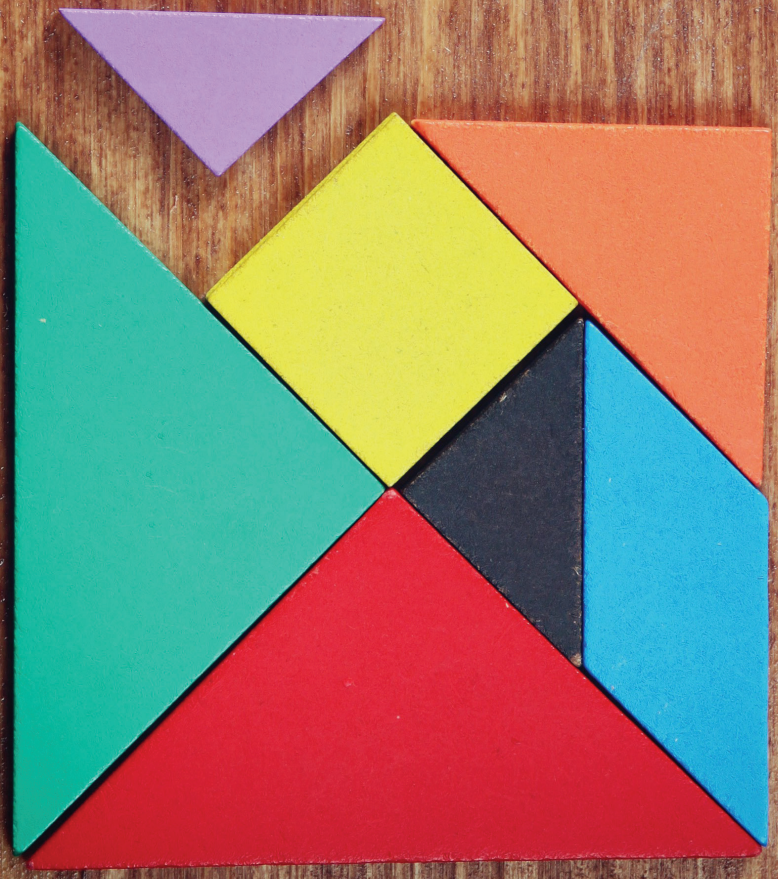
GLOBAL EDITION

Problem Solving and Program Design in C

EIGHTH EDITION

Jeri R. Hanly • Elliot B. Koffman



ALWAYS LEARNING

PEARSON

EIGHTH EDITION
GLOBAL EDITION

PROBLEM SOLVING AND PROGRAM DESIGN

in

this page intentionally left blank

EIGHTH EDITION
GLOBAL EDITION

PROBLEM SOLVING AND PROGRAM DESIGN in C

Jeri R. Hanly, University of Wyoming
Elliot B. Koffman, Temple University

Global Edition Contributions by
Mohit P. Tahiliani, National Institute of Technology, Surathkal

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi
Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director, ECS: Marcia Horton
Executive Editor: Tracy Johnson
Product Marketing Manager: Bram van Kempen
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Project Management Team Lead: Scott Disanno
Program Manager: Carole Snyder
Senior Project Manager: Camille Trentacoste
Assistant Acquisitions Editor, Global Edition: Aditee Agarwal
Associate Project Editor, Global Edition: Sinjita Basu
Media Production Manager, Global Edition: Vikram Kumar
Senior Manufacturing Controller, Production, Global Edition: Trudy Kimber
Operations Supervisor: Vincent Scelta
Operations Specialist: Maura Zaldivar-Garcia
Full-Service Project Management: Cenveo® Publisher Services
Cover Design: Lumina Datamatics
Cover Photo Source: Shutterstock
Cover Printer: Courier Kendallville

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

ISBN 10: 1-292-09881-3
ISBN 13: 978-1-292-09881-4

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

10 9 8 7 6 5 4 3 2 1
14 13 12 11 10

Typeset in New Caledonia 10/12 by Cenveo® Publisher Services.

Printed and bound by Courier Kendallville in the United States of America.

This book is dedicated to

Jeri Hanly's family:
Brian, Kevin, Laura, Grace, and Caleb
Trinity, Alex, Eva, Eli, and Jonah
Eric, Jennifier, Mical, Micah, Josiah, and Rachel

Elliot Koffman's family:
Caryn and Deborah
Richard, Jacquie, and Dustin
Robin, Jeffrey, Jonathan, and Eliana

this page intentionally left blank

PREFACE

Problem Solving and Program Design in C teaches a disciplined approach to problem solving, applying widely accepted software engineering methods to design program solutions as cohesive, readable, reusable modules. We present as an implementation vehicle for these modules a subset of ANSI C—a standardized, industrial-strength programming language known for its power and portability. This text can be used for a first course in programming methods: It assumes no prior knowledge of computers or programming. The text's broad selection of case studies and exercises allows an instructor to design an introductory programming course in C for computer science majors or for students from a wide range of other disciplines.

New to This Edition

Several changes to this edition are listed below. The majority of these changes are in response to the recommendations of our reviewers.

- Chapter 0 information on professional opportunities in computing has been extensively updated.
- Hardware examples in Chapter 1 have been brought up-to-date to reflect current technology.
- Discussion of programming languages in Chapter 1 has been revised to list the most popular languages in use today.
- Chapters on arrays, strings, files, and dynamic data structures have been renamed and reworked to place greater emphasis on the use of pointers.
- Chapter 6 coverage of levels of testing has been updated.
- All chapters contain new programming project problems, and beginning with Chapter 5, some projects are marked as especially appropriate for team programming.
- Three chapters contain new "C in Focus" articles—"Team Programming" (Chapter 5), "Defensive Programming" (Chapter 8), and "Evolving Standards" (Chapter 10).
- Format of many tables, including those that trace execution of code, has been altered to improve readability.
- Exercises on bitwise operations have been added to Appendix C.

Using C to Teach Program Development

Two of our goals—teaching program design and teaching C—may be seen by some as contradictory. C is widely perceived as a language to be tackled only after one has

learned the fundamentals of programming in some other, friendlier language. The perception that C is excessively difficult is traceable to the history of the language. Designed as a vehicle for programming the UNIX operating system, C found its original clientele among programmers who understood the complexities of the operating system and the underlying machine and who considered it natural to exploit this knowledge in their programs. Therefore, it is not surprising that many textbooks whose primary goal is to teach C expose the student to program examples requiring an understanding of machine concepts that are not in the syllabus of a standard introductory programming course.

In this text, we are able to teach both a rational approach to program development and an introduction to ANSI C because we have chosen the first goal as our primary one. One might fear that this choice would lead to a watered-down treatment of ANSI C. On the contrary, we find that the blended presentation of programming concepts and of the implementation of these concepts in C captures a focused picture of the power of ANSI C as a high-level programming language, a picture that is often blurred in texts whose foremost objective is the coverage of all of ANSI C. Even following this approach of giving program design precedence over discussion of C language features, we have arrived at coverage of the essential constructs of C that is quite comprehensive.

Pointers and the Organization of the Book

The order in which C language topics are presented is dictated by our view of the needs of the beginning programmer rather than by the structure of the C programming language. The reader may be surprised to discover that there are multiple chapter titles that include the word "Pointers." This follows from our treatment of C as a high-level language, and our recognition that the critical role of pointers in C is often a challenging concept for students to grasp.

Whereas other high-level languages have separate language constructs for output parameters and arrays, C openly folds these concepts into its notion of a pointer, drastically increasing the complexity of learning the language. We simplify the learning process by discussing pointers from these separate perspectives where such topics normally arise when teaching other programming languages, thus allowing a student to absorb the intricacies of pointer usage a little at a time. Our approach makes possible the presentation of fundamental concepts using traditional high-level language terminology—output parameter, array, array subscript, string—and makes it easier for students without prior assembly language background to master the many aspects of pointer usage. This approach is also helpful for students studying C as a second programming language, since it facilitates their understanding of the many aspects of pointer use as simply C's means of implementing constructs they have already met.

Therefore, this text has not one but five chapters that discuss pointer concepts. Chapter 6 (Pointers and Modular Programming) begins with a discussion of pointers, indirect reference, and the use of pointers to files. It then discusses the use of pointers as simple output and input/output parameters, Chapter 7 deals with

arrays, Chapter 8 presents strings and arrays of pointers. Chapter 11 discusses file pointers again. Chapter 13 describes dynamic memory allocation after reviewing pointer uses previously covered.

Software Engineering Concepts

The book presents many aspects of software engineering. Some are explicitly discussed and others are taught only by example. The connection between good problem-solving skills and effective software development is established early in Chapter 1 with a section that discusses the art and science of problem solving. The five-phase software development method presented in Chapter 1 is used to solve the first case study and is applied uniformly to case studies throughout the text. Major program style issues are highlighted in special displays, and the coding style used in examples is based on guidelines followed in segments of the C software industry. There are sections in several chapters that discuss algorithm tracing, program debugging, and testing.

Chapter 3 introduces procedural abstraction through selected C library functions, parameterless void functions, and functions that take input parameters and return a value. Chapters 4 and 5 include additional function examples including the use of a function as a parameter, and Chapter 6 completes the study of functions that have simple parameters. The chapter discusses the use of pointers to represent output and input/output parameters.

Case studies and sample programs in Chapters 6, 7, and 10 introduce by example the concepts of data abstraction and encapsulation of a data type and operators. Chapter 12 presents C's facilities for formalizing procedural and data abstraction in personal libraries defined by separate header and implementation files. Chapter 14 (on the textbook website) introduces essential concepts of multiprocessing, such as parent and child processes, interprocess communication, mutual exclusion locking, and deadlock avoidance. Chapter 15 (on the textbook website) describes how object-oriented design is implemented by C++.

The use of visible function interfaces is emphasized throughout the text. We do not mention the possibility of using a global variable until Chapter 12, and then we carefully describe both the dangers and the value of global variable usage.

Optional Graphics Sections

Many computer science faculty find that the use of graphics is an excellent motivator in the study of introductory programming and as a vehicle to help students understand how to use libraries and to call functions. This text offers three optional sections with graphics examples:

Section 3.6: Introduction to Computer Graphics
Section 5.11: Loops in Graphics Programs
Section 7.10: Graphics Programs with Arrays

To reduce the overhead required to introduce graphics, we use WinBGIm (Windows BGI with mouse), which is a package based on the Turbo Pascal BGI

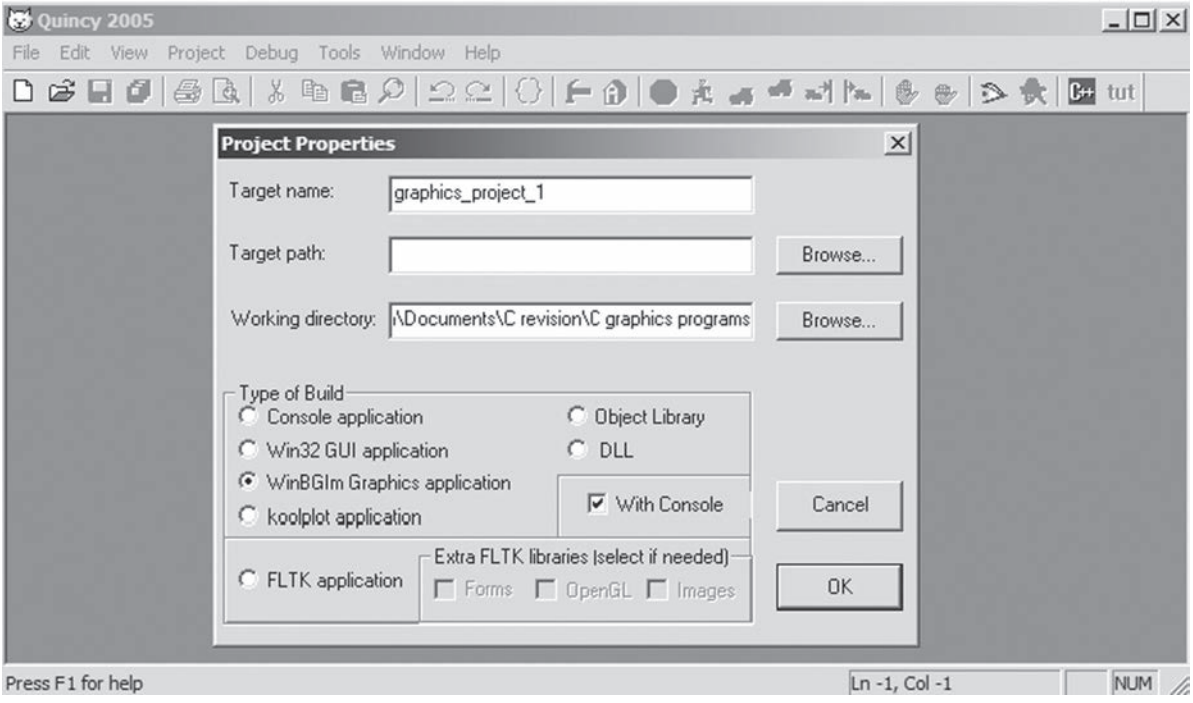


FIGURE 1

(Borland Graphics Interface) library. WinBGIm was created to run on top of the Win32 library by Michael Main and his students at the University of Colorado. Several development platforms appropriate for CS 1 courses have incorporated WinBGIm. Quincy (developed by Al Stevens) is an open-source student-oriented C++ IDE that includes WinBGIm as well as more advanced libraries (http://www.codecutter.net/tools/quincy). Figure 1 shows the Quincy new project window (File → New → Project) with WinBGIm Graphics application selected.

A command-line platform based on the open-source GNU g++ compiler and the emacs program editor is distributed by the University of Colorado (http://www.codecutter.net/tools/winbgim). WinBGIm is also available for Bloodshed Software's Dev-C++ and Microsoft's Visual Studio C++.

Pedagogical Features

We employ the following pedagogical features to enhance the usefulness of this book as a learning tool:

End-of-Section Exercises Most sections end with a number of Self-Check Exercises. These include exercises that require analysis of program fragments as well as short programming exercises.

Examples and Case Studies The book contains a wide variety of programming examples. Whenever possible, examples contain complete programs or functions rather than incomplete program fragments. Each chapter contains one or more substantial case studies that are solved following the software development method. Numerous case studies give the student glimpses of important applications of computing, including database searching, business applications such as billing and sales analysis, word processing, and environmental applications such as radiation level monitoring and water conservation.

Syntax Display Boxes The syntax displays describe the syntax and semantics of new C features and provide examples.

Program Style Displays The program style displays discuss major issues of good programming style.

Error Discussions and Chapter Review Each chapter concludes with a section that discusses common programming errors. The Chapter Review includes a table of new C constructs.

End-of-Chapter Exercises Quick-Check Exercises with answers follow each Chapter Review. There are also review exercises available in each chapter.

End-of-Chapter Projects Each chapter ends with Programming Projects giving students an opportunity to practice what they learned in the chapter.

Appendices

Reference tables of ANSI C constructs appear on the inside covers of the book. Because this text covers only a subset of ANSI C, the appendices play a vital role in increasing the value of the book as a reference. Throughout the book, array referencing is done with subscript notation; Appendix A is the only coverage of pointer arithmetic. Appendix B is an alphabetized table of ANSI C standard libraries. The table in Appendix C shows the precedence and associativity of all ANSI C operators; the operators not previously defined are explained in this appendix. Exercises for practicing some of the bitwise operators are included. Appendix D presents character set tables, and Appendix E lists all ANSI C reserved words.

Supplements

The following supplemental materials are available to all readers of this book at www.pearsonglobaleditions.com/Hanly:

- Source code
- Known errata
- Answers to odd-numbered Self-Check exercises.

The following instructor supplement is available only to qualified instructors at the Pearson Instructor Resource Center. Visit www.pearsonglobaleditions.com/Hanly or contact your local Pearson sales representative to gain access to the IRC.

- Solutions Manual

Acknowledgments

Many people participated in the development of this textbook. The reviewers for this edition, who suggested most of the changes, include Michael Geiger, UMASS Lowell, Lowell, MA; Qi Hao, University of Alabama, Tuscaloosa, AL; Haibing Lu, Santa Clara University, Santa Clara, CA; Susan Mengel, Texas Tech University, Lubbock, TX; Shensheng Tang, Missouri Western State University, St. Joseph, MO; Kevin Mess, College of Southern Nevada, Las Vegas, NV; Samir Iabbassen, Long Island University, Brooklyn, NY; and Ray Lauff, Temple University, Philadelphia, PA. We would also like to acknowledge Michael Main for his assistance with the graphics examples and his students at the University of Colorado who adapted WinBGI to create WinBGIm (Grant Macklem, Gregory Schmelter, Alan Schmidt, and Ivan Stashak).

We also want to thank Charlotte Young of South Plains College for her help in creating Chapter 0, and Jeff Warsaw of WaveRules, LLC, who contributed substantially to Chapter 14. Joan C. Horvath of the Jet Propulsion Laboratory, California Institute of Technology, contributed several programming exercises, and Nelson Max of the University of California, Davis suggested numerous improvements to the text. Jeri acknowledges the assistance of her former colleagues at Loyola University Maryland—James R. Glenn, Dawn J. Lawrie, and Roberta E. Sabin—who contributed several programming projects. We are also grateful for the assistance over the years of several Temple University, University of Wyoming, and Howard University former students who helped to verify the programming examples and who provided answer keys for the host of exercises, including Mark Thoney, Lynne Doherty, Andrew Wrobel, Steve Babiak, Donna Chrupcala, Masoud Kermani, Thayne Routh, and Paul Onakoya.

It has been a pleasure to work with the Pearson team in this endeavor. Tracy Johnson (Executive Editor), Carole Snyder (Program Manager), and Camille Trentacoste (Senior Project Manager) provided ideas and guidance throughout the various phases of manuscript revision.

J.R.H.
E.B.K.

