, referred to as C++11, is to make the C++ 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 C++ features are introduced in this book.

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

QUICK REVIEW

1. A computer is an electronic device capable of performing arithmetic and logical operations.
2. A computer system has two components: hardware and software.
3. The central processing unit (CPU) and the main memory are examples of hardware components.
4. All programs must be brought into main memory before they can be executed.
5. When the power is switched off, everything in main memory is lost.
6. Secondary storage provides permanent storage for information. Hard disks, flash drives, 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 computer 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^{10} ¼ 1024 bytes; one megabyte (MB) is 2^{20} ¼ 1,048,576 bytes; one gigabyte (GB) is 2^{30} ¼ 1,073,741,824 bytes; one terabyte (TB) is 2^{40} ¼ 1,099,511,627,776 bytes; one petabyte (PB) is 2^{50} ¼ 1,125,899,906,842,624 bytes; one exabyte (EB) is 2^{60} ¼ 1,152,921,504,606,846,976 bytes; and one zettabyte (ZB) is 2^{70} ¼ 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 execu table 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. 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 interacting objects.
26. An object consists of data and operations on that data.
27. The ANSI/ISO Standard C++ syntax was approved in mid-1998.
28. The second standard of C++, C++11, was approved in 2011.

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 first device known to carry out calculations was the abacus. (1)

b. The calculating device Pascaline could perform only addition and subtraction. (1)

c. The CPU functions under the control of main memory. (2)

d. Information stored in secondary storage is permanent. (2)

e. The language of a computer is a sequence of 0s and 1s. (3)

f. In Unicode every character is coded as a sequence of 16 bits. (3)

g. ZB stands for zero byte. (3)

h. A program written in a high-level programming language is called a source program. (4)

i. A compiler translates the source program into an object program. (6)

j. A linker links and loads the object code from main memory into the CPU for execution. (8)

k. Processing of a C++ program includes six steps. (9)

l. The first step in the problem-solving process is to analyze the problem. (10)

m. The term OOD stands for object-oriented development. (10)

n. In object-oriented design, a program is a collection of interacting functions. (10)

2. What are the two main components of a computer? (2)

3. What is the purpose of main memory? (2)

4. Why is secondary storage needed? (2)

5. What is the function of an operating system? (2)

6. What are the two types of programs? (2)

7. What are the differences between machine languages and high-level languages? (3)

8. What is an object program? (5)

9. What is the function of a loader? (8)

10. What is linking? (8)

11. What kind of errors are reported by a compiler? (8)

12. What is an algorithm? (9)

13. Describe the three steps of the problem-solving process in a programming environment. (9)

14. What are the advantages of problem analysis and algorithm design over directly writing a program in a high-level language? (9)

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: (9)

    testScore1 weightTestScore1
    ...

    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. (9)

17. Given the radius, in inches, and price of a pizza, design an algorithm to find the price of the pizza per square inch. (9)

18. To make a profit, the prices of the items sold in a furniture store are marked up by 80%. After marking up the prices, each item is put on sale at a discount of 10%. 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? (9)

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

20. 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 5000 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)

21. 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 outputs the total billing amount. Your algorithm must use a loop (repetition 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)

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 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)

23. 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)

24. 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 semester 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 student? Design an algorithm that computes the cost of each math class. (9)

25. 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.) (9)

In this chapter, you will learn the basics of C++. As your objective is to learn the C++ programming language, two questions naturally arise. First, what is a computer program? Second, what is programming? A computer program, or a program, is a sequence of statements whose objective is to accomplish a task. Programming is a process of planning and creating a program. These two definitions tell the truth, but not the whole truth, about programming. It may very well take an entire book to give a good and satisfactory definition of programming. You might gain a better grasp of the nature of programming from

an analogy, so let us turn to a topic about which almost everyone has some knowledge—cooking. A recipe is also a program, and everyone with some cooking experience can agree on the following:

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

In this chapter, you will learn the basic elements and concepts of the C++ programming language to create C++ programs. In addition to giving examples to illustrate various concepts, we will also show C++ programs to clarify these concepts. In this section, we provide an example of a C++ 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 introduced 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 C++ 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