Pearson wishes to thank and acknowledge the following reviewers for their work on the Global Edition:

Vikas Deep Dhiman, *Amity University*

Ll Xin Cindy, *The Hong Kong University of Science and Technology*

Piyali Sengupta, *Freelance Writer*

CONTENTS

0. Computer Science as a Career Path 21

Section 1 Why Computer Science May be the Right Field for You 22
Section 2 The College Experience: Computer Disciplines and Majors to Choose From 24
Section 3 Career Opportunities 29

1. Overview of Computers and Programming 33

1.1 Electronic Computers Then and Now 34
1.2 Computer Hardware 37
1.3 Computer Software 45
1.4 The Software Development Method 52
1.5 Applying the Software Development Method 56
Case Study: Converting Miles to Kilometers 56
1.6 Professional Ethics for Computer Programmers 59
Chapter Review 61

2. Overview of C 65

2.1 C Language Elements 66
2.2 Variable Declarations and Data Types 73
2.3 Executable Statements 79
2.4 General Form of a C Program 89
2.5 Arithmetic Expressions 92
Case Study: Supermarket Coin Processor 102
2.6 Formatting Numbers in Program Output 107
2.7 Interactive Mode, Batch Mode, and Data Files 110
2.8 Common Programming Errors 113
Chapter Review 119

3. Top-Down Design with Functions 127

3.1 Building Programs from Existing Information 128
Case Study: Finding the Area and Circumference of a Circle 129
Case Study: Computing the Weight of a Batch of Flat Washers 132
3.2 Library Functions 137
3.3 Top-Down Design and Structure Charts 144
Case Study: Drawing Simple Diagrams 144
3.4 Functions Without Arguments 146
3.5 Functions with Input Arguments 156
3.6 Introduction to Computer Graphics (Optional) 166
3.7 Common Programming Errors 183
Chapter Review 184

4. Selection Structures: if and switch Statements 193

4.1 Control Structures 194
4.2 Conditions 195
4.3 The if Statement 205
4.4 if Statements with Compound Statements 211
4.5 Decision Steps in Algorithms 214
Case Study: Water Bill Problem 215
4.6 More Problem Solving 224
Case Study: Water Bill with Conservation Requirements 225
4.7 Nested if Statements and Multiple-Alternative Decisions 227
4.8 The switch Statement 237
C in Focus: The Unix Connection 241
4.9 Common Programming Errors 243
Chapter Review 244

5. Repetition and Loop Statements 255

5.1 Repetition in Programs 256
5.2 Counting Loops and the while Statement 258
5.3 Computing a Sum or a Product in a Loop 262
5.4 The for Statement 267
5.5 Conditional Loops 276
5.6 Loop Design 281
5.7 Nested Loops 288
5.8 The do-while Statement and Flag-Controlled Loops 293
5.9 Iterative Approximations 296
Case Study: Bisection Method for Finding Roots 298

5.10 How to Debug and Test Programs 307
C in Focus: Team Programming 309
5.11 Loops in Graphics Programs (Optional) 311
5.12 Common Programming Errors 318
Chapter Review 321

6. Pointers and Modular Programming 337

6.1 Pointers and the Indirection Operator 338
6.2 Functions with Output Parameters 342
6.3 Multiple Calls to a Function with Input/Output Parameters 350
6.4 Scope of Names 356
6.5 Formal Output Parameters as Actual Arguments 358
6.6 Problem Solving Illustrated 362
Case Study: Collecting Area for Solar-Heated House 362
Case Study: Arithmetic with Common Fractions 369
6.7 Debugging and Testing a Program System 378
6.8 Common Programming Errors 381
Chapter Review 381

7. Array Pointers 397

7.1 Declaring and Referencing Arrays 398
7.2 Array Subscripts 401
7.3 Using for Loops for Sequential Access 403
7.4 Using Array Elements as Function Arguments 408
7.5 Array Arguments 410
7.6 Searching and Sorting an Array 423
7.7 Parallel Arrays and Enumerated Types 428
7.8 Multidimensional Arrays 436
7.9 Array Processing Illustrated 441
Case Study: Summary of Hospital Revenue 441
7.10 Graphics Programs with Arrays (Optional) 450
7.11 Common Programming Errors 459
Chapter Review 460

8. Strings 475

8.1 String Basics 476
8.2 String Library Functions: Assignment and Substrings 482
8.3 Longer Strings: Concatenation and Whole-Line Input 491

8.4 String Comparison 496
C in Focus: Defensive Programming 498
8.5 Arrays of Pointers 500
8.6 Character Operations 506
8.7 String-to-Number and Number-to-String Conversions 511
8.8 String Processing Illustrated 518
Case Study: Text Editor 518
8.9 Common Programming Errors 527
Chapter Review 529

9. Recursion 541

9.1 The Nature of Recursion 542
9.2 Tracing a Recursive Function 548
9.3 Recursive Mathematical Functions 556
9.4 Recursive Functions with Array and String Parameters 562
Case Study: Finding Capital Letters in a String 562
Case Study: Recursive Selection Sort 565
9.5 Problem Solving with Recursion 569
Case Study: Operations on Sets 569
9.6 A Classic Case Study in Recursion: Towers of Hanoi 577
9.7 Common Programming Errors 582
Chapter Review 584

10. Structure and Union Types 591

10.1 User-Defined Structure Types 592
10.2 Structure Type Data as Input and Output Parameters 598
10.3 Functions Whose Result Values Are Structured 604
C in Focus: Evolving Standards 606
10.4 Problem Solving with Structure Types 608
Case Study: A User-Defined Type for Complex Numbers 608
10.5 Parallel Arrays and Arrays of Structures 616
Case Study: Universal Measurement Conversion 618
10.6 Union Types (Optional) 627
10.7 Common Programming Errors 634
Chapter Review 634

11. Text and Binary File Pointers 649

11.1 Input/Output Files: Review and Further Study 650
11.2 Binary Files 660

11.3 Searching a Database 666
Case Study: Database Inquiry 667
11.4 Common Programming Errors 676
Chapter Review 677

12. Programming in the Large 685

12.1 Using Abstraction to Manage Complexity 686
12.2 Personal Libraries: Header Files 689
12.3 Personal Libraries: Implementation Files 694
12.4 Storage Classes 697
12.5 Modifying Functions for Inclusion in a Library 701
12.6 Conditional Compilation 704
12.7 Arguments to Function main 708
12.8 Defining Macros with Parameters 711
12.9 Common Programming Errors 716
Chapter Review 717

13. Pointers and Dynamic Data Structures 725

13.1 Pointers 726
13.2 Dynamic Memory Allocation 731
13.3 Linked Lists 736
13.4 Linked List Operators 742
13.5 Representing a Stack with a Linked List 747
13.6 Representing a Queue with a Linked List 751
13.7 Ordered Lists 757
Case Study: Maintaining an Ordered List of Integers 758
13.8 Binary Trees 769
13.9 Common Programming Errors 779
Chapter Review 780

14. Multiprocessing Using Processes and Threads
(Online at www.pearsonglobaleditions.com/Hanly)

14.1 Multitasking
14.2 Processes
14.3 Interprocess Communications and Pipes
14.4 Threads
14.5 Threads Illustrated
Case Study: The Producer/Consumer Model

14.6 Common Programming Errors
Chapter Review

15. On to C++
(Online at www.pearsonglobaleditions.com/Hanly)

15.1 C++ Control Structures, Input/Output, and Functions
15.2 C++ Support for Object-Oriented Programming
Chapter Review

Appendices

A More about Pointers 789
B ANSI C Standard Libraries 795
C C Operators 813
D Character Sets 819
E ANSI C Reserved Words 821

Answers to Odd-Numbered Self-Check Exercises
(Online at www.pearsonglobaleditions.com/Hanly)

Glossary 823

Index 829

EIGHTH EDITION
GLOBAL EDITION

PROBLEM SOLVING AND PROGRAM DESIGN

in

this page intentionally left blank

CHAPTER

0

Computer Science as a Career Path

CHAPTER OBJECTIVES

- To learn why computer science may be the right field for you
- To become familiar with different computer disciplines and related college majors
- To find out about career opportunities

Introduction

In order to choose a course of study and eventually a desirable career path, we may ask many important questions. Why would we choose this field? Will we be good at it? Will there be jobs for us when we finish our education? Will we enjoy our work? This chapter sheds some light on these types of questions for anyone contemplating a degree in computer science or a related field.

Section 1 Why Computer Science May Be the Right Field for You

Reasons to Major in Computer Science

Millennials Those born from 1982 on are said to be confident, social and team-oriented, proud of achievement, prone to use analytic skills to make decisions, and determined to seek security, stability, and balance for themselves

Almost everything we do is influenced by computing. Today's generation of college students, dubbed the **Millennials**, are not surprised by this statement. They have grown up with computers, the Internet, instant communication, social networking, and electronic entertainment. They embrace new technology and expect it to do fantastic things.

However, previous generations are not as comfortable with technology and try to solve problems without always thinking of technology first. Many people in the workforce resist the changes that technology requires. They often turn to the youngest employees to take over technology issues and to make choices that will have important consequences.

This difference among generations creates a great environment for bright and dedicated students to choose to major in computer science or a related field. The computer industry is one of the fastest-growing segments of our economy and promises to continue to see growth well into the future. In order to be competitive, businesses must continue to hire well-trained professionals not only to produce high-quality products for the present, but also to plan creative scientific and engineering advances for the future.

A person who is part of the computer industry can choose from a wide variety of fields where many interesting and challenging problems will need to be solved. In addition to all the business and communication jobs that may first come to mind, people with degrees in computer science are working on problems from all spectrums of life. A quick review of technical articles highlights such areas as developing electronic balloting for state and national elections, using signals from wireless devices to update vehicle and pedestrian travel times in order to make the best decisions for traffic signals or management of construction zones, and using a supercomputer-powered "virtual earthquake" to study benefits of an early warning system using 3-D models of actual geographic locations and damage scenarios.

Some problems being worked on right now by computer professionals in the medical world include understanding how the human brain works by modeling brain activation patterns with emphasis on helping people impacted by autism or disorders like paranoid schizophrenia; customizing a wide array of helpful devices for the physically impaired, from programmable robotic prostheses to digital "sight"; gathering information from implanted pacemakers in order to make timely decisions in times of crisis; developing a computer system capable of recognizing human emotional states by analyzing a human face in real time; and developing human–computer interfaces that allow a computer to be operated solely by human gestures in order to manipulate virtual objects.

The fields of security and law enforcement present many challenges to the computer professional, and include the following: The U.S. government is performing observational studies on normal behavior in online worlds in hopes of developing techniques for uncovering online activities of terrorist groups. Advancements in voice biometrics technology allow speech to be analyzed by computer software to determine identity, truthfulness, and emotional states. Electronic protection against malicious software is of great concern to national economies and security interests.

Some of our world's most challenging problems will be worked on by teams of professionals from many disciplines. Obviously, these teams will include computer professionals who are creative and possess the knowledge of how to best use technology. In the near future, we will see much innovation in the areas of the human genome project, environmental monitoring, AIDS vaccine research, clean fuels, tracking weather changes by using robots in potentially dangerous areas, and using supercomputers to simulate the earth's architecture and functions in order to predict natural disasters. A way to make a positive difference in the world would be to study computing, either as your primary focus or as a means of advancing technology in another field.

Traits of a Computer Scientist

An individual's personality and character traits typically influence the field he or she chooses to study and eventually in which he or she will work. The demands of certain fields are met by individuals with certain capabilities. It makes sense that people who are successful computer science students will have many common traits. Read the following description and decide if it sounds like you.

Foremost, you must love the challenge of solving problems. Computer science is more about finding solutions to problems than it is about using the current computer hardware or programming language. Solving problems requires being creative and "thinking outside the box." You must be willing to try things that are different from the "accepted" solution.

You enjoy working with technology and enjoy being a lifelong learner. You enjoy puzzles and work tenaciously to find solutions. You probably don't even notice that the hours have flown by as you are narrowing in on the answer. You enjoy building things, both in the actual world and in a "virtual world." You can see how to customize a particular object to make it work in a specific environment. You like to tackle large projects and see them to completion. You like to build things that are useful to people and that will have a positive impact on their lives.

To be successful in the workplace, you must also be a good communicator. You should be able to explain your plans and solutions well to both technical and nontechnical people. You must be able to write clearly and concisely in the technical environment. Since most projects involve multiple people, it is important to work well in a group. If you plan to become a manager or run your own company, it is very important to be able to work with different personalities.

Frederick P. Brooks, famous for leading the team that developed the operating system for the IBM System/360, wrote a book in the 1970s titled *The Mythical Man Month—Essays on Software Engineering*. Even though much has changed in the computing world since he wrote the book, his essays still hold a lot of relevance today. He listed the "Joys of the Craft" as the following: First is the sheer joy of making things of your own design. Second is the pleasure of making things that are useful to and respected by other people. Third is the joy of fashioning complex puzzle-like entities into a system that works correctly. Fourth is the joy of always learning because of the nonrepetitive nature of the work. Finally, there is the joy of working with a very tractable medium. The programmer can create in his or her imagination and readily produce a product that can be tested and easily changed and reworked. Wouldn't the sculptor or civil engineer enjoy such easy tractability!

The IBM System/360 was a mainframe computer system family announced by IBM in 1964. It was the first family of computers making a clear distinction between architecture and implementation, allowing IBM to release a suite of compatible designs at different price points. The design is considered by many to be one of the most successful computers in history, influencing computer design for years to come (see Figure 0.1).

FIGURE 0.1

IBM introduced the system/360 family of business mainframe computers in 1964. (©2012 akg-images/Paul Almasy/Newscom. Unauthorized use not permitted.)

Section 2 The College Experience: Computer Disciplines and Majors to Choose From

Most professionals in the computing industry have at least an undergraduate degree in mathematics, computer science, or a related field. Many have advanced degrees, especially those involved primarily in research or education.

Computing is a broad discipline that intersects many other fields such as mathematics, science, engineering, and business. Because of such a wide range of choices, it is impossible for anyone to be an expert in all of them. A career involving computing requires the individual to focus his or her efforts while obtaining a college degree.

There are many different degrees that involve computing offered at institutions of higher learning. These degrees can even be from different departments within the same institution. Although computing degrees can share some of the same courses, they can also be quite different from each other. Choosing among them can be confusing.

To ease this confusion, it is wise for students to consult with academic advisors in the computer science department, the computer or electrical engineering department, and the business school to explore the options available in their specific institution. Next we summarize some of the degree programs that your institution may offer.

Computer Science

Computer science as a discipline encompasses a wide range of topics from theoretical and algorithmic foundations to cutting-edge developments. The work computer scientists are trained to do can be arranged into three categories:

- Designing and implementing useful software
- Devising new ways to use computers
- Developing effective ways to solve computing problems

A computer science degree consists of courses that include computing theory, programming, and mathematics. These courses ultimately develop the logic and reasoning skills integral to becoming a computer scientist. The math sequence includes calculus I and II (and in many cases, calculus III) as well as discrete mathematics. Some students also study linear algebra and probability and statistics. A computer science degree offers a comprehensive foundation that permits graduates to understand and adapt to new technologies and new ideas. Computer science departments are often found at universities as part of the science, engineering, or mathematics divisions.

Computer scientists take on challenging programming jobs, supervise other programmers, and advise other programmers on the best approaches to be taken. Computer science researchers are working with scientists from other fields to perform such tasks as using databases to create and organize new knowledge, making robots that will be practical and intelligent aides, and using computers to help decipher the secrets of human DNA. Their theoretical background allows them to determine the best performance possible for new technologies and their study of algorithms helps them to develop creative approaches to new (and old) problems.

Computer Engineering

For students who are more interested in understanding and designing the actual computing devices, many opportunities are available in computer engineering,

which is concerned with the design and construction of computers and computer-based systems. A computer engineering degree involves the study of hardware, software, communications, and the interaction among them, and is a customized blend of an electrical engineering degree with a computer science degree.

The computer engineering curriculum includes courses on the theories, principles, and practices of traditional electrical engineering as well as mathematics through the standard calculus sequence and beyond. This knowledge is then applied in courses dealing with designing computers and computer-based devices. In addition, programming courses are required so that the computer engineer can develop software for digital devices and their interfaces.

Currently, an important area for computer engineers involves embedded systems. This involves the development of devices that have software and hardware embedded in them such as cell phones, digital music players, alarm systems, medical diagnostic devices, laser surgical tools, and so on. The devices a computer engineer might work with are limitless as he or she applies his or her knowledge of how to integrate hardware and software systems.

Information Systems

The information systems (IS) field focuses on integrating technology into businesses and other enterprises to manage their information in an efficient and secure manner. In this area, technology is viewed as an instrument for generating, processing, and distributing information. Therefore, the focus in this field is on business and organizational principles.

Most IS programs are located in the business school of a university or college, and IS degrees combine business and computing coursework, and the math that is required has a business application focus. These degrees may be found under such programs as Computer Information Systems (CIS) or Management Information Systems (MIS). Degree program names are not always consistent, but they all have their focus on business principles and applications of technology with less emphasis on the theory of computer science or the digital design of computer engineering.

IS specialists must understand both technical and organizational factors, and must be able to help an organization determine how to use information and technology to provide a competitive edge. These professionals serve as a bridge between the technical community and the management community within an organization. They are called on to determine the best way to use technology, organize information, and communicate effectively.

Information Technology

An Information Technology (IT) program prepares students to meet the computer technology needs of business, government, healthcare, schools, and other organizations. IT has its emphasis on the technology itself, more than on the information

handled, the theory behind it, or how to design hardware or software. IT professionals work with computer systems to ensure they work properly, are secure, are upgraded and maintained, and are replaced as appropriate.

Because computers have become integral parts of the work environment for all employees at all levels of the organization, many enterprises must maintain departments of IT workers. Organizations of every kind are dependent on information technology on a daily basis and the need for qualified workers is great.

Degree programs in IT are commonly found in business or information management departments, or as an alternate degree in a computer science department. IT programs in business departments focus on using applications to meet the requirements of networking, systems integration, and resource planning. The emphasis is less on programming and more on using programs already written to the best advantage. IT programs in computer science departments often have more emphasis on programming for computer users, with a focus on writing software for interactive web pages, multimedia, and cloud computing.

IT specialists select appropriate hardware and software products for an organization and then integrate these products within the existing infrastructure. They install and customize and maintain the software as needed. Other examples of responsibilities include network administration and security, design and implementation of web pages, development of multimedia resources, oversight of e-mail systems, and installation of communication components. User support and training are often important responsibilities for the IT professional as well.

Software Engineering

Software engineering (SE) is the discipline of developing and maintaining large software systems. These systems must behave reliably and efficiently, be affordable, and satisfy all requirements defined for them. SE seeks to integrate the theory of computer science and mathematics with the practical engineering principles developed for physical objects.

An SE degree program is closely related to the computer science degree program, and they are usually offered within the same department. In fact, most computer science curricula require one or more software engineering courses. An SE degree can be considered a specialized degree within the confines of the field of computer science.

SE students learn more about software reliability and maintenance of large systems and focus more on techniques for developing and maintaining software that is engineered to be correct from its inception. Most programs require SE students to participate in group projects for the development of software that will be used in earnest by others. Students assess customer needs, develop usable software, test the product thoroughly, and analyze its usefulness.

Professionals who hold a software engineering degree expect to be involved with the creation and maintenance of large software systems that may be used by

many different organizations. Their focus will be on the design principles that make the system viable for many people and through many years.

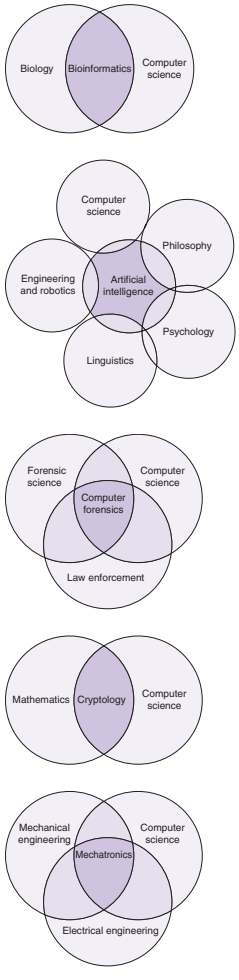
Although an SE degree has a recognized description, the term *software engineer* is merely a job label in the workplace. There is no standard definition for this term when used in a job description, and its meaning can vary widely among employers. An employer may think of a programmer or an IT specialist as a software engineer.

Mixed Disciplinary Majors

Technology is opening doors for fields of study that combine different sciences or engineering fields with computing as illustrated by Figure 0.2. Institutes of higher learning have responded by offering courses or programs for multidisciplinary majors. Some examples follow.

FIGURE 0.2

Illustrations of the overlapping fields within mixed disciplinary majors



- **Bioinformatics** is the use of computer science to maintain, analyze, and store biological data as well as to assist in solving biological problems—usually on the molecular level. Such biological problems include protein folding, protein function prediction, and phylogeny (the history, origin, and evolution of a set of organisms). The core principle of bioinformatics involves using computing resources to help solve problems on scales of magnitude too great for human observation.
- **Artificial Intelligence** (AI) is the implementation and study of systems that can exhibit autonomous intelligence or behaviors. AI research draws from many fields including computer science, psychology, philosophy, linguistics, neuroscience, logic, and economics. Applications include robotics, control systems, scheduling, logistics, speech recognition, handwriting recognition, understanding natural language, proving mathematical theorems, data mining, and facial recognition.
- **Computer Forensics** is a branch of forensic science pertaining to legal evidence that may be found in computers and digital storage devices. The collection of this evidence must adhere to standards of evidence admissible in a court of law. Computer forensics involves the fields of law, law enforcement, and business.
- **Cryptology** (or cryptography) is the practice and study of hiding information and involves mathematics, computer science, and engineering. Electronic data security for commerce, personal uses, and military uses continue to be of vast importance.
- **Mechatronics** is the combination of mechanical engineering, electronic engineering, and software engineering in order to design advanced hybrid systems. Examples of mechatronics include production systems, planetary exploration rovers, automotive subsystems such as anti-lock braking systems, and autofocus cameras.

Even when the definitions are given for the different computing disciplines mentioned in this chapter, it is easy to see that there is great overlap among all of them.

In fact, many professionals who have earned a computer science degree may be working in jobs that are closer to an information systems description or vice versa. The student is encouraged to choose a computing field that seems closest to his or her personal goals.

Section 3 Career Opportunities

The Bureau of Labor Statistics is the principal fact-finding agency for the U.S. Federal Government in the field of labor economics and statistics. This agency publishes *The Occupational Outlook Handbook*, which is a nationally recognized source of career information, designed to provide valuable assistance to individuals making decisions about their future work lives. The *Handbook* is revised every two years. Table 0.1 gives an overview of some of the major computer occupations tracked by the U.S. Bureau of Labor Statistics.

The Demand in the United States and in the World

According to the *BLS Occupational Outlook Handbook,* software developer, database administrator, and network/computer systems administrator are three of the occupations projected to grow at the fastest rates over the 2010–2020 decade. Strong employment growth combined with a limited supply of qualified workers will result in excellent employment prospects. Those with practical experience and at least a bachelor's degree in one of the computing fields described in Section 2 should have the best opportunities. Employers will continue to seek computer professionals with strong programming, systems analysis, interpersonal, and business skills.

The growing need for computer professionals is increased by the ongoing retirement of a generation of baby boomers, and all of this is occurring as the U.S. government projects continued rapid growth in many computer science and IT occupations.

Today's students need not worry about the impact that outsourcing computer jobs to other countries will have on their ability to find a job. The fact is that many companies have been disappointed in the results when outsourcing entire projects. Some of the more mundane aspects of coding can be outsourced, but the more creative work is best kept in house. For example, during the design and development of a new system, interaction with specialists from other disciplines and communication with other team members and potential system users are of utmost importance. These activities are negatively impacted by communication difficulties across cultures and long distances. Many companies are rethinking outsourcing and doing more system development at home.

The number of graduates from the computing fields will not meet the demand in the marketplace in the foreseeable future. Projections and statistics show that there will be plenty of jobs to be offered to the qualified computer professional and the salaries will be higher than the average full-time worker earns in the United States.

TABLE 0.1 Computer, Computer Engineering, and Information Technology Occupations

Occupation	Job Summary	Entry-Level Education
Computer and Information Research Scientists	Computer and information research scientists invent and design new technology and find new uses for existing technology. They study and solve complex problems in computing for business, science, medicine, and other uses.	Doctoral or professional degree
Computer Programmers	Computer programmers write code to create software programs. They turn the program designs created by software developers and engineers into instructions that a computer can follow.	Bachelor's degree
Computer Support Specialists	Computer support specialists provide help and advice to people and organizations using computer software or equipment. Some, called technical support specialists, support information technology (IT) employees within their organization. Others, called help-desk technicians, assist non-IT users who are having computer problems.	Some college, no degree
Computer Systems Analysts	Computer systems analysts study an organization's current computer systems and procedures and make recommendations to management to help the organization operate more efficiently and effectively. They bring business and information technology (IT) together by understanding the needs and limitations of both.	Bachelor's degree
Database Administrators	Database administrators use software to store and organize data, such as financial information and customer shipping records. They make sure that data are available to users and are secure from unauthorized access.	Bachelor's degree
Information Security Analysts, Web Developers, and Computer Network Architects	Information security analysts, web developers, and computer network architects all use information technology (IT) to advance their organization's goals. Security analysts ensure a firm's information stays safe from cyber attacks. Web developers create websites to help firms have a public face. Computer network architects create the internal networks all workers within organizations use.	Bachelor's degree
Network and Computer Systems Administrators	Network and computer systems administrators are responsible for the day-to-day operation of an organization's computer networks. They organize, install, and support an organization's computer systems, including local area networks (LANs), wide area networks (WANs), network segments, intranets, and other data communication systems.	Bachelor's degree

Software Developers	Software developers are the creative minds behind computer programs. Some develop the applications that allow people to do specific tasks on a computer or other device. Others develop the underlying systems that run the devices or control networks.	Bachelor's degree
Computer Hardware Engineers	Computer equipment such as chips, circuit boards, or routers. By solving complex problems in computer hardware, these engineers create rapid advances in computer technology.	Bachelor's degree

Excerpted with permission from U.S. Bureau of Labor Statistics, Office of Occupational Statistics and Employment Projections, March 29, 2012

The Demand for Underrepresented Groups

The demand for women and minorities to fill computer-related jobs is higher than ever. The computer-related fields have traditionally seen small numbers of women and minorities in the workplace. Colleges and universities want to attract these groups to computer science and IS departments and often offer good scholarships and opportunities.

According to a study by the National Center for Women and Information Technology, the most successful IT teams were also the most diverse. The study showed that diversity of thought leads to innovation, and that companies should be aware of the significance of diversity. Prospective students should not be turned away by the stereotypical view of a "computer geek" who sits in front of a computer all day, but should realize all the opportunities to be found in such a diverse and fast-growing field. Computer professionals will be creating the applications that allow computers to solve real-world problems.

New Careers Constantly on the Horizon

It is clear that there will be a healthy need for computer professionals in the career paths that are known about today. For the student just starting to plan a career, there will surely be opportunities that have not even been imagined yet. The possibilities are amazing and the rewards are many.

One such window into the future can be seen in the work of Bruce Donald, a professor of Computer Science and of Biochemistry at Duke University. Through his research, Professor Donald has developed microscopic robots that can be controlled individually or as a group. These robots (see Figure 0.3) are measured in microns (millionths of a meter) and are almost 100 times smaller than previous robotic designs of their kind. Each robot can respond differently to the same single

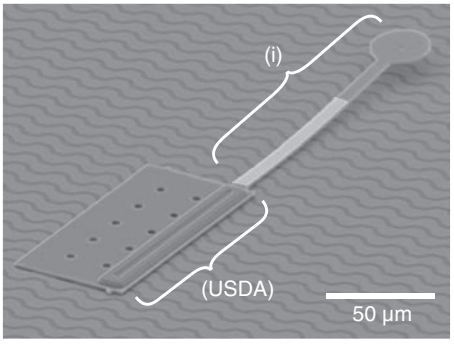


FIGURE 0.3

Scanning-electron micrograph of a microrobot consisting of an untethered scratch-drive actuator (USDA) that provides forward motion, and a curved steering-arm actuator (i) that moves robot in straight line or turns.
(© 2013 *The International Journal of Robotics Research*)

"global control signal" as voltages charge and discharge on their working parts. A budding computer scientist should see many fantastic applications for these devices!

The student who chooses to major in computer science or a related field can look forward to challenging and interesting classes. The job market will be wide open upon graduation, with the assurance that such degrees will be highly marketable. A new employee or researcher will have opportunities to be at the forefront of innovative technology in a constantly changing world. The prospects are limited only by the imagination.

CHAPTER 1

Overview of Computers and Programming

CHAPTER OBJECTIVES

- To learn about the different categories of computers
- To understand the role of each component in a computer
- To understand the purpose of an operating system
- To learn the differences between machine language, assembly language, and higher-level languages
- To understand what processes are required to run a C program
- To learn how to solve a programming problem in a careful, disciplined way
- To understand and appreciate ethical issues related to the use of computers and programming

In developed countries, life in the twenty-first century is conducted in a veritable sea of computers. From the coffeepot that turns itself on to brew your morning coffee to the microwave that cooks your breakfast to the automobile that you drive to work to the automated teller machine you stop by for cash, virtually every aspect of your life depends on **computers**. These machines which receive, store, process, and output information can deal with data of all kinds: numbers, text, images, graphics, and sound, to name a few.

computer a machine that can receive, store, transform, and output data of all kinds

The computer program's role in this technology is essential; without a list of instructions to follow, the computer is virtually useless. Programming languages allow us to write those programs and thus to communicate with computers.

You are about to begin the study of computer science using one of the most versatile programming languages available today: the C language. This chapter introduces you to the computer and its components and to the major categories of programming languages. It discusses how C programs are processed by a computer. It also describes a systematic approach to solving programming problems called the *software development method* and shows you how to apply it.

1.1 Electronic Computers Then and Now

In our everyday life, we come in contact with computers frequently, some of us using computers for creating presentations and other documents, tabulating data in spreadsheets, or even having studied programming in high school. But it wasn't always this way. Not so long ago, most people considered computers to be mysterious devices whose secrets were known only by a few computer wizards.

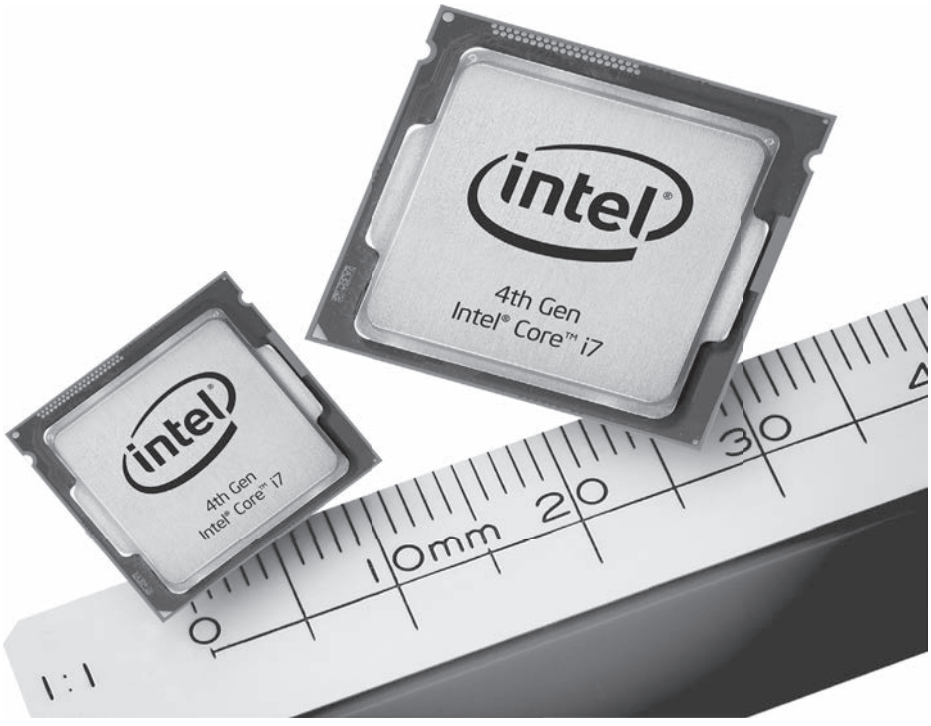
The first electronic computer was built in the late 1930s by Dr. John Atanasoff and Clifford Berry at Iowa State University. Atanasoff designed his computer to assist graduate students in nuclear physics with their mathematical computations.

The first large-scale, general-purpose electronic digital computer, called the ENIAC, was completed in 1946 at the University of Pennsylvania with funding from the U.S. Army. Weighing 30 tons and occupying a 30-by-50-foot space, the ENIAC was used to compute ballistics tables, predict the weather, and make atomic energy calculations.

These early computers used vacuum tubes as their basic electronic component. Technological advances in the design and manufacture of electronic components led to new generations of computers that were considerably smaller, faster, and less expensive than previous ones.

FIGURE 1.1

The Intel® Core™ i7 processor chip contains the full circuitry of a central processing unit in an integrated circuit whose small size and low power requirements make it suitable for use in mobile internet devices. (Intel Corporation Pressroom Photo Archives)



computer chip (processor chip) a silicon chip containing the circuitry for a computer processor

Using today's technology, the entire circuitry of a computer processor can be packaged in a single electronic component called a **computer** or **processor chip** (Fig. 1.1), which is less than one-fourth the size of a standard postage stamp. Their affordability and small size enable computer chips to be installed in watches, cellphones, GPS systems, cameras, home appliances, automobiles, and, of course, computers.

Today, a common sight in offices and homes is a personal computer, which can cost less than $500 and sit on a desk (Fig. 1.2a) and yet has as much computational power as one that 40 years ago cost more than $100,000 and filled a 9-by-12-foot room. Even smaller computers can fit inside a briefcase or purse (Fig. 1.2b, c) or your hand (Fig. 1.2d).

Modern computers are categorized according to their size and performance. *Personal computers*, shown in Fig. 1.2, are used by a single person at a time. Large real-time transaction processing systems, such as ATMs and other banking networks, and corporate reservations systems for motels, airlines, and rental cars use *mainframes*, very powerful and reliable computers. The largest capacity and fastest computers are called *supercomputers* and are used by research laboratories and in computationally intensive applications such as weather forecasting.

FIGURE 1.2

(a) Desktop Computer, iMac. (© Hugh Threlfall/Alamy). (b) Hewlett Packard Laptop. (© Hewlett-Packard Company). (c) iPad. (© D. Hurst/Alamy). (d) Android phone, LG Thrill 4G. (© Handout/MCT/Newscom).

a

b

c

d

hardware the actual computer equipment

software the set of programs associated with a computer

program a list of instructions that enables a computer to perform a specific task

binary number a number whose digits are 0 and 1

The elements of a computer system fall into two major categories: hardware and software. **Hardware** is the equipment used to perform the necessary computations and includes the central processing unit (CPU), monitor, keyboard, mouse, printer, and speakers. **Software** consists of the **programs** that enable us to solve problems with a computer by providing it with lists of instructions to perform.

Programming a computer has undergone significant changes over the years. Initially, the task was very difficult, requiring programmers to write their program instructions as long **binary numbers** (sequences of 0s and 1s). High-level programming languages such as C make programming much easier.

EXERCISES FOR SECTION 1.1

Self-Check

1. Is a computer program a piece of hardware or is it software?
2. For what applications are mainframes used?

1.2 Computer Hardware

Despite significant variations in cost, size, and capabilities, modern computers resemble one another in many basic ways. Essentially, most consist of the following components:

- Main memory
- Secondary memory, which includes storage devices such as hard disks, CDs, DVDs, and flash drives
- Central processing unit
- Input devices, such as keyboards, mice, touchpads, touch screens, scanners, joysticks, and microphones
- Output devices, such as screens, printers, and speakers

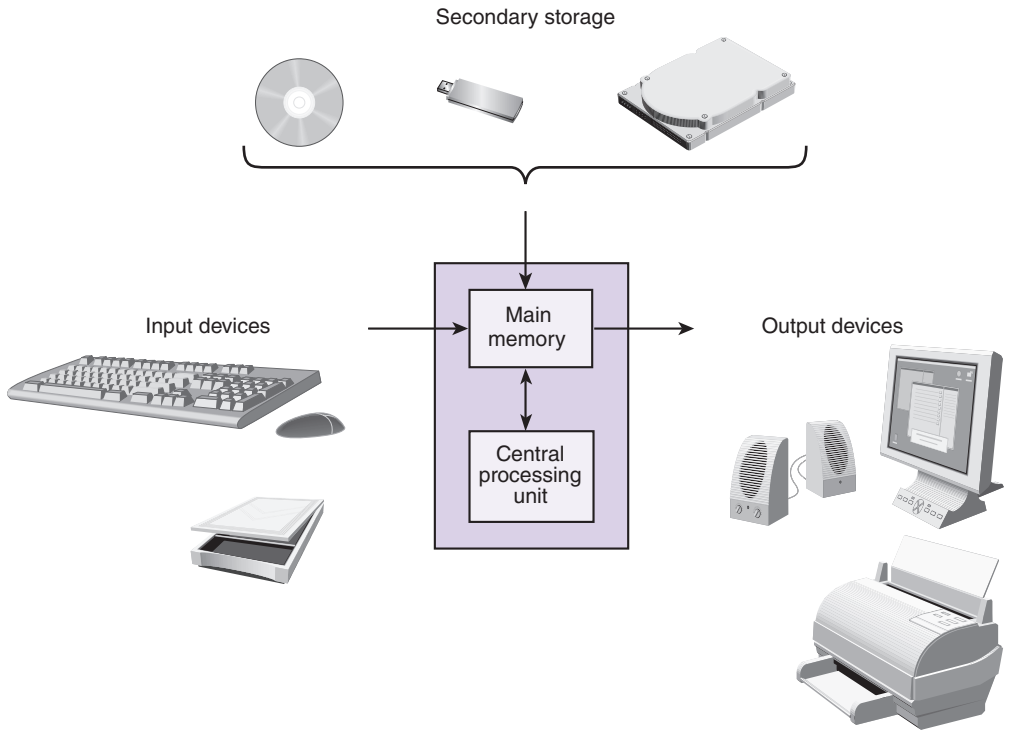
Figure 1.3 shows how these components interact in a computer, with the arrows pointing in the direction of information flow. The program must first be transferred from *secondary storage* to *main memory* before it can be executed. Normally, the person using a program (the *program user*) must supply some data to be processed. These data are entered through an *input device* and are stored in the computer's *main memory,* where they can be accessed and manipulated by the *central processing unit.* The results of this manipulation are then stored back in *main memory.* Finally, the information in main memory can be displayed through an *output device.* In the remainder of this section, we describe these components in more detail.

Memory

Memory is an essential component in any computer. Let's look at what it consists of and how the computer works with it.

FIGURE 1.3

Components of a Computer



memory cell an individual storage location in memory

address of a memory cell the relative position of a memory cell in the computer's main memory

contents of a memory cell the information stored in a memory cell, either a program instruction or data

stored program concept a computer's ability to store program instructions in main memory for execution

Anatomy of Memory Imagine the memory of a computer as an ordered sequence of storage locations called **memory cells** (Fig. 1.4). To store and access information, the computer must have some way of identifying the individual memory cells. Therefore, each memory cell has a unique **address** that indicates its relative position in memory. Figure 1.4 shows a computer memory consisting of 1000 memory cells with addresses 0 through 999. Most computers, however, have millions of individual memory cells, each with its own address.

The data stored in a memory cell are called the **contents** of the cell. Every memory cell always has some contents, although we may have no idea what they are. In Fig. 1.4, the contents of memory cell 3 are the number −26 and the contents of memory cell 4 are the letter H.

Although not shown in Fig. 1.4, a memory cell can also contain a program instruction. The ability to store programs as well as data is called the **stored program concept**: A program's instructions must be stored in main memory before they can be executed. We can change the computer's operation by storing a different program in memory.

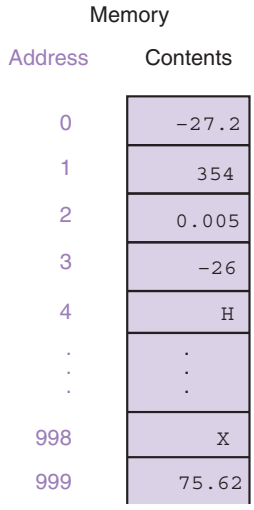


FIGURE 1.4

1000 Memory Cells in Main Memory

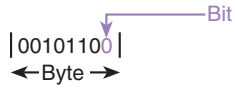


FIGURE 1.5

Relationship Between a Byte and a Bit

byte the amount of storage required to store a single character

bit a binary digit; a 0 or a 1

data storage setting the individual bits of a memory cell to 0 or 1, destroying its previous contents

data retrieval copying the contents of a particular memory cell to another storage area

random access memory (RAM) the part of main memory that temporarily stores programs, data, and results

Bytes and Bits A memory cell is actually a grouping of smaller units called bytes. A **byte** is the amount of storage required to store a single character, such as the letter H in memory cell 4 of Fig. 1.4. The number of bytes a memory cell can contain varies from computer to computer. A byte is composed of even smaller units of storage called bits (Fig. 1.5). The term **bit**, derived from the words **b**inary dig**it**, is the smallest element a computer can deal with. Binary refers to a number system based on two numbers, 0 and 1, so a bit is either a 0 or a 1. Generally, there are 8 bits to a byte.

Storage and Retrieval of Information in Memory Each value in memory is represented by a particular pattern of 0s and 1s. A computer can either store a value or retrieve a value. To **store** a value, the computer sets each bit of a selected memory cell to either 0 or 1, destroying the previous contents of the cell in the process. To **retrieve** a value from a memory cell, the computer copies the pattern of 0s and 1s stored in that cell to another storage area for processing; the copy operation does not destroy the contents of the cell whose value is retrieved. This process is the same regardless of the kind of information—character, number, or program instruction—to be stored or retrieved.

Main Memory Main memory stores programs, data, and results. Most computers have two types of main memory: **random access memory (RAM)**, which offers

read-only memory (ROM) the part of main memory that permanently stores programs or data

volatile memory memory whose contents disappear when the computer is switched off

temporary storage of programs and data, and read-only memory (ROM), which stores programs or data permanently. RAM temporarily stores programs while they are being executed (carried out) by the computer. It also temporarily stores such data as numbers, names, and even pictures while a program is manipulating them. RAM is usually volatile memory, which means that everything in RAM will be lost when the computer is switched off.

ROM, on the other hand, stores information permanently within the computer. The computer can retrieve (or read), but cannot store (or write) information in ROM, hence its name, read-only. Because ROM is not volatile, the data stored there do not disappear when the computer is switched off. Start-up instructions and other critical instructions are burned into ROM chips at the factory. When we refer to main memory in this text, we mean RAM because that is the part of main memory that is normally accessible to the programmer.

Secondary Storage Devices Computer systems provide storage in addition to main memory for two reasons. First, computers need storage that is permanent or semipermanent so that information can be retained during a power loss or when the computer is turned off. Second, systems typically store more information than will fit in memory.

secondary storage units such as disks or flash drives that retain data even when the power to the drive is off

disk thin platter of metal or plastic on which data are represented by magnetized spots arranged in tracks

Figure 1.6 shows some of the most frequently encountered **secondary storage** devices and storage media. Most personal computers use two types of disk drives as their secondary storage devices—hard drives and optical drives. Hard **disks** are attached to their disk drives and are coated with a magnetic material. Each data bit is a magnetized spot on the disk, and the spots are arranged in concentric circles called tracks. The disk drive read/write head accesses data by moving across the spinning disk to the correct track and then sensing the spots as they move by. The hard disks in personal computers usually hold several hundred gigabytes (GB) of data, but clusters of hard drives that store data for an entire network may provide as much as several terabytes (TB) of storage (see Table 1.1).

optical drive device that uses a laser to access or store data on a CD or DVD or Blu-ray Disk

Many of today's personal computers are equipped with **optical drives** for storing and retrieving data on compact disks (CDs) or digital versatile disks (DVDs) that can be removed from the drive. A CD is a silvery plastic platter on which a laser records data as a sequence of tiny pits in a spiral track on one side of the disk. One CD can hold 680 MB of data. A DVD uses smaller pits packed in a tighter spiral. Some Blu-ray disks (high-density optical disks read with a blue-violet laser) can hold multiple layers of data—for a total capacity of 200 GB, sufficient storage for many hours of studio-quality video and multi-channel audio.

FIGURE 1.6

Secondary Storage Media

TABLE 1.1 Terms Used to Quantify Storage Capacities

Term	Abbreviation	Equivalent to	Comparison to Power of 10
Byte	B	8 bits	
Kilobyte	KB	1,024 (2^{10}) bytes	> 10^3
Megabyte	MB	1,048,576 (2^{20}) bytes	> 10^6
Gigabyte	GB	1,073,741,824 (2^{30}) bytes	> 10^9
Terabyte	TB	1,099,511,627,776 (2^{40}) bytes	> 10^{12}
Petabyte	PB	1,125,899,906,842,624 (2^{50}) bytes	> 10^{15}

flash drive device that plugs into USB port and stores data bits as trapped electrons

Flash drives such as the one pictured in Fig. 1.6 use flash memory packaged in small plastic cases about three inches long that can be plugged into any of a computer's USB (Universal Serial Bus) ports. Unlike hard drives and optical drives that must spin their disks for access to data, flash drives have no moving parts and all data transfer is by electronic signal only. In flash memory, bits are represented as electrons trapped in microscopic chambers of silicon dioxide. Typical USB flash drives store several GB of data, but drives holding up to one terabyte (1 TB) are also available.

file named collection of data stored on a disk

directory a list of the names of files stored on a disk

subdirectory a list of the names of files that relate to a particular topic

Information stored on a disk is organized into separate collections called **files**. One file may contain a C program. Another file may contain the data to be processed by that program (a *data file*). A third file could contain the results generated by a program (an *output file*). The names of all files stored on a disk are listed in the disk's **directory**. This directory may be broken into one or more levels of subdirectories or folders, where each **subdirectory** stores the names of files that relate to the same general topic. For example, you might have separate subdirectories of files that contain homework assignments and programs for each course you are taking this semester. The details of how files are named and grouped in directories vary with each computer system. Follow the naming conventions that apply to your system.

Central Processing Unit

central processing unit (CPU) coordinates all computer operations and performs arithmetic and logical operations on data

The **central processing unit (CPU)** or **processor** has two roles: coordinating all computer operations and performing arithmetic and logical operations on data. The processor follows the instructions contained in a computer program to determine which operations should be carried out and in what order. It then transmits coordinating control signals to the other computer components. For example, if the instruction requires scanning a data item, the CPU sends the necessary control signals to the input device.

fetching an instruction retrieving an instruction from main memory

To process a program stored in main memory, the CPU retrieves each instruction in sequence (called **fetching an instruction**), interprets the instruction to determine what should be done, and then retrieves any data needed to carry out

that instruction. Next, the CPU performs the actual manipulation, or processing, of the data it retrieved. The CPU stores the results in main memory.

The CPU can perform such arithmetic operations as addition, subtraction, multiplication, and division. The CPU can also compare the contents of two memory cells (for example, Which contains the larger value? Are the values equal?) and make decisions based on the results of that comparison.

register high-speed memory location inside the CPU

The circuitry of a modern CPU is housed in a single integrated circuit or chip, millions of miniature circuits manufactured in a sliver of silicon. A processor's current instruction and data values are stored temporarily inside the CPU in special high-speed memory locations called **registers**.

multiprocessor a computer with more than one CPU

Some **multiprocessor** computers have multiple CPU chips or a multi-core processor (a single chip containing multiple CPUs). These computers are capable of faster speeds because they can process different sets of instructions at the same time.

Input/Output Devices

We use *input/output (I/O) devices* to communicate with the computer. Specifically, they allow us to enter data for a computation and to observe the results of that computation.

You will be using a *keyboard* as an input device and a *monitor* (display screen) as an output device. When you press a letter or digit key on a keyboard, that character is sent to main memory and is also displayed on the screen at the position of the **cursor**, a moving place marker (often a blinking line or rectangle). A computer keyboard has keys for letters, numbers, and punctuation marks plus some extra keys for performing special functions. The twelve **function keys** along the top row of the keyboard are labeled F1 through F12. The activity performed when you press a function key depends on the program currently being executed; that is, pressing F1 in one program will usually not produce the same results as pressing F1 in another program. Other special keys enable you to delete characters, move the cursor, and "enter" a line of data you typed at the keyboard.

cursor a moving place marker that appears on the screen

function keys special keyboard keys used to select a particular operation; operation selected depends on program being used

Other common input devices provide pointing capability to allow you to select an operation. Moving a **mouse** around on your desktop or dragging your finger across a **touchpad** moves the *cursor* (normally an arrow, vertical line, or a small rectangle) displayed on the screen. You select an operation by moving the cursor to a word or **icon** (picture) that represents the computer operation you wish to perform and then clicking a button to activate the operation selected.

mouse, touchpad input devices that move a cursor on the computer screen to select an operation

icon a picture representing a computer operation

A screen provides a temporary display of information. If you want **hard copy** (a printed version) of some information, you must send that information to an output device called a *printer* .

hard copy a printed version of information

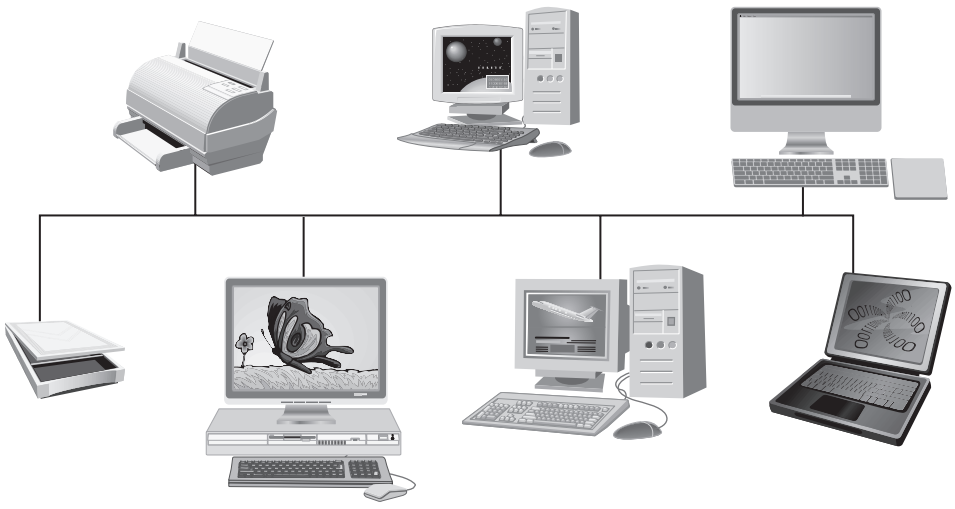
Computer Networks

local area network (LAN) computers, printers, scanners, and storage devices connected by cables for intercommunication

The explosion we are experiencing in worldwide information access is primarily due to the fact that computers are now linked together in networks so they can communicate with one another. In a **local area network (LAN)**, computers and other

FIGURE 1.7

Local Area Network



devices in a building are connected by cables or a wireless network, allowing them to share information and resources such as printers, scanners, and secondary storage devices (Fig. 1.7). A computer that controls access to a secondary storage device such as a large hard disk is called a **file server**.

file server the computer in a network that controls access to a secondary storage device such as a hard disk

Local area networks can be connected to other LANs using the same technology as telephone networks. Communications over intermediate distances use phone lines, fiber-optics cables or wireless technology, and long-range communications use either phone lines or microwave signals that may be relayed by satellite (Fig. 1.8).

A network that links many individual computers and local area networks over a large geographic area is called a **wide area network (WAN)**. The most well-known WAN is the Internet, a network of university, corporate, government, and public-access networks. The Internet is a descendant of the computer network designed by the U.S. Defense Department's 1969 ARPAnet project. The goal of the project was to create a computer network that could continue to operate even if partially destroyed. The most widely used aspect of the Internet is the **World Wide Web (WWW)**, the universe of Internet-accessible resources that are navigable through the use of a **graphical user interface (GUI)**.

wide area network (WAN) a network such as the Internet that connects computers and LANs over a large geographic area

World Wide Web (WWW) a part of the Internet whose graphical user interfaces make associated network resources easily navigable

graphical user interface (GUI) pictures and menus displayed to allow user to select commands and data

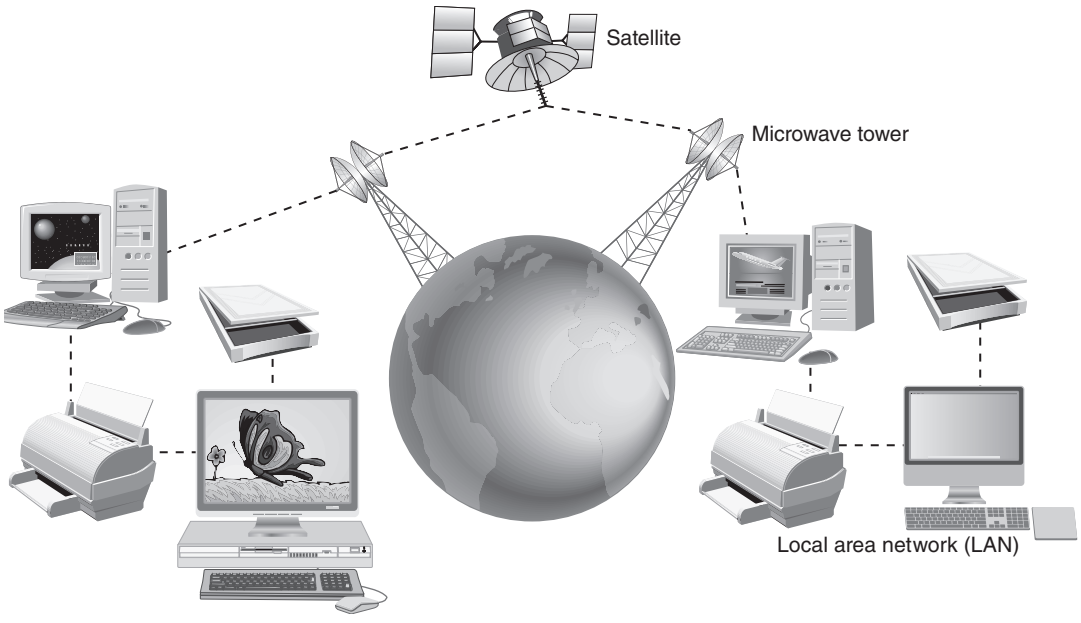
If you have a computer with a modem, you can connect to the information superhighway through a telephone line, television or fiber-optic cable, or through wireless or satellite communications. A **modem** (**mo**dulator/**dem**odulator) converts binary computer data into audio tones that can be transmitted to another computer over a normal telephone circuit. At the computer on the receiving end, another modem converts the audio tones back to binary data.

modem device that converts binary data into analog signals that can be transmitted between computers over telephone lines

Early modems for telephone lines transmitted at only 300 baud (300 bits per second). Today's modems can transmit over 150 megabits per second, if attached to a line that handles such speeds.

FIGURE 1.8

A Wide Area Network with Satellite Relays of Microwave Signals



cable Internet access two-way high-speed transmission of Internet data through two of the hundreds of channels available over the coaxial cable that carries cable television signals

Cable Internet access brings Internet data to your computer at speeds of several billion bits per second using the same coaxial cable that carries cable TV. Wireless and satellite communications provide data speeds comparable to cable.

EXERCISES FOR SECTION 1.2

Self-Check

1. If a computer executes instructions to sum the contents of memory cells 2 and 999 in Fig. 1.4 and store the result in cell 0, what would then be the contents of cells 0, 2, and 999?
2. One bit can have two values, 0 or 1. A combination of 2 bits can have four values: 00, 01, 10, and 11. List all of the values you can form with a combination of 3 bits. Do the same for 4 bits.
3. List the following in order of smallest to largest: byte, bit, WAN, main memory, memory cell, LAN, secondary storage.

1.3 Computer Software

In the previous section, we surveyed the components of a computer system, components referred to collectively as hardware. We also studied the fundamental operations that allow a computer to accomplish tasks: repeated fetching and execution of instructions. In this section, we focus on these all-important lists of instructions called computer programs or computer software. We will consider first the software that makes the hardware friendly to the user. We will then look at the various levels of computer languages in which software is written and at the process of creating and running a new program.

Operating System

operating system (OS) software that controls interaction of user and computer hardware and that manages allocation of computer resources

The collection of computer programs that control the interaction of the user and the computer hardware is called the **operating system (OS)**. The operating system of a computer is often compared to the conductor of an orchestra, for it is the software that is responsible for directing all computer operations and managing all computer resources. Usually, part of the operating system is stored permanently in a read-only memory (ROM) chip so that it is available as soon as the computer is turned on. A computer can look at the values in read-only memory, but cannot write new values to the chip. The ROM-based portion of the OS contains the instructions necessary for loading into memory the rest of the operating system code, which typically resides on a disk. Loading the operating system into memory is called **booting the computer**.

booting a computer loading the operating system from disk into memory

Here is a list of some of the operating system's many responsibilities:

1. Communicating with the computer user: receiving commands and carrying them out or rejecting them with an error message.
2. Managing allocation of memory, of processor time, and of other resources for various tasks.
3. Collecting input from the keyboard, touchpad, mouse, and other input devices, and providing this data to the currently running program.
4. Conveying program output to the screen, printer, or other output device.
5. Accessing data from secondary storage.
6. Writing data to secondary storage.

In addition to these responsibilities, the operating system of a computer with multiple users must verify each individual's right to use the computer and must ensure that each user can access only data for which he or she has proper authorization. When multiple programs are running on a computer concurrently, the OS makes sure that each one accesses only the memory and other resources specifically allocated to it.

An operating system that uses a command-line interface displays a brief message, called a *prompt*, that indicates its readiness to receive input, and the user then types a command at the keyboard. Figure 1.9 shows an entry of a UNIX command (`ls temp/misc`) requesting a list of the names of all the files (`Gridvar.c,`

FIGURE 1.9 Entering a UNIX Command for Directory Display

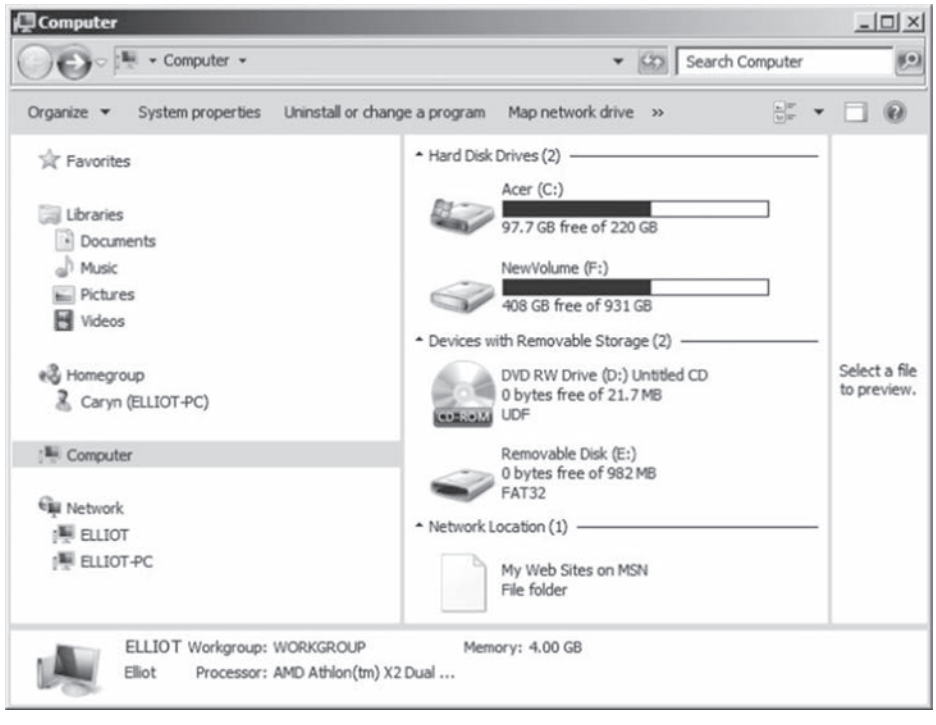
```
1. mycomputer:~> ls temp/misc
2. Gridvar.c      Gridvar.exe      Gridok.txt
3.
4. mycomputer:~>
```

`Gridvar.exe` , `Gridok.txt`) in subdirectory `misc` of directory `temp`. In this case, the prompt is `mycomputer:~>` (In this figure, and in all subsequent figures showing program runs, input typed by the user is shown in color to distinguish it from computer-generated text.)

Most modern operating systems have a graphical user interface that provides the user with a system of icons and menus. To issue commands, the user moves the mouse or touchpad cursor to point to the appropriate icon or menu selection and clicks a button once or twice. Figure 1.10 shows the window that

FIGURE 1.10

Accessing Secondary Storage Devices Through Windows



pops up in Microsoft Windows 7 when you left-click on the Start icon and then left-click on Computer. You can view the directories of the hard drive (C:), backup drive (F:), optical drive (D:), or flash drive (E:) by double-clicking the appropriate icon.

Application Software

application software used for a specific task such as word processing, accounting, database management, playing a game, or checking the weather forecast

Application programs are developed to assist a computer user in accomplishing specific tasks. For example, a word-processing application such as Microsoft Word or OpenOffice.org Writer helps to create a document, a spreadsheet application such as Microsoft Office Excel helps to automate tedious numerical calculations and to generate charts that depict data, and a database management application such as Microsoft Office Access or Oracle assists in data storage and quick keyword-based access to large collections of records.

install make an application available on a computer by copying it to the computer's hard drive

Computer users typically purchase application software by downloading files from the Internet and **install** the software by following on-screen directions that copy the programs to the hard disk. When buying software, you must always check that the program you are purchasing is compatible with both the operating system and the computer hardware you plan to use. We have already discussed some of the differences among operating systems; now we will investigate the different languages understood by different processors.

Computer Languages

Developing new software requires writing lists of instructions for a computer to execute. However, software developers rarely write in the language directly understood by a processor, since this **machine language** is a collection of binary numbers. Another drawback of machine language is that it is not standardized: There is a different machine language for every type of CPU. This same drawback also applies to the somewhat more readable **assembly language**, a language in which computer operations are represented by mnemonic codes rather than binary numbers and variables can be given names rather than binary memory addresses. Table 1.2 shows a tiny machine language program fragment that adds two numbers and the equivalent fragment in assembly language. Notice that each assembly language instruction corresponds to exactly one machine instruction: The assembly language memory cells labeled A and B are space for variables; they are not instructions. The symbol ? indicates that we do not know the contents of the memory cells with addresses 00000101 and 00000110.

machine language binary number codes understood by a specific CPU

assembly language mnemonic codes that correspond to machine language instructions

high-level language machine-independent programming language that combines algebraic expressions and English symbols

To write programs that are independent of the processor on which they will be executed, software designers use **high-level languages** that combine algebraic expressions and symbols taken from English. For example, the machine/assembly

TABLE 1.2 A Machine Language Program Fragment and Its Assembly Language Equivalent

Memory Addresses	Machine Language Instructions	Assembly Language Instructions
00000000	00000000	CLA
00000001	00010101	ADD A
00000010	00010110	ADD B
00000011	00110101	STA A
00000100	01110111	HLT
00000101	?	A ?
00000110	?	B ?

language program fragment shown in Table 1.2 would be a single statement in a high-level language:

```
a = a + b;
```

This statement means "add the values of variables `a` and `b`, and store the result in variable `a` (replacing `a`'s previous value)."

There are many high-level languages available. Table 1.3 lists some of the most widely used ones along with some information about their origins and the application areas that first popularized them. Although programmers find it far easier to express problem solutions in high-level languages, there remains the problem that computers do NOT understand these languages. Thus, before a high-level language program can be executed, it must first be translated into the target processor's machine language. The program that does this translation is called a **compiler**. Figure 1.11 illustrates the role of the compiler in the process of developing and testing a high-level language program. Both the input to and the output from the compiler (when it is successful) are programs. The input to the compiler is a **source file** containing the text of a high-level language program. The software developer creates this file by using a word processor or editor. The format of the source file is text, which means that it is a collection of character codes. For example, you might type a program into a file called `myprog.c`. The compiler will scan this source file, checking the program to see if it follows the high-level language's **syntax** (grammar) rules. If the program is syntactically correct, the compiler saves in an **object file** the machine language instructions that carry out the program's purpose. For program `myprog.c`, the object file created might be named `myprog.obj`. Notice that this file's format is binary. This means that you should not send it to a printer, display it on your screen, or try to work with it in a word processor

compiler software that translates a high-level language program into machine language

source file file containing a program written in a high-level language; the input for a compiler

syntax grammar rules of a programming language

object file file of machine language instructions that is the output of a compiler

TABLE 1.3 High-Level Languages

Language	*Origin / Application Area*
C	Systems programming language originally used in the development of the UNIX operating system; now used in a wide range of applications
C++	Adds object-oriented features to C; used in a wide range of applications including game development
Objective C	Adds to C object capabilities and messaging; used by Apple for systems programming and iPhone application software
Perl	Used in creation of interactive web pages, graphics, and network programming
Python	General-purpose high-level language that emphasizes readability of code
Java	General-purpose language used in wide range of applications including Web programming and Android application software
JavaScript	Used in web browsers both for client-side interaction with user and in web server programs
PHP	Designed for web development; code is interpreted by a program running on a web server to generate a dynamic web page
C#	Adds object-oriented and component-oriented features to C; developed by Microsoft for its .NET initiative

because it will appear to be meaningless garbage to a word processor, printer, or screen. If the source program contains syntax errors, the compiler lists these errors but does not create an object file. The developer must return to the word processor, correct the errors, and recompile the program.

Although an object file contains machine instructions, not all of the instructions are complete. High-level languages provide the software developer with many named chunks of code for operations that the developer will likely need. Almost all high-level language programs use at least one of these chunks of code called *functions* that reside in other object files available to the system. The **linker** program combines these prefabricated functions with the object file, creating a complete machine language program that is ready to run. For your sample program, the linker might name the executable file it creates `myprog.exe`.

linker software that combines object files and resolves cross-references to create an executable machine language program

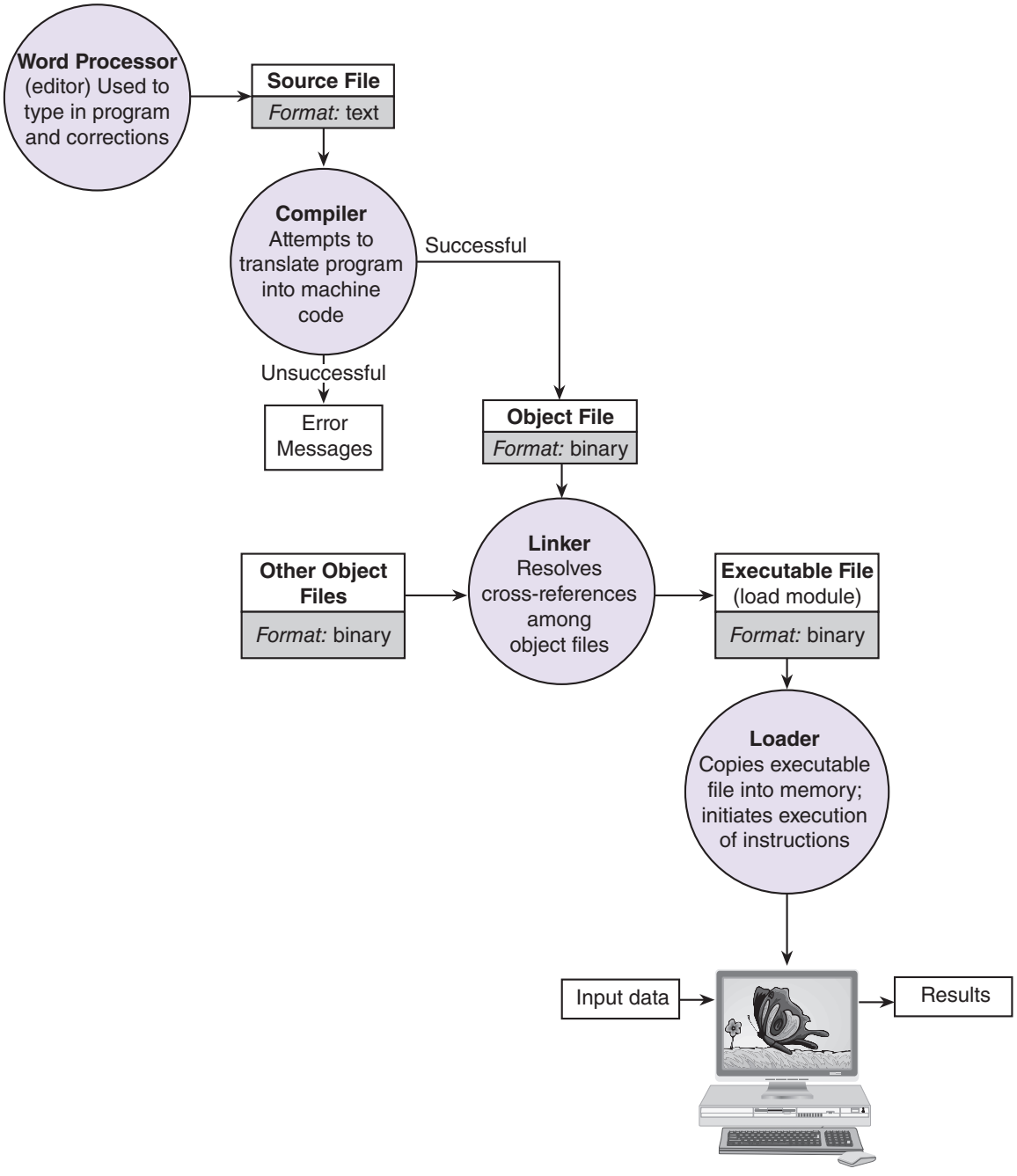
As long as `myprog.exe` is just stored on your disk, it does nothing. To run it, the loader must copy all its instructions into memory and direct the CPU to begin execution with the first instruction. As the program executes, it takes input data from one or more sources and sends results to output and/or secondary storage devices.

Some computer systems require the user to ask the OS to carry out separately each step illustrated in Fig. 1.11. However, most high-level language compilers are sold as part of an **integrated development environment (IDE)**, a package that combines a simple word processor with a compiler, linker, and loader. Such environments give the developer menus from which to select the next step, and if the developer tries a step

integrated development environment (IDE) software package combining a word processor, compiler, linker, loader, and tools for finding errors

FIGURE 1.11

Entering, Translating, and Running a High-Level Language Program



that is out of sequence, the environment simply fills in the missing steps automatically. An IDE will frequently use the term *build* to mean "compile and link."

The user of an integrated development environment should be aware that the environment may not automatically save to disk the source, object, and executable files. Rather, it may simply leave these versions of the program in memory. Such an approach saves the expenditure of time and disk space needed to make copies and keeps the code readily available in memory for application of the next step in the translation/execution process. However, the developer can risk losing the only copy of the source file in the event of a power outage or serious program error. To prevent such a loss when using an IDE, be sure to explicitly save the source file to disk after every modification before attempting to run the program.

Executing a Program

To execute a machine language program, the processor must examine each program instruction in memory and send out the command signals required to carry out the instruction. Although the instructions normally are executed in sequence, as we will discuss later, it is possible to have the CPU skip over some instructions or execute some instructions more than once.

input data the data values that are scanned by a program

program output the lines displayed by a program

During execution, data can be entered into memory and manipulated in some specified way. Special program instructions are used for entering or scanning a program's data (called **input data**) into memory. After the input data have been processed, instructions for displaying or printing values in memory can be executed to display the program results. The lines displayed by a program are called the **program output**.

Let's use the situation described in Fig. 1.12—executing a water bill program stored in memory—as an example. The first step of the program scans into memory data that describe the amount of water used. In step 2, the program manipulates the data and stores the results of the computations in memory. In the final step, the computational results are displayed as a water bill.

EXERCISES FOR SECTION 1.3

Self-Check

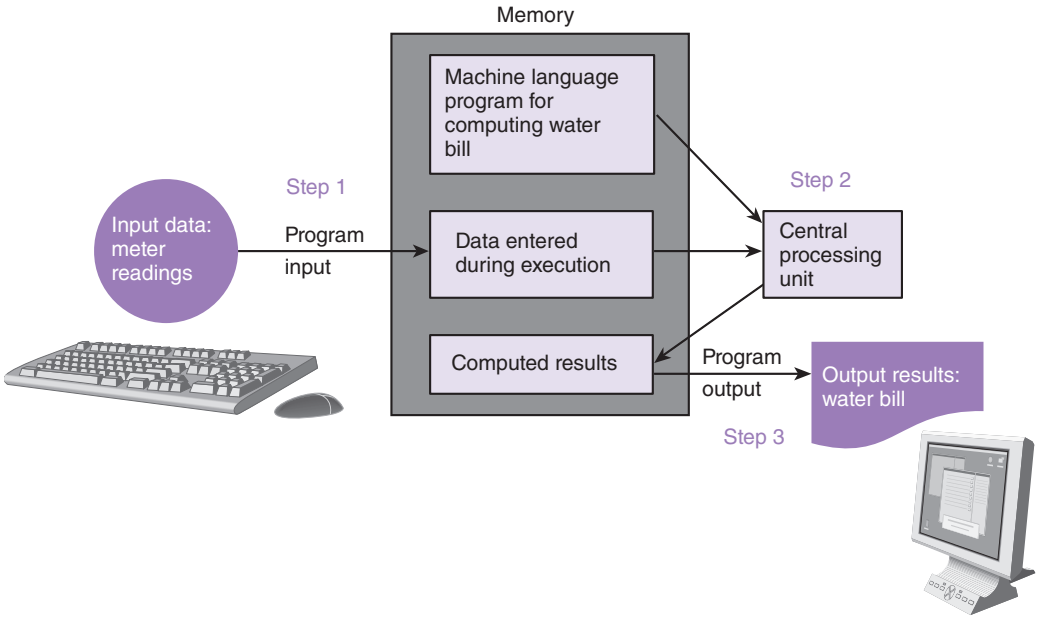
1. What do you think these five high-level language statements mean?

```
x = a + b + c;    x = y / z;    d = c - b + a;
z = z + 1;    kelvin = celsius + 273.15;
```

2. List two reasons why it would be preferable to write a program in C rather than in machine language.

FIGURE 1.12

Flow of Information During Program Execution



3. Would a syntax error be found in a source program or an object program? What system program would find a syntax error if one existed?
4. Explain the differences among the source program, the object program, and an executable program. Which do you create, and which does the compiler create? Which does the linker create?

1.4 The Software Development Method

Programming is a problem-solving activity. If you are a good problem solver, you have the potential to become a good programmer. Therefore, one goal of this book is to help you improve your problem-solving ability. Problem-solving methods are covered in many subject areas. Business students learn to solve problems with a *systems approach* while engineering and science students use the *engineering and scientific method.* Programmers use the *software development method.*

Software Development Method

1. Specify the problem requirements.
2. Analyze the problem.

3. Design the algorithm to solve the problem.
4. Implement the algorithm.
5. Test and verify the completed program.
6. Maintain and update the program.

PROBLEM

Specifying the problem requirements forces you to state the problem clearly and unambiguously and to gain a clear understanding of what is required for its solution. Your objective is to eliminate unimportant aspects and zero in on the root problem. This goal may not be as easy to achieve as it sounds. You may find you need more information from the person who posed the problem.

ANALYSIS

Analyzing the problem involves identifying the problem (a) *inputs,* that is, the data you have to work with; (b) *outputs*, that is, the desired results; and (c) any additional requirements or constraints on the solution. At this stage, you should also determine the required format in which the results should be displayed (for example, as a table with specific column headings) and develop a list of problem variables and their relationships. These relationships may be expressed as formulas.

If steps 1 and 2 are not done properly, you will solve the wrong problem. Read the problem statement carefully, first, to obtain a clear idea of the problem and second, to determine the inputs and outputs. You may find it helpful to underline phrases in the problem statement that identify the inputs and outputs, as in the problem statement below.

Compute and display the total cost of apples given the number of pounds of apples purchased and the cost per pound of apples.

Next, summarize the information contained in the underlined phrases:

Problem Inputs

```
quantity of apples purchased (in pounds)
cost per pound of apples (in dollars per pound)
```

Problem Output

```
total cost of apples (in dollars)
```

Once you know the problem inputs and outputs, develop a list of formulas that specify relationships between them. The general formula

$$\textit{Total cost} = \textit{Unit cost} \times \textit{Number of units}$$

computes the total cost of any item purchased. Substituting the variables for our particular problem yields the formula

$$\textit{Total cost of apples} = \textit{Cost per pound} \times \textit{Pounds of apples}$$

abstraction the process of modeling a problem by extracting the essential variables and their relationships

In some situations, you may need to make certain assumptions or simplifications to derive these relationships. This process of modeling a problem by extracting the essential variables and their relationships is called **abstraction**.

DESIGN

algorithm a list of steps for solving a problem

top-down design breaking a problem into its major subproblems and then solving the subproblems

Designing the algorithm to solve the problem requires you to develop a list of steps called an **algorithm** to solve the problem and then to verify that the algorithm solves the problem as intended. Writing the algorithm is often the most difficult part of the problem-solving process. Don't attempt to solve every detail of the problem at the beginning; instead, discipline yourself to use top-down design. In **top-down design** (also called *divide and conquer*), you first list the major steps, or subproblems, that need to be solved. Then you solve the original problem by solving each of its subproblems. Most computer algorithms consist of at least the following subproblems.

ALGORITHM FOR A PROGRAMMING PROBLEM

1. Get the data.
2. Perform the computations.
3. Display the results.

Once you know the subproblems, you can attack each one individually. For example, the perform-the-computations step may need to be broken down into a more detailed list of steps through a process called **stepwise refinement**.

stepwise refinement development of a detailed list of steps to solve a particular step in the original algorithm

You may be familiar with top-down design if you use an outline when writing a term paper. Your first step is to create an outline of the major topics, which you then refine by filling in subtopics for each major topic. Once the outline is complete, you begin writing the text for each subtopic.

desk checking the step-by-step simulation of the computer execution of an algorithm

Desk checking is an important part of algorithm design that is often overlooked. To **desk check** an algorithm, you must carefully perform each algorithm step (or its refinements) just as a computer would and verify that the algorithm works as intended. You'll save time and effort if you locate algorithm errors early in the problem-solving process.

IMPLEMENTATION

Implementing the algorithm (step 4 in the software development method) involves writing it as a program. You must convert each algorithm step into one or more statements in a programming language.

TESTING

Testing and verifying the program requires testing the completed program to verify that it works as desired. Don't rely on just one test case. Run the program several times using different sets of data to make sure that it works correctly for every situation provided for in the algorithm.

MAINTENANCE

Maintaining and updating the program involves modifying a program to remove previously undetected errors and to keep it up-to-date as government regulations or company policies change. Many organizations maintain a program for five years or more, often after the programmers who originally coded it have left or moved on to other positions.

A disciplined approach is essential if you want to create programs that are easy to read, understand, and maintain. You must follow accepted program style guidelines (which will be stressed in this book) and avoid tricks and programming shortcuts.

Caution: Failure Is Part of the Process

Although having a step-by-step approach to problem solving is helpful, we must avoid jumping to the conclusion that if we follow these steps, we are *guaranteed* a correct solution the first time, every time. The fact that verification is so important implies an essential truth of problem solving: The first (also the second, the third, or the twentieth) attempt at a solution *may be wrong.* Probably the most important distinction between outstanding problem solvers and less proficient ones is that outstanding problem solvers are not discouraged by initial failures. Rather, they see the faulty and near-correct early solutions as a means of gaining a better understanding of the problem. One of the most inventive problem solvers of all time, Thomas Edison, is noted for his positive interpretation of the thousands of failed experiments that contributed to his incredible record of inventions. His friends report that he always saw those failures in terms of the helpful data they yielded about what did *not* work.

EXERCISES FOR SECTION 1.4

Self-Check

1. List the steps of the software development method.
2. In which phase is the algorithm developed? In which phase do you identify the problem inputs and outputs?

1.5 Applying the Software Development Method

Throughout this book, we use the first five steps of the software development method to solve programming problems. These example problems, presented as Case Studies, begin with a *problem statement.* As part of the problem *analysis,* we identify the data requirements for the problem, indicating the problem inputs and the desired outputs. Next, we *design* and refine the initial algorithm. Finally, we *implement* the algorithm as a C program. We also provide a sample run of the program and discuss how to *test* the program.

We walk you through a sample case study next. This example includes a running commentary on the process, which you can use as a model in solving other problems.

CASE STUDY Converting Miles to Kilometers

PROBLEM

Your summer surveying job requires you to study some maps that give distances in kilometers and some that use miles. You and your coworkers prefer to deal in metric measurements. Write a program that performs the necessary conversion.

ANALYSIS

The first step in solving this problem is to determine what you are asked to do. You must convert from one system of measurement to another, but are you supposed to convert from kilometers to miles, or vice versa? The problem states that you prefer to deal in metric measurements, so you must convert distance measurements in miles to kilometers. Therefore, the problem input is **distance in miles** and the problem output is **distance in kilometers**. To write the program, you need to know the relationship between miles and kilometers. Consulting a metric table shows that one mile equals 1.609 kilometers.

The data requirements and relevant formulas are listed below. `miles` identifies the memory cell that will contain the problem input and `kms` identifies the memory cell that will contain the program result, or the problem output.

DATA REQUIREMENTS

Problem Input

```
miles    /* the distance in miles*/
```

Problem Output

```
kms    /* the distance in kilometers */
```

Relevant Formula

1 mile = 1.609 kilometers

DESIGN

Next, formulate the algorithm that solves the problem. Begin by listing the three major steps, or subproblems, of the algorithm.

ALGORITHM

1. Get the distance in miles.
2. Convert the distance to kilometers.
3. Display the distance in kilometers.

Now decide whether any steps of the algorithm need further refinement or whether they are perfectly clear as stated. Step 1 (getting the data) and step 3 (displaying a value) are basic steps and require no further refinement. Step 2 is fairly straightforward, but some detail might help:

Step 2 Refinement

2.1 The distance in kilometers is 1.609 times the distance in miles.

We list the complete algorithm with refinements below to show you how it all fits together. The algorithm resembles an outline for a term paper. The refinement of step 2 is numbered as step 2.1 and is indented under step 2.

ALGORITHM WITH REFINEMENTS

1. Get the distance in miles.
2. Convert the distance to kilometers.
   2.1 The distance in kilometers is 1.609 times the distance in miles.
3. Display the distance in kilometers.

Let's desk check the algorithm before going further. If step 1 gets a distance of 10.0 miles, step 2.1 would convert it to 1.609×10.00 or 16.09 kilometers. This correct result would be displayed by step 3.

IMPLEMENTATION

To implement the solution, you must write the algorithm as a C program. To do this, you must first tell the C compiler about the problem data requirements—that

FIGURE 1.13 Miles-to-Kilometers Conversion Program

```
1.  /*
2.   * Converts distance in miles to kilometers.
3.   */
4.  #include <stdio.h>             /* printf, scanf definitions */
5.  #define KMS_PER_MILE 1.609     /* conversion constant       */
6.
7.  int
8.  main(void)
9.  {
10.       double miles,  /* input - distance in miles.      */
11.              kms;    /* output - distance in kilometers */
12.
13.       /* Get the distance in miles. */
14.       printf("Enter the distance in miles> ");
15.       scanf("%lf", &miles);
16.
17.       /* Convert the distance to kilometers. */
18.       kms = KMS_PER_MILE * miles;
19.
20.       /* Display the distance in kilometers. */
21.       printf("That equals %f kilometers.\n", kms);
22.
23.       return (0);
24. }
```

Sample Run

```
Enter the distance in miles> 10.00
That equals 16.090000 kilometers.
```

is, what memory cell names you are using and what kind of data will be stored in each memory cell. Next, convert each algorithm step into one or more C statements. If an algorithm step has been refined, you must convert the refinements, not the original step, into C statements.

Figure 1.13 shows the C program along with a sample execution or run. For easy identification, the program statements corresponding to algorithm steps are in color as is the input data typed in by the program user. Don't worry about understanding the details of this program yet. We explain the program in the next chapter.

TESTING

How do you know the sample run is correct? You should always examine program results carefully to make sure that they make sense. In this run, a distance of 10.0 miles is converted to 16.09 kilometers, as it should be. To verify that the program works properly, enter a few more test values of miles. You don't need to try more than a few test cases to verify that a simple program like this is correct.

EXERCISES FOR SECTION 1.5

Self-Check

1. Change the algorithm for the metric conversion program to convert distance in kilometers to miles.
2. List the data requirements, formulas, and algorithm for a program that converts a volume from quarts to liters.

1.6 Professional Ethics for Computer Programmers

We end this introductory chapter with a discussion of professional ethics for computer programmers. Like other professionals, computer programmers and software system designers (called software engineers) need to follow certain standards of professional conduct.

Privacy and Misuse of Data

As part of their jobs, programmers may have access to large data banks or databases containing sensitive information on financial transactions, personnel, or private health data, information that is classified as "secret" or "top secret." Programmers should always behave in a socially responsible manner and not retrieve information that they are not entitled to see. They should not use information to which they are given access for their own personal gain, or do anything that would be considered illegal, unethical, or harmful to others. Just as doctors and lawyers must keep patient information confidential, programmers must respect an individual's rights to privacy.

computer theft (computer fraud) Illegally obtaining money by falsifying information in a computer database

A programmer who changes information in a database containing financial records for his or her own personal gain—for example, changes the amount of money in a bank account—is guilty of **computer theft** or **computer fraud**. This is a felony that can lead to fines and imprisonment.

Computer Hacking

You may have heard about "computer hackers" who break into secure data banks by using their own computer to call the computer that controls access to the data bank. Classified or confidential information retrieved in this way has been sold to intelligence agencies of other countries. Other hackers have tried to break into computers to retrieve information for their own amusement or as a prank, to gain media attention, or just to demonstrate that they can do it. Regardless of the intent, this activity is illegal, and the government will prosecute anyone who does it. Your university probably addresses this kind of activity in your student handbook. The punishment is likely similar to penalties for other criminal activity, because that is exactly what hacking is—a crime.

virus Code attached to another program that spreads through a computer's disk memory, disrupting the computer or erasing information

worm A virus that can disrupt a network by replicating itself on other network computers

Another illegal activity sometimes practiced by hackers is attaching harmful code, called a **virus**, to another program so that the virus code copies itself throughout a computer's disk memory. A virus can cause sporadic activities to disrupt the operation of the host computer—for example, unusual messages may appear on the screen at certain times—or cause the host computer to erase portions of its own disk memory, destroying valuable information and programs. Viruses are spread from one computer to another in various ways—for example, if you copy a file that originated on another computer that has a virus, or if you open an e-mail message that is sent from an infected computer. A computer **worm** is a virus that can replicate itself on other network computers, causing these computers to send multiple messages over the network to disrupt its operation or shut it down. Certainly, data theft and virus propagation should not be considered harmless pranks; they are illegal and carry serious penalties.

Plagiarism and Software Piracy

Using someone else's programs without permission is also unprofessional behavior. Although it is certainly permissible to use modules from libraries that have been developed for reuse by their own company's programmers, you cannot use another programmer's personal programs or programs from another company without getting permission beforehand. Doing so could lead to a lawsuit, with you or your company having to pay damages.

Modifying another student's code and submitting it as your own is a fraudulent practice—specifically, plagiarism—and is no different than copying paragraphs of information from a book or journal article and calling it your own. Most universities have severe penalties for plagiarism that may include failing the course and/or being dismissed from the university. Be aware that even if you modify the code slightly or substitute your own comments or different variable names, you are still guilty of plagiarism if you are using another person's ideas and code. To avoid any question of plagiarism, find out beforehand your instructor's rules about working with others on a project. If group efforts are not allowed, make sure that you work independently and submit only your own code.

software piracy
Violating copyright agreements by illegally copying software for use in another computer

Many commercial software packages are protected by copyright laws against **software piracy**—the practice of illegally copying software for use on another computer. If you violate this law, your company or university can be fined heavily for allowing this activity to occur. Besides the fact that software piracy is against the law, using software copied from another computer increases the possibility that your computer will receive a virus. For all these reasons, you should read the copyright restrictions on each software package and adhere to them.

Misuse of a Computer Resource

Computer system access privileges or user account codes are private property. These privileges are usually granted for a specific purpose—for example, for work to be done in a particular course or for work to be done during the time you are a student at your university. The privilege should be protected; it should not be loaned to or shared with anyone else and should not be used for any purpose for which it was not intended. When you leave the institution, this privilege is normally terminated and any accounts associated with the privilege will be closed.

Computers, computer programs, data, and access (account) codes are like any other property. If they belong to someone else and you are not explicitly given permission to use them, then do not use them. If you are granted a use privilege for a specific purpose, do not abuse the privilege or it will be taken away.

Legal issues aside, it is important that we apply the same principles of right and wrong to computerized property and access rights as to all other property rights and privileges. If you are not sure about the propriety of something you want to do, ask first. As students and professionals in computing, we set an example for others. If we set a bad example, others are sure to follow.

EXERCISES FOR SECTION 1.6

Self-Check

1. Some computer users will not open an e-mail message unless they know the person who sent it. Why might someone adopt this policy?
2. Find out the penalty for plagiarism at your school.
3. Why is it a good policy to be selective about opening e-mail attachments?
4. Define the terms *virus* and *worm*.

Chapter Review

1. The basic components of a computer are main memory and secondary storage, the CPU, and input and output devices.
2. All data manipulated by a computer are represented digitally, as base 2 numbers composed of strings of the digits 0 and 1.

3. Main memory is organized into individual storage locations called memory cells.
   - Each memory cell has a unique address.
   - A memory cell is a collection of bytes; a byte is a collection of 8 bits.
   - A memory cell is never empty, but its initial contents may be meaningless to your program.
   - The current contents of a memory cell are destroyed whenever new information is stored in that cell.
   - Programs *must* be loaded into the memory of the computer before they can be executed.
   - Data cannot be manipulated by the computer until they are first stored in memory.
4. Information in secondary storage is organized into files: program files and data files. Secondary storage provides a low-cost means of storing large quantities of information in semipermanent form.
5. A CPU runs a computer program by repeatedly fetching and executing simple machine-code instructions.
6. Connecting computers in networks allows sharing of resources—local resources on LANs and worldwide resources on a WAN such as the Internet.
7. Programming languages range from machine language (meaningful to a computer) to high-level language (meaningful to a programmer).
8. Several system programs are used to prepare a high-level language program for execution. An editor enters a high-level language program into a file. A compiler translates a high-level language program (the source program) into machine language (the object program). The linker links this object program to other object files, creating an executable file, and the loader loads the executable file into memory. All of these programs are combined in an integrated development environment (IDE).
9. Through the operating system, you can issue commands to the computer and manage files.
10. Follow the first five steps of the software development method to solve programming problems: (1) specify the problem, (2) analyze the problem, (3) design the algorithm, (4) implement the algorithm, and (5) test and verify the completed program. Write programs in a consistent style that is easy to read, understand, and maintain.
11. Follow ethical standards of conduct in everything you do pertaining to computers. This means do not copy software that is copyright protected, do not hack into someone else's computer, do not send files that may be infected to others, and do not submit someone else's work as your own or lend your work to another student.

Quick-Check Exercises

1. A _____ translates a high-level language program into _____.
2. A(n) _____ provides access to system programs for program entry and compilation, database maintenance, web browsing, and so on.
3. Specify the correct order for these operations: execution, translation, linking, loading.
4. A high-level language program is saved on disk as a(n) _____ file.
5. The _____ finds syntax errors in the _____.
6. Before linking, a machine language program is saved on disk as a(n) _____ file.
7. After linking, a machine language program is saved on disk as a(n) _____ file.
8. Computer programs are _____ components of a computer system while a disk drive is _____.
9. A(n) _____ is a package that combines a simple word processor with a compiler, linker, and loader.
10. _____ have no moving parts, and all data transfers occur via electronic signal only, unlike in the case of hard drives, which must spin their disks for access to data.
11. On a magnetic disk, data are represented as _____ arranged in concentric tracks.
12. On a CD, DVD, or Blu-ray disk, data are represented as laser-written pits arranged in a(n) _____.
13. A(n) _____stores the names of files that are related to the same general topic.
14. A _____ converts binary computer data into audio tones that can be transmitted between computers over telephone lines.

Answers to Quick-Check Exercises

1. compiler, machine language
2. operating system
3. translation, linking, loading, execution
4. source
5. compiler, source file
6. object
7. executable
8. software, hardware
9. integrated development environment (IDE)
10. Flash drives
11. magnetized spots
12. spiral
13. subdirectory
14. modem

Review Questions

1. List at least three kinds of information stored in a computer.
2. List two functions of the CPU.
3. List two input devices, two output devices, and two secondary storage devices.
4. Describe three categories of programming languages.
5. What is a syntax error?
6. What processes are needed to transform a C program to a machine language program that is ready for execution?
7. Explain the relationship between memory cells, bytes, and bits.
8. Name three high-level languages and describe their application areas.
9. What are the differences between RAM and ROM?
10. What is the World Wide Web?
11. What is computer theft?
12. What is software piracy?
13. How many bytes equal 1 TB?

CHAPTER 2

Overview of C

CHAPTER OBJECTIVES

- To become familiar with the general form of a C program and the basic elements in a program
- To appreciate the importance of writing comments in a program
- To understand the use of data types and the differences between the data types `int`, `double`, and `char`
- To know how to declare variables
- To understand how to write assignment statements to change the values of variables
- To learn how C evaluates arithmetic expressions and how to write them in C
- To learn how to read data values into a program and to display results
- To understand how to write format strings for data entry and display
- To learn how to use redirection to enable the use of files for input/output
- To understand the differences between syntax errors, run-time errors, and logic errors, and how to avoid them and to correct them

This chapter introduces C—a high-level programming language developed in 1972 by Dennis Ritchie at AT&T Bell Laboratories. Because C was designed as a language in which to write the UNIX® operating system, it was originally used primarily for systems programming. Over the years, however, the power and flexibility of C, together with the availability of high-quality C compilers for computers of all sizes, have made it a popular language in industry for a wide variety of applications.

This chapter describes the elements of a C program and the types of data that can be processed by C. It also describes C statements for performing computations, for entering data, and for displaying results.

2.1 C Language Elements

One advantage of C is that it lets you write programs that resemble everyday English. Even though you do not yet know how to write your own programs, you can probably read and understand the program in Fig. 1.14. Figure 2.1 repeats this figure with the basic features of C highlighted. We identify them briefly below, and explain them in detail in Sections 2.2 to 2.4. The line numbers shown in all code figures are not part of the C programming.

Preprocessor Directives

preprocessor directive a C program line beginning with # that provides an instruction to the preprocessor

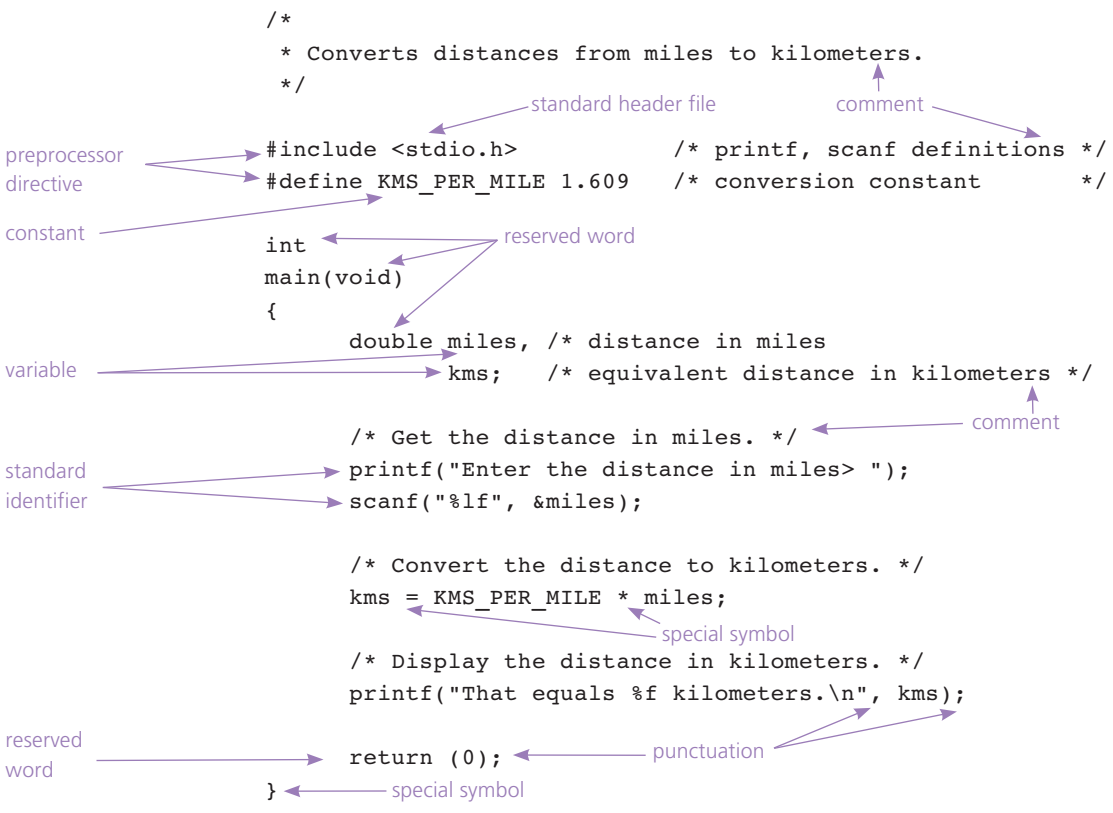
preprocessor a system program that modifies a C program prior to its compilation

library a collection of useful functions and symbols that may be accessed by a program

The C program in Fig. 2.1 has two parts: preprocessor directives and the main function. The **preprocessor directives** are commands that give instructions to the C **preprocessor**, whose job it is to modify the text of a C program *before* it is compiled. A preprocessor directive begins with a number symbol (`#`) as its first nonblank character. The two most common directives appear in Fig. 2.1: `#include` and `#define`.

The C language explicitly defines only a small number of operations: Many actions that are necessary in a computer program are not defined directly by C. Instead, every C implementation contains collections of useful functions and symbols called **libraries**. The ANSI (American National Standards Institute) standard for C requires that certain *standard libraries* be provided in every ANSI C implementation. A C system may expand the number of operations available by supplying additional libraries; an individual programmer can also create libraries of functions. Each library has a standard header file whose name ends with the symbols `.h`.

FIGURE 2.1 C Language Elements in Miles-to-Kilometers Conversion Program



The `#include` directive gives a program access to a library. This directive causes the preprocessor to insert definitions from a standard header file into a program before compilation. The directive

```
#include <stdio.h>        /* printf, scanf definitions */
```

notifies the preprocessor that some names used in the program (such as `scanf` and `printf`) are found in the standard header file `<stdio.h>`.

The other preprocessor directive in Fig. 2.1

```
#define KMS_PER_MILE 1.609    /* conversion constant */
```

constant macro a name that is replaced by a particular constant value before the program is sent to the compiler

associates the **constant macro** `KMS_PER_MILE` with the meaning `1.609`. This directive instructs the preprocessor to replace each occurrence of `KMS_PER_MILE` in the text of the C program by `1.609` before compilation begins. As a result, the line

```
kms = KMS_PER_MILE * miles;
```

would read

```
kms = 1.609 * miles;
```

by the time it was sent to the C compiler. Only data values that never change (or change very rarely) should be given names using a `#define`, because an executing C program cannot change the value of a name defined as a constant macro. Using the constant macro `KMS_PER_MILE` in the text of a program for the value `1.609` makes it easier to understand and maintain the program.

comment text beginning with /* and ending with */ that provides supplementary information but is ignored by the preprocessor and compiler

The text on the right of each preprocessor directive, starting with `/*` and ending with `*/`, is a **comment**. Comments provide supplementary information making it easier for us to understand the program, but comments are ignored by the C preprocessor and compiler.

Syntax Displays for Preprocessor Directives

For each new C construct introduced in this book, we provide a syntax display that describes and explains the construct's syntax and shows examples of its use. The following syntax displays describe the two preprocessor directives. The italicized elements in each construct are discussed in the interpretation section.

#include Directive for Defining Identifiers from Standard Libraries

SYNTAX: `#include <`*standard header file*`>`

EXAMPLES:
```
#include <stdio.h>
#include <math.h>
```

INTERPRETATION: `#include` directives tell the preprocessor where to find the meanings of standard identifiers used in the program. These meanings are collected in files called *standard header files.* The header file `stdio.h` contains information about standard input and output functions such as `scanf` and `printf`. Descriptions of common mathematical functions are found in the header file `math.h`. We will investigate header files associated with other standard libraries in later chapters.

#define Directive for Creating Constant Macros

SYNTAX: `#define` *NAME value*

EXAMPLES:
```
#define MILES_PER_KM 0.62137
#define PI 3.141593
#define MAX_LENGTH 100
```

INTERPRETATION: The C preprocessor is notified that it is to replace each use of the identifier *NAME* by *value*. C program statements cannot change the value associated with *NAME*.

Function main

The two-line heading

```
int
main(void)
```

marks the beginning of the main function where program execution begins. Every C program has a main function. The remaining lines of the program form the *body* of the function, which is enclosed in braces `{ }`.

declarations the part of a program that tells the compiler the names of memory cells in a program

A function body has two parts: declarations and executable statements. The **declarations** tell the compiler what memory cells are needed in the function (for example, `miles` and `kms` in Fig. 2.1). To create this part of the function, the programmer uses the problem data requirements identified during problem analysis. The **executable statements** (derived from the algorithm) are translated into machine language and later executed.

executable statements program lines that are converted to machine language instructions and executed by the computer

The main function contains *punctuation* and *special symbols* (`*`, `=`). Commas separate items in a list, a semicolon appears at the end of several lines, and braces (`{`, `}`) mark the beginning and end of the body of function `main`.

main Function Definition

SYNTAX:
```
int
main(void)
{
      function body
}
```

(continued)

EXAMPLE:

```
int
main(void)
{
   printf("Hello world\n");
   return (0);
}
```

INTERPRETATION: Program execution begins with the main function. Braces enclose the main *function body,* which contains declarations and executable statements. The line `int` indicates that the main function returns an integer value (`0`) to the operating system when it finishes normal execution. The symbols (`void`) indicate that the main function receives no data from the operating system before it begins execution.

Reserved Words

reserved word a word that has special meaning in C

Each line of Fig. 2.1 contains a number of different words classified as **reserved words**, identifiers from standard libraries, and names for memory cells. All the reserved words appear in lowercase; they have special meaning in C and cannot be used for other purposes. A complete list of ANSI C reserved words is found in Appendix E. Table 2.1 describes the reserved words in Fig 2.1.

Standard Identifiers

standard identifier a word having special meaning but one that a programmer may redefine (but redefinition is not recommended!)

The other words in Fig. 2.1 are identifiers that come in two varieties: standard and user-defined. Like reserved words, **standard identifiers** have special meaning in C. In Fig. 2.1, the standard identifiers `printf` and `scanf` are names of operations defined in the standard input/output library. Unlike reserved words, standard identifiers can be redefined and used by the programmer for other purposes—however, we don't recommend this practice. If you redefine a standard identifier, C will no longer be able to use it for its original purpose.

TABLE 2.1 Reserved Words in Fig. 2.1

Reserved Word	Meaning
`int`	integer; indicates that the main function returns an integer value
`void`	indicates that the main function receives no data from the operating system
`double`	indicates that the memory cells store real numbers
`return`	returns control from the main function to the operating system

User-Defined Identifiers

We choose our own identifiers (called *user-defined identifiers*) to name memory cells that will hold data and program results and to name operations that we define (more on this in Chapter 3). The first user-defined identifier in Fig. 2.1, `KMS_PER_MILE`, is the name of a constant macro.

You have some freedom in selecting identifiers. The syntax rules and some valid identifiers follow. Table 2.2 shows some invalid identifiers.

1. An identifier must consist only of letters, digits, and underscores.
2. An identifier cannot begin with a digit.
3. A C reserved word cannot be used as an identifier.
4. An identifier defined in a C standard library should not be redefined.*

Valid Identifiers

```
letter_1, letter_2, inches, cent, CENT_PER_INCH, Hello, variable
```

Although the syntax rules for identifiers do not place a limit on length, some ANSI C compilers do not consider two names to be different unless there is a variation within the first 31 characters. The two identifiers

```
per_capita_meat_consumption_in_1980
per_capita_meat_consumption_in_1995
```

would be viewed as identical by a C compiler that considered only the first 31 characters to be significant.

Table 2.3 lists the category of each identifier appearing in the main function of Fig. 2.1.

TABLE 2.2 Invalid Identifiers

Invalid Identifier	Reason Invalid
`1Letter`	begins with a letter
`double`	reserved word
`int`	reserved word
`TWO*FOUR`	character * not allowed
`joe's`	character ' not allowed

*Rule 4 is actually advice from the authors rather than ANSI C syntax.

TABLE 2.3 Reserved Words and Identifiers in Fig. 2.1

Reserved Words	Standard Identifiers	User-Defined Identifiers
`int, void, double, return`	`printf, scanf`	`KMS_PER_MILE, main, miles, kms`

Uppercase and Lowercase Letters

The C programmer must take great care in the use of uppercase and lowercase letters because the C compiler considers such usage significant. The names `Rate`, `rate`, and `RATE` are viewed by the compiler as *different* identifiers. Adopting a consistent pattern in the way you use uppercase and lowercase letters is helpful to the readers of your programs. You will see that all reserved words in C and the names of all standard library functions use only lowercase letters. One style that has been widely adopted in industry uses all uppercase letters in the names of constant macros. We follow this convention in this text; for other identifiers, we use all lowercase letters.

Program Style *Choosing Identifier Names*

We discuss program style throughout the text in displays like this one. A program that "looks good" is easier to read and understand than one that is sloppy. Most programs will be examined or studied by someone other than the original programmers. In industry, programmers spend considerably more time on program maintenance (that is, updating and modifying the program) than they do on its original design or coding. A program that is neatly stated and whose meaning is clear makes everyone's job simpler.

Pick a meaningful name for a user-defined identifier, so its use is easy to understand. For example, the identifier `salary` would be a good name for a memory cell used to store a person's salary, whereas the identifier `s` or `bagel` would be a bad choice. If an identifier consists of two or more words, placing the underscore character (_) between words will improve the readability of the name (`dollars_per_hour` rather than `dollarsperhour`).

Choose identifiers long enough to convey your meaning, but avoid excessively long names because you are more likely to make a typing error in a longer name. For example, use the shorter identifier `lbs_per_sq_in` instead of the longer identifier `pounds_per_square_inch`.

If you mistype a name so that the identifier looks like the name of another memory cell, often the compiler cannot help you detect your error. For this reason and to avoid confusion, do not choose names that are similar to each other. Especially avoid

selecting two names that are different only in their use of uppercase and lowercase letters, such as `LARGE` and `large`. Also try not to use two names that differ only in the presence or absence of an underscore (`xcoord` and `x_coord`).

EXERCISES FOR SECTION 2.1

Self-Check

1. Which of the following identifiers are (a) C reserved words, (b) standard identifiers, (c) conventionally used as constant macro names, (d) other valid identifiers, and (e) invalid identifiers?

```
void        MAX_ENTRIES     double     time        G          Sue's
return      printf          xyz123     part#2      "char"     #insert
this_is_a_long_one
```

2. Why should `E (2.7182818)` be defined as a constant macro?
3. What part of a C implementation changes the text of a C program just before it is compiled? Name two directives that give instructions about these changes.
4. Why shouldn't you use a standard identifier as the name of a memory cell in a program? Can you use a reserved word instead?

2.2 Variable Declarations and Data Types

Variable Declarations

variable a name associated with a memory cell whose value can change

variable declarations statements that communicate to the compiler the names of variables in the program and the kind of information stored in each variable

The memory cells used for storing a program's input data and its computational results are called **variables** because the values stored in variables can change (and usually do) as the program executes. The **variable declarations** in a C program communicate to the C compiler the names of all variables used in a program. They also tell the compiler what kind of information will be stored in each variable and how that information will be represented in memory. The variable declarations

```
double miles; /* input - distance in miles.        */
double kms;   /* output - distance in kilometers   */
```

give the names of two variables (`miles`, `kms`) used to store real numbers. Note that C ignores the comments on the right of each line describing the usage of each variable.

A variable declaration begins with an identifier (for example, `double`) that tells the C compiler the type of data (such as a real number) stored in a particular variable. You can declare variables for any data type. C requires you to declare every variable used in a program.

Syntax Display for Declarations

SYNTAX:
```
int variable_list;
double variable_list;
char variable_list;
```

EXAMPLES:
```
int count,
    large;
double x, y, z;
char first_initial;
char ans;
```

INTERPRETATION: A memory cell is allocated for each name in the *variable_list*. The type of data (`double`, `int`, `char`) to be stored in each variable is specified at the beginning of the statement. One statement may extend over multiple lines. A single data type can appear in more than one variable declaration, so the following two declaration sections are equally acceptable ways of declaring the variables `rate`, `time`, and `age`.

```
double rate, time;
int age;
```

```
double   rate;
int      age;
double   time;
```

Data Types

data type a set of values and operations that can be performed on those values

A **data type** is a set of values and a set of operations on those values. Knowledge of the data type of an item (a variable or value) enables the C compiler to correctly specify operations on that item. A standard data type in C is a data type that is predefined, such as `char`, `double`, and `int`. We use the standard data types `double` and `int` as abstractions for the real numbers and integers (in the mathematical sense).

Objects of a data type can be variables or constants. A positive numeric constant (or number) in a C program can be written with or without a + sign. A numeric constant cannot contain a comma.

Numeric constants in C are considered nonnegative numbers. Although you can use a number like `-10500` in a program, C views the minus sign as the negation operator (applied to the positive constant `10500`) rather than as a part of the constant.

Data Type int In mathematics, integers are whole numbers. The `int` data type is used to represent integers in C. Because of the finite size of a memory cell, not all integers can be represented by type `int`. ANSI C specifies that the range of data type

`int` must include at least the values `-32767` through `32767`. You can store an integer in a type `int` variable, perform the common arithmetic operations (add, subtract, multiply, and divide), and compare two integers. Some values that you can store in a type `int` variable are

```
-10500     435     +15     -25     32767
```

Data Type double A real number has an integral part and a fractional part that are separated by a decimal point. In C, the data type `double` is used to represent real numbers (for example, `3.14159`, `0.0005`, and `150.0`). You can store a real number in a type `double` variable, perform the common arithmetic operations (add, subtract, multiply, and divide), and compare them.

We can use scientific notation to represent real numbers (usually for very large or very small values). In normal scientific notation, the real number 1.23×10^5 is equivalent to 123000.0 where the exponent 5 means "move the decimal point five places to the right." In C scientific notation, we write this number as `1.23e5` or `1.23E5`. Read the letter `e` or `E` as "times 10 to the power": `1.23e5` means 1.23 times 10 to the power 5. If the exponent has a minus sign, the decimal point is moved to the left (for example, `0.34e-4` is equivalent to `0.000034`). Table 2.4 lists some real numbers and indicates which ones can be stored in a type `double` variable. The last line shows we can write a type `double` constant in C scientific notation without a decimal point.

Data type `double` is an abstraction for the real numbers because it does not include them all. Some real numbers are too large or too small, and some real numbers cannot be represented precisely because of the finite size of a memory cell. However, we can certainly represent enough of the real numbers in C to carry out most of the computations we wish to perform with sufficient accuracy.

TABLE 2.4 Type double Constants (real numbers)

Valid double Constants	Invalid double Constants
`3.14159`	`150` (no decimal point)
`0.005`	`.12345e` (missing exponent)
`12345.0`	`15e-0.3` (`0.3` is invalid exponent)
`15.0e-04` (value is `0.0015`)	
`2.345e2` (value is `234.5`)	`12.5e.3` (`.3` is invalid exponent)
`1.15e-3` (value is `0.00115`)	`34,500.99` (comma is not allowed)
`12e+5` (value is `1200000.0`)	

FIGURE 2.2
Internal Formats of Type int and Type double

Differences Between Numeric Types

You may wonder why having more than one numeric type is necessary. Can the data type `double` be used for all numbers? Yes, but on many computers, operations involving integers are faster than those involving numbers of type `double`. Less storage space is needed to store type `int` values. Also, operations with integers are always precise, whereas some loss of accuracy or *round-off error* may occur when dealing with type `double` numbers.

These differences result from the way numbers are represented in the computer's memory. All data are represented in memory as *binary strings*, strings of 0s and 1s. However, the binary string stored for the type `int` value 13 is not the same as the binary string stored for the type `double` number 13.0. The actual internal representation is computer dependent, and type `double` numbers usually require more bytes of computer memory than type `int`. Compare the sample `int` and `double` formats shown in Fig. 2.2.

Positive integers are represented by standard binary numbers. If you are familiar with the binary number system, you know that the integer 13 is represented as the binary number 01101.

The format of type `double` values (also called *floating-point format*) is analogous to scientific notation. The storage area occupied by the number is divided into three sections: the sign (0 for positive numbers, 1 for negative numbers), the *mantissa*, and the *exponent*. The mantissa is a binary fraction between 0.5 and 1.0. The exponent is an integer. The mantissa and exponent are chosen so that the following formula is correct.

$$real\ number = mantissa \times 2^{exponent}$$

If 64 bits are used for storage of a type `double` number, the sign would occupy 1 bit, the exponent 11 bits, and the mantissa 52 bits. Because of the finite size of a memory cell, not all real numbers in the range allowed can be represented precisely as type `double` values. We will discuss this concept later.

We have seen that type `double` values may include a fractional part, whereas type `int` values cannot. An additional advantage of the type `double` format is that a much larger range of numbers can be represented as compared to type `int`. Actual ranges vary from one implementation to another, but the ANSI standard for C specifies that the minimum range of positive values of type `int` is from 1 to 32,767 (approximately 3.3×10^4). The minimum range specified for positive values

TABLE 2.5 Integer Types in C

Type	Range in Typical Implementation
`short`	–32,767 .. 32,767
`unsigned short`	0 .. 65,535
`int`	–2,147,483,647 .. 2,147,483,647
`unsigned`	0 .. 4,294,967,295
`long`	–2,147,483,647 .. 2,147,483,647
`unsigned long`	0 .. 4,294,967,295

of type `double` is from 10^{-37} to 10^{37}. To understand how small 10^{-37} is, consider the fact that the mass of one electron is approximately 10^{-27} grams, and 10^{-37} is one ten-billionth of 10^{-27}. The enormity of 10^{37} may be clearer when you realize that if you multiply the diameter of the Milky Way galaxy in kilometers by a trillion, your numeric result is just one ten-thousandth of 10^{37}.

ANSI C provides several integer data types in addition to `int`. Table 2.5 lists these types along with their ranges in a typical C implementation (`short <= int <= long`). Notice that the largest number represented by an `unsigned` integer type is about twice the magnitude of the largest value in the corresponding `signed` type. This results from using the sign bit as part of the number's magnitude.

Similarly, ANSI C defines three floating-point types that differ in their memory requirements: `float`, `double`, and `long double`. Values of type `float` must have at least six decimal digits of precision; both type `double` and `long double` values must have at least ten decimal digits. Table 2.6 lists the range of positive numbers representable by each of these types in a typical C implementation.

Data Type `char`

Data type `char` represents an individual character value—a letter, a digit, or a special symbol. Each type `char` value is enclosed in apostrophes (single quotes) as shown here.

```
'A'   'z'   '2'   '9'   '*'   ':'   '"'   ' '
```

TABLE 2.6 Floating-Point Types in C

Type	Approximate Range*	Significant Digits*
`float`	10^{-37} .. 10^{38}	6
`double`	10^{-307} .. 10^{308}	15
`long double`	10^{-4931} .. 10^{4932}	19

*In a typical C implementation

In the line above Table 2.6, the character value `'"'` represents the character `"`; the character value `' '` represents the blank character, which is typed by pressing the apostrophe key, the space bar, and the apostrophe key.

Although a type `char` value in a program requires apostrophes, a type `char` data value should not have them. Thus, for example, when entering the letter `z` as a character data item to be read by a program, press the `z` key instead of the sequence `'z'`.

The ASCII Code

ASCII code a particular code that specifies the integer representing each `char` value

You should know that a character is represented in memory as an integer. The value stored is determined by the code used by your C compiler. The **ASCII code** (American Standard Code for Information Interchange) is the most common. Table 2.7 shows the ASCII (pronounced "askey") code values for several characters. Appendix D shows the complete ASCII code.

The digit characters `'0'` through `'9'` have code values of 48 through 57 (decimal). The order relationship that follows holds for the digit characters (i.e., `'0' < '1'`, `'1' < '2'`, and so on).

```
'0' < '1' < '2' < '3' < '4' < '5' < '6' < '7' < '8' < '9'
```

In ASCII, uppercase letters have the decimal code values 65 through 90. The order relationship that follows holds for uppercase letters.

```
'A' < 'B' < 'C' < ... < 'X' < 'Y' < 'Z'
```

Lowercase letters have the consecutive decimal code values 97 through 122, and the following order relationship holds:

```
'a' < 'b' < 'c' < ... < 'x' < 'y' < 'z'
```

TABLE 2.7 ASCII Codes for Characters

Character	ASCII Code
`' '`	32
`'*'`	42
`'A'`	65
`'B'`	66
`'Z'`	90
`'a'`	97
`'b'`	98
`'z'`	122
`'0'`	48
`'9'`	57

In ASCII, the *printable characters* have codes from 32 (code for a blank or space) to 126 (code for the symbol ~). The other codes represent nonprintable *control characters.* Sending a control character to an output device causes the device to perform a special operation such as returning the cursor to column one, advancing the cursor to the next line, or ringing a bell.

EXERCISES FOR SECTION 2.2

Self-Check

1. a. Write the following numbers in normal decimal notation:

   ```
   103e-4    1.2345e+6    123.45e+3
   ```

   b. Write the following numbers in C scientific notation:

   ```
   1300    123.45    0.00426
   ```

2. Indicate which of the following are valid type `int`, `double`, or `char` constants in C and which are not. Identify the data type of each valid constant.

   ```
   'PQR'    15E-2    35    'h'    -37.491    .912    4,719    'true'    "T"
   & 4.5e3    '$'
   ```

3. What would be the best variable type for the area of a circle in square inches? Which type for the number of cars passing through an intersection in an hour? The first letter of your last name?

Programming

1. Write the `#define` preprocessor directive and declarations for a program that has a constant macro for `PI` (3.14159) and variables `radius`, `area`, and `circumf` declared as `double`, variable `num_circ` as an `int`, and variable `circ_name` as a `char`.

2.3 Executable Statements

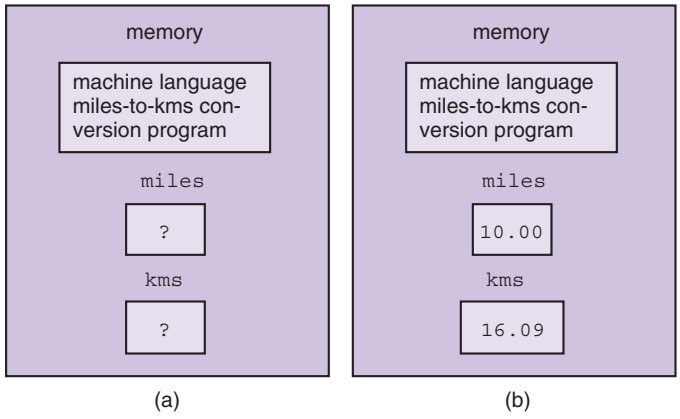
The executable statements follow the declarations in a function. They are the C statements used to write or code the algorithm and its refinements. The C compiler translates the executable statements into machine language; the computer executes the machine language version of these statements when we run the program.

Programs in Memory

Before examining the executable statements in the miles-to-kilometers conversion program (Fig. 2.1), let's see what computer memory looks like before and

FIGURE 2.3

Memory (a) Before and (b) After Execution of a Program



after that program executes. Figure 2.3a shows the program loaded into memory and the program memory area before the program executes. The question marks in memory cells `miles` and `kms` indicate that the values of these cells are undefined before program execution begins. During program execution, the data value `10.00` is copied from the input device into the variable `miles`. After the program executes, the variables are defined as shown in Fig. 2.3b. We will see why next.

Assignment Statements

assignment statement an instruction that stores a value or a computational result in a variable

An **assignment statement** stores a value or a computational result in a variable, and is used to perform most arithmetic operations in a program. The assignment statement

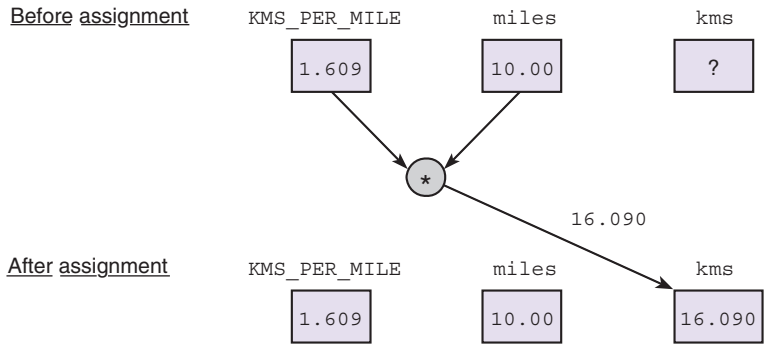
```
kms = KMS_PER_MILE * miles;
```

assigns a value to the variable `kms`. The value assigned is the result of the multiplication (`*` means multiply) of the constant macro `KMS_PER_MILE` (1.609) by the variable `miles`. The memory cell for `miles` must contain valid information (in this case, a real number) before the assignment statement is executed. Figure 2.4 shows the contents of memory before and after the assignment statement executes; only the value of `kms` is changed.

In C the symbol = is the assignment operator. Read it as "becomes," "gets," or "takes the value of" rather than "equals" because it is *not* equivalent to the equal sign of mathematics. In mathematics, this symbol states a relationship between two values, but in C it represents an action to be carried out by the computer.

FIGURE 2.4

Effect of `kms = KMS_PER_MILE * miles;`



Assignment Statement

FORM: *variable* = *expression*;

EXAMPLE: `x = y + z + 2.0;`

INTERPRETATION: The *variable* before the assignment operator is assigned the value of the *expression* after it. The previous value of *variable* is destroyed. The *expression* can be a variable, a constant, or a combination of these connected by appropriate operators (for example, `+`, `-`, `/`, and `*`).

EXAMPLE 2.1

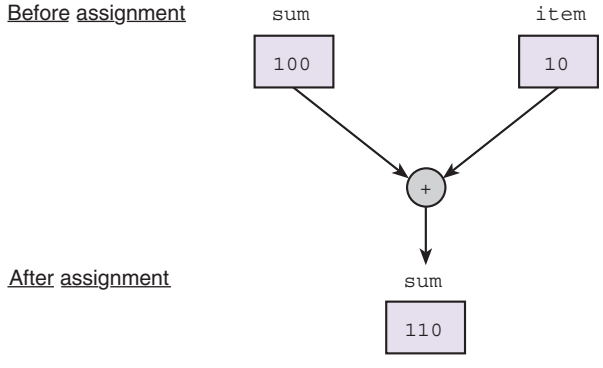
In C you can write assignment statements of the form

```
sum = sum + item;
```

where the variable `sum` appears on both sides of the assignment operator. This is obviously not an algebraic equation, but it illustrates a common programming practice. This statement instructs the computer to add the current value of `sum` to the value of `item`; the result is then stored back into `sum`. The previous value of `sum` is destroyed in the process, as illustrated in Fig. 2.5. The value of `item`, however, is unchanged.

FIGURE 2.5

Effect of `sum = sum + item;`



EXAMPLE 2.2

You can also write assignment statements that assign the value of a single variable or constant to a variable. If `x` and `new_x` are type `double` variables, the statement

```
new_x = x;
```

copies the value of variable `x` into variable `new_x`. The statement

```
new_x = -x;
```

instructs the computer to get the value of `x`, negate that value, and store the result in `new_x`. For example, if `x is 3.5`, `new_x is -3.5`. Neither of the assignment statements above changes the value of `x`.

Section 2.5 continues the discussion of type `int` and `double` expressions and operators.

Assignment to a `char` Variable

The `char` variable `next_letter` is assigned the character value `'A'` by the assignment statement

```
next_letter = 'A';
```

A single character variable or value may appear on the right-hand side of a character assignment statement.

Input/Output Operations and Functions

Data can be stored in memory in two different ways: either by assignment to a variable or by copying the data from an input device into a variable using a function like `scanf`. You copy data into a variable if you want a program to manipulate different data each time it executes. This data transfer from the outside world into memory is called an **input operation**.

input operation an instruction that copies data from an input device into memory

As it executes, a program performs computations and stores the results in memory. These program results can be displayed to the program user by an **output operation**.

output operation an instruction that displays information stored in memory

All input/output operations in C are performed by special program units called **input/output functions**. The most common input/output functions are supplied as part of the C standard input/output library to which we gain access through the preprocessor directive

input/output function a C function that performs an input or output operation

```
#include <stdio.h>
```

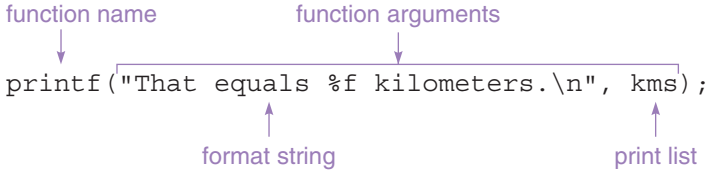
In this section, we show how to use the input function `scanf` and the output function `printf`.

function call calling or activating a function

In C a **function call** is used to call or activate a function. Calling a function is analogous to asking a friend to perform an urgent task. You tell your friend what to do (but not how to do it) and wait for your friend to report back that the task is finished. After hearing from your friend, you can go on and do something else.

The printf Function

To see the results of a program execution, we must have a way to specify what variable values should be displayed. In Fig. 2.1, the statement



function argument enclosed in parentheses following the function name; provides information needed by the function

format string in a call to `printf`, a string of characters enclosed in quotes (`"`), which specifies the form of the output line

print list in a call to `printf`, the variables or expressions whose values are displayed

placeholder a symbol beginning with % in a format string that indicates where to display the output value

calls function `printf` (pronounced "print-eff") to display a line of program output. A function call consists of two parts: the function name and the **function arguments**, enclosed in parentheses. The arguments for `printf` consist of a **format string** (in quotes) and a **print list** (the variable `kms`). The function call above displays the line

```
That equals 16.090000 kilometers.
```

which is the result of displaying the format string `"That equals %f kilometers.\n"` after substituting the value of `kms` for its placeholder (`%f`) in the format string. A **placeholder** always begins with the symbol `%`. Here the placeholder `%f` marks the display position for a type `double` variable.

Table 2.8 shows placeholders for type `char`, `double`, and `int` variables. Each placeholder is an abbreviation for the type of data it represents. C uses `%f` (or `%lf`) and not `%d` with type `double` because programmers often refer to real numbers as *floating-point numbers*.

The placeholders used with `scanf` are the same as those used with `printf` except for variables of type `double`. Type `double` variables use a `%f` placeholder in a `printf` format string and a `%lf` placeholder in a `scanf` format string.

newline escape sequence the character sequence \n, which is used in a format string to terminate an output line

The format string shown above also contains the **newline escape sequence** `\n`. Like all C escape sequences, `\n` begins with the backslash character. Including this sequence at the end of the format string terminates the current output line.

Multiple Placeholders Format strings can have multiple placeholders. If the print list of a `printf` call has several variables, the format string should contain the same number of placeholders. C matches variables with placeholders in left-to-right order.

TABLE 2.8 Placeholders in Format Strings

Placeholder	Variable Type	Function Use
`%c`	`char`	`printf/scanf`
`%d`	`int`	`printf/scanf`
`%f`	`double`	`printf`
`%lf`	`double`	`scanf`

EXAMPLE 2.3

If `letter_1`, `letter_2`, and `letter_3` are type `char` variables and `age` is type `int`, the `printf` call

```
printf("Hi %c%c%c - your age is %d\n",
       letter_1, letter_2, letter_3, age);
```

displays a line such as

```
Hi EBK - your age is 35
```

The placeholders `%c%c%c` indicate the display position of the letters (`E`, `B`, and `K`) stored in the three type `char` variables, and the placeholder `%d` indicates the position of the value of `age (35)`.

Syntax Display for printf Function Call

SYNTAX: `printf(`*format string, print list*`);`
`printf(`*format string*`);`

EXAMPLES:
```
printf("I am %d years old, and my gpa is %f\n",
       age, gpa);
printf("Enter the object mass in grams> ");
```

INTERPRETATION: The `printf` function displays the value of its *format string* after substituting in left-to-right order the values of the expressions in the *print list* for their placeholders in the *format string* and after replacing escape sequences such as `\n` by their meanings.

cursor a moving place marker that indicates the next position on the screen where information will be displayed

More About \n The **cursor** is a moving place marker that indicates the next position on the screen where information will be displayed. When executing a `printf` function call, the cursor is advanced to the start of the next line on the screen if the `\n` escape sequence is encountered in the format string.

We often end a `printf` format string with a `\n` (newline escape sequence) so that the call to `printf` produces a completed line of output. If no characters